TOBB ETÜ

Ekonomi ve Teknoloji Üniversitesi

ROADRUNNER

Low Level Design Report

- Utku Ceylan
- Mustafa Tufan
- Ahmet Salih Kaya
- Batuhan Efe Alkış
- Doruk Demirci

TABLE OF CONTENTS

1. Introduction

In the current digital landscape, travel planning remains a disjointed and labor-intensive process. A typical traveler is forced to navigate a distributed ecosystem of unconnected websites and applications using one platform to research destinations, another to compare transport options, and a third to organize daily activities. This fragmentation results in significant complexity and information overload, as users must navigate through massive amounts of generic content, advertisements, and irrelevant sponsored material. Furthermore, existing tools rely on rigid, generic filters (such as price, star rating) that fail to capture nuanced traveler preferences such as "quiet cafes," "historical battlefields," or specific dietary needs.

To address these challenges, we propose **"RoadRunner,"** an AI-powered travel assistant designed to centralize and personalize the entire itinerary planning lifecycle. Unlike traditional search engines, RoadRunner utilizes Large Language Models (LLMs) and Natural Language Processing (NLP) to transform unstructured "User Stories" conversational descriptions of a user's desires into optimized, executable travel plans. The system acts as a smart intermediary, aggregating data from third-party sources (such as Wikipedia, Google Places, and OpenStreetMap) to provide a unified, stress-free planning experience. Each user's likings and preferences are stored in the database and are continuously updated as user makes comments on the generated trip routes. The dynamic, AI based and user centric system aims to unify the travel planning system in a single environment and create a travel plan for each user's own preferences.

This report illustrates the low-level architecture and design of RoadRunner. The report includes Object design trade-offs, Engineering standards, Packages, and Class interfaces sections, concluding with diagrams and explanations of software components. Related Glossary and References sections are also given in the end.

1.1. Object design trade-offs

1.1.1. Usability vs. Functionality

In RoadRunner, we place the usability aspect of the integrated platform above all other metrics, in order to achieve it we have built a conversational interface in addition to a 'traditional button using' interface. We aim to hide the immense complexity of travel logistics (coordinate mapping, time-window constraints, and routing algorithms) behind a simple chat window. While the backend supports complex parameter tuning (such as specific weight overrides for "historical" vs. "nature" sites), exposing all these controls would clutter the UI and overwhelm the user. Therefore, we prioritize a clean, intuitive chat interface over exposing every possible

configuration knob to the end-user, relying on the AI to infer these functional parameters from natural language.

1.1.2. Efficiency vs. Accuracy

One of our core features is generating multi-stop itineraries that respect time and budget constraints. The "Traveling Salesperson Problem" inherent in route optimization is computationally expensive to solve perfectly. Additionally, waiting for a Large Language Model (LLM) to generate a "perfect" response can introduce high latency. To ensure a responsive user experience, we prioritize efficiency by using heuristic routing strategies (via OSRM) and caching mechanisms. We accept a near-optimal route that is generated in seconds over a mathematically perfect route that takes minutes to compute.

1.1.3. Security vs. Usability

While seamless access is critical for a travel companion, we enforce strict security boundaries to protect user data. For example, the system utilizes token-based authentication (JWT) and persistent sessions. We determined that sessions must eventually expire to prevent unauthorized access on shared devices, even though this requires the user to occasionally re-authenticate. This trade-off ensures that personal travel history and preferences remain secure, even if it slightly reduces the convenience of indefinite "always-on" access.

1.1.4. Reliability vs. Compatibility

RoadRunner is built on a microservices-inspired architecture to ensure high reliability and scalability. Rather than attempting to support every legacy browser or operating system, we focus on modern web standards to ensure the reliability of our complex interactive maps (Leaflet) and real-time chat components. By limiting compatibility to modern, standards-compliant environments, we reduce the risk of client-side crashes and ensure a robust experience for the majority of users.

1.2. Interface Documentation Guidelines

UML diagrams are given and each class is explained in the paragraphs below. The class diagrams are further illustrated by the use of ‘blue and white’ model attribute and method description tables. All classes, attributes and methods are named in the camelCase format except for the LLM Service package, where snake_case is used instead. Class interface descriptions for Java classes are given in the following format:

class ClassName
Attributes
typeOfAttribute nameOfAttribute
Methods
scope returnType nameOfMethod(parameters)

For types in TypeScript, the below format is used:

type TypeName
Attributes
typeOfAttribute nameOfAttribute

For React classes, the below format is used:

class ClassName
Props
typeOfProp nameOfProp
State
typeOfState nameOfState
Methods
scope returnType nameOfMethod(parameters)

1.3. Engineering Standards

This report uses UML as the primary engineering notation to express both structural and behavioral aspects of the system. Class diagrams define responsibilities, attributes, and method contracts for server and client modules; sequence diagrams capture request–response flows and internal call order; and use-case diagrams describe externally visible user goals and system-provided capabilities. To keep the diagrams reproducible and consistent across

revisions, the UML content is maintained in a text-based form so that changes can be tracked through version control and diagram drift is avoided when the implementation evolves.

In addition, the report follows IEEE conventions for technical reporting and referencing. Sources are cited using the IEEE citation style, and the document structure is organized to mirror an engineering design deliverable: package-level decomposition, explicit class interface sections, and use-case coverage are presented as separate, traceable parts of the specification. Beyond documentation format, the system interfaces and data exchange are specified using standard, implementation-friendly conventions: REST-like HTTP endpoints with JSON payloads, stable field naming rules, and consistent time/coordinate representations. These conventions reduce ambiguity between teams and ensure that the design can be implemented and tested with predictable contracts.

1.4. Definitions, Acronyms, and Abbreviations

Tool_calling: using pretested and determined static function templates and having the AI model decide which function with which parameters should be invoked.

userVector: A key–value map received from the frontend that encodes a user's trip-specific answers and derived preference signals for the current planning session (e.g., pace/tempo, social preference, interest priorities, budget/time constraints).

overrideWeights: A weight map used to bias category/type selection during route generation (e.g., higher weight → higher probability or count allocation for that POI type).

mandatoryTypes: A set of POI types/categories that must appear in the generated route (hard constraint).

Reroll (indexed POI): The operation of replacing exactly one POI at a specified route index while keeping all other POIs fixed (locked) and recomputing only the affected travel segments and totals.

Feasible route: A route whose totals and composition satisfy TripRequest constraints (e.g., time/budget/stop count and mandatory types).

AI: Artificial Intelligence

API: Application Programming Interface

HTTP: Hypertext Transfer Protocol

IEEE: Institute of Electrical and Electronics Engineers

JSON: JavaScript Object Notation

LLM: Large Language Model

NLP: Natural Language Processing

OSRM: Open Source Routing Machine

POI: Point of Interest

UI: User Interface

UML: Unified Modeling Language

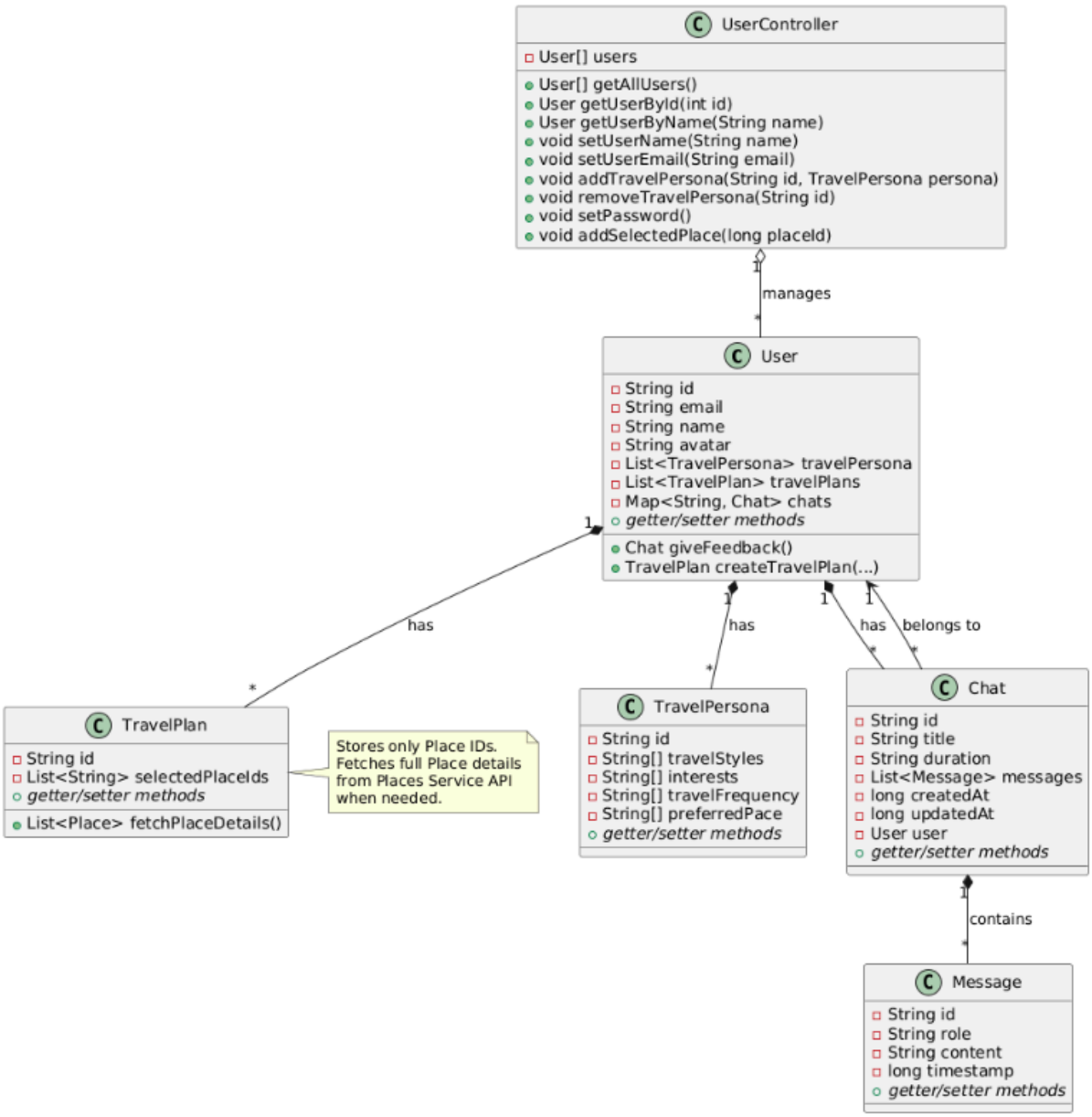
URL: Uniform Resource Locator

XAI: Explainable Artificial Intelligence

2. Packages

2.1. Server

2.1.1. User



User: This class represents a registered user account in the travel planning system. It stores core identity attributes and maintains collections of the user's travel profiles, generated plans, and chat interactions through Map structures keyed by unique identifiers. The travelPersona map allows a single user to define multiple preference profiles for different trip contexts, while travelPlans stores IDs of all itineraries the user has created. The user owns the lifecycle of its associated Chats through composition. User provides domain methods for feedback submission (giveFeedback) and plan creation (createTravelPlan).

TravelPersona: This class represents a reusable travel preference profile that defines how a user wants to travel. It stores arrays of categorical preferences including travel styles, specific interests, travel frequency patterns, and preferred activity pacing. TravelPersona is designed as an entity that can be stored in User's travelPersona map, allowing users to maintain multiple distinct personas without recreating preferences for each planning session. During trip planning, the selected TravelPersona's attributes inform recommendation algorithms and constraint generation, ensuring generated itineraries align with the user's specified travel style and interests.

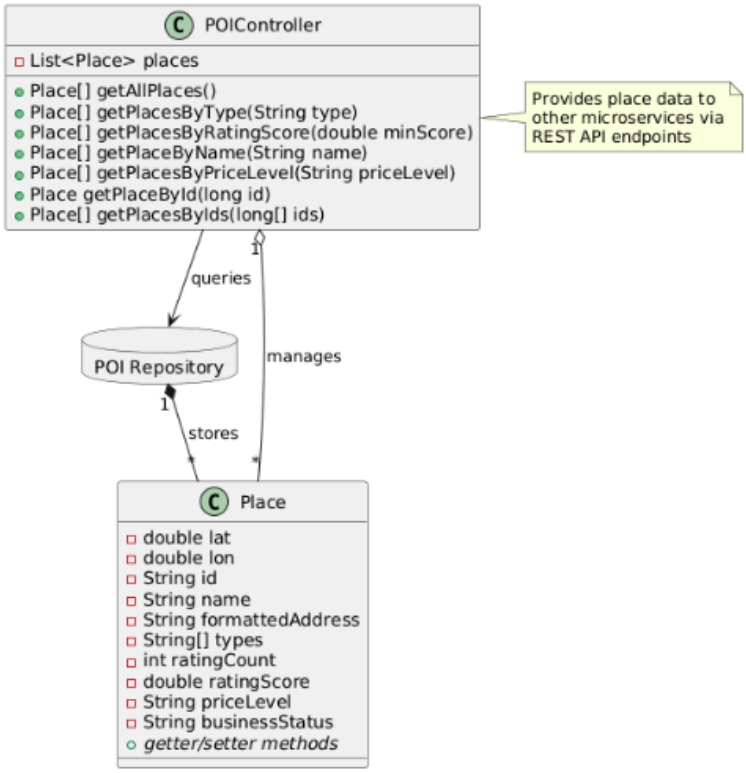
TravelPlan: This class represents a generated or user-curated travel itinerary. It maintains a list of selected places IDs that constitute the planned visits for a trip. TravelPlan serves as a container for places chosen either through automated route generation or manual user selection, and can be persisted in the User's travelPlans map for future reference and modification.

UserController: This class implements the service layer for user management operations in the Spring Boot application. It maintains a collection of all User instances and exposes methods for user CRUD operations including retrieval by ID or name, profile updates (setUserName, setUserEmail, setPassword), and travel persona management (addTravelPersona, removeTravelPersona). UserController also provides place selection functionality (addSelectedPlace). As a controller, this class handles the orchestration logic between HTTP requests and domain model operations, managing the users collection while delegating entity-specific behavior to User instances.

Chat: This class represents a conversation session between a user and the system's AI assistant. It stores conversation metadata and maintains an ordered list of Messages exchanged during the session. The messages list preserves chronological ordering of the conversation, supporting features like context retrieval, conversation history display, and multi-turn dialogue management.

Message: This class represents a single message exchange within a Chat conversation. It stores the message content along with metadata including unique identifier, sender role classification, and timestamp for chronological ordering. Message is an entity that captures one turn in the conversation. The role field enables the system to distinguish between different message sources for rendering, access control, and conversation flow analysis. Messages are owned by their parent Chat and are typically created in chronological sequence, forming the complete conversation transcript when aggregated.

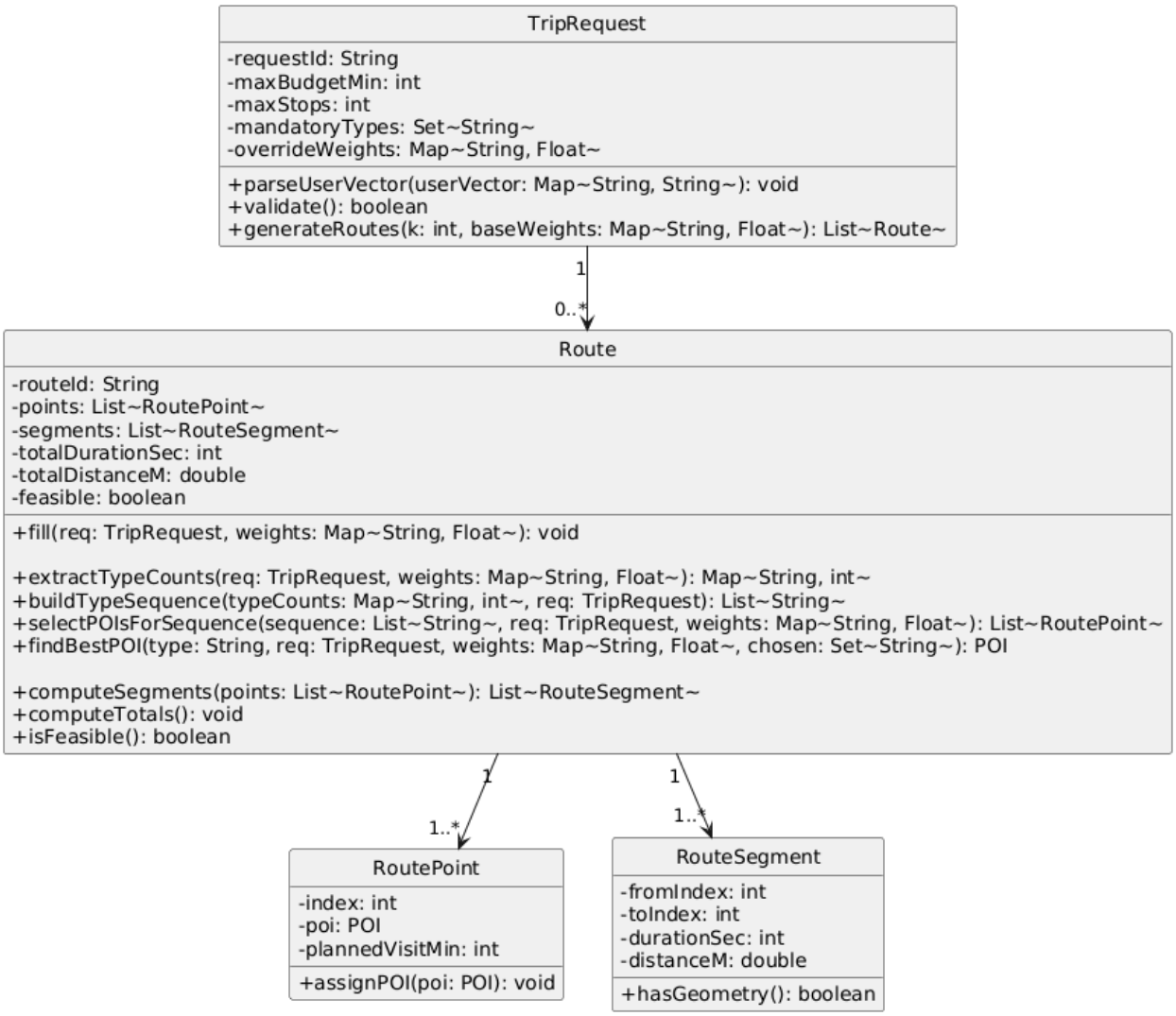
2.1.2. POI (Places)



Place: This class represents a Point of Interest entity in the Places Service microservice, serving as the single source of truth for all place-related data in the system. Geographic coordinates are stored directly in the Place record, aligned with a single-table database design optimized for spatial queries. Place contains attributes extracted from Google Places API including name, formattedAddress for display purposes, a types array for multi-category classification, rating metrics, pricing information, and operational status. Place data is immutable from the perspective of other microservices. Only the Places Service can modify Place records, typically through periodic synchronization with external APIs. Other services, such as User Service, reference places exclusively by ID and retrieve full details through REST API calls.

POIController: This class implements the service layer for place query and retrieval operations in the Places Service microservice. It maintains access to the places collection through the POI Repository and exposes a set of REST API endpoints for various search, filter, and lookup operations. POIController provides fundamental query methods including getAllPlaces() for bulk retrieval with pagination support, getPlacesByType() for category-based filtering using the types array, getPlacesByRatingScore() for quality-based filtering with minimum threshold support, getPlaceByName() for text-based search with fuzzy matching capabilities, and getPlacesByPriceLevel() for budget-constraint filtering.

2.1.3. Route Generation



TripRequest: This class represents a single trip-planning request constructed from the userVector received from the frontend. It stores session-specific constraints and parameters, including numeric limits such as maxBudgetMin and maxStops, mandatory category requirements (mandatoryTypes), and the preference weight map (overrideWeights) derived from the userVector. TripRequest is not a persistent user profile; it is a snapshot that can differ across planning sessions even for the same user. TripRequest is responsible for parsing the incoming userVector into these internal fields and for initiating route generation so that all produced routes conform to the same set of constraints and weights.

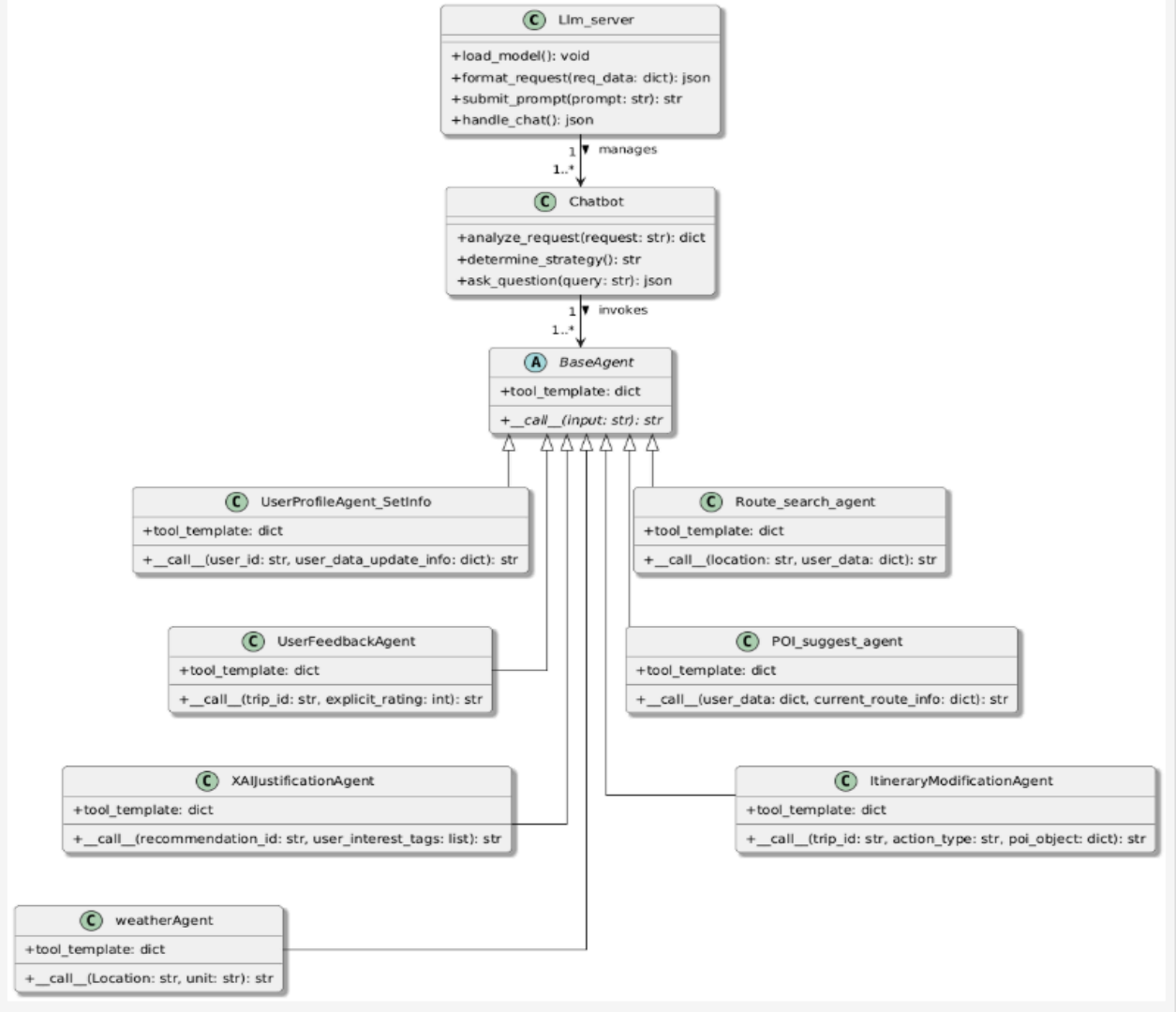
Route: This class represents a fully generated itinerary candidate for a given TripRequest. It contains the ordered visit list (points) and the travel segments between consecutive visits (segments). Aggregated evaluation values such as total duration, total distance, and feasibility are stored directly as fields in Route. Route performs the internal planning steps as methods: it

derives a type distribution and produces an ordered type sequence based on TripRequest weights and constraints, selects concrete POIs for each position in that sequence, and computes segments and totals using routing calculations. A single TripRequest may produce multiple Route instances to provide alternative itineraries.

RoutePoint: This class represents one ordered visit item in a Route. Each RoutePoint references exactly one Place from Place Class in POI Package and preserves ordering while storing per-visit planning metadata without modifying the Place entity itself. RoutePoint includes fields such as visit index and planned visit duration, and it provides a structured place to attach schedule-related information if scheduling is extended in later iterations.

RouteSegment: This class represents the travel segment between two consecutive RoutePoints in a Route. It stores computed travel attributes such as duration and distance, and it can optionally carry path geometry for map rendering. RouteSegment enables the system to compute overall travel cost, support map visualization, and retain edge-level travel information needed to evaluate feasibility against TripRequest limits.

2.1.4. LLM Agent



This class is responsible for the AI generated responses to user queries or internal tasks.

The package makes use of ‘ai_agents’ where each AI agent is a template of a function that is invoked as necessary. The chatbot dynamically decides on **if** or **which** ai_agent should be invoked with **which** parameters. This approach allows mixing ‘statically’ testable and ‘dynamically’ invokable agents to be used. In a way, this is essentially ‘tool calling’.

LLM Server: This class is responsible for managing communication with the Large Language Model (LLM). It encapsulates all low-level details related to model loading, configuration, request formatting, and response handling. The LLM Server exposes a clean interface for submitting prompts and receiving generated outputs, abstracting away deployment-specific details (e.g., local models or tool call templates).

By isolating LLM interaction logic, this class improves modularity, maintainability, and allows the underlying model to be replaced without affecting higher-level components.

Chatbot: This class acts as the central orchestration layer for AI-generated responses to user queries and internal system tasks. It analyzes incoming requests, determines the appropriate processing strategy, and decides whether to respond directly using the LLM or to invoke one or more specialized AI agents.

BaseAgent: This abstract base class defines the standardized interface (tool_template and __call__) for all specific AI tools, ensuring that the Chatbot can discover, validate, and invoke any agent via the Llm_server regardless of its underlying logic.

UserProfileAgent_SetInfo: This agent handles write operations for the User Profile Service, dynamically updating the persistent database with new preferences or constraints discovered during the conversation to ensure the system evolves with the user. This agent is responsible for sending necessary ‘action’ information to the frontend.

UserFeedbackAgent: This agent implements the system’s adaptive learning mechanism by processing explicit ratings on trips or POIs, mathematically adjusting the weights of specific interest tags in the user's profile to refine the accuracy of future recommendations.

Route_search_agent: This agent interfaces with the routing logic to calculate optimal travel paths and select appropriate transport modes between specific vertices, ensuring the physical feasibility of moving between locations within the generated schedule.

POI_suggest_agent: This agent interfaces with the routing logic to calculate optimal travel paths and select appropriate transport modes between specific vertices, given the user input. This agent is responsible for generation of one or more points on the route, rather than creating a totally new route this agent updates some nodes to better fit user likings.

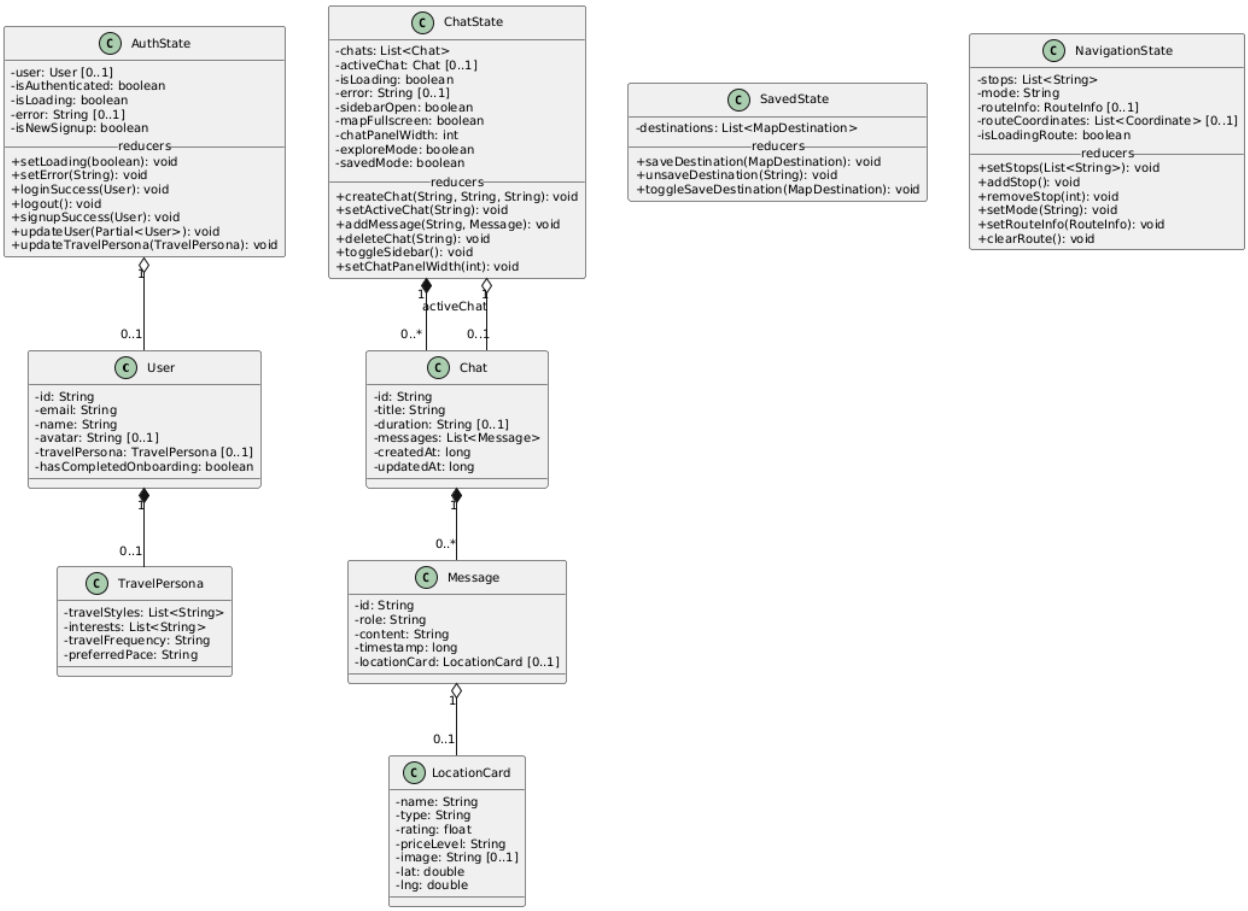
ItineraryModificationAgent: This agent provides the functional capability for the Chatbot to directly manipulate the active map state, executing "write" actions such as adding, removing, or reordering POIs in the serialized itinerary upon user confirmation.

weatherAgent: This agent retrieves current or forecasted meteorological data for specific locations and dates, providing essential environmental context that allows the system to optimize activity scheduling.

XAIJustificationAgent: This agent fulfills the ethical requirement for Explainable AI (XAI) by generating text-based justifications for specific recommendations, explicitly linking suggested POIs to user preferences to build trust and transparency.

2.2. Client

2.2.1. State Management



AuthState: This class represents the authentication and user identity state of the application. It holds the currently logged-in User object, authentication status flags (isAuthenticated, isLoading), error messages, and a flag indicating whether the user has just completed signup. AuthState is responsible for persisting user session data to user repository and restoring it on application load, enabling session continuity across browser refreshes. By centralizing authentication logic in a single state slice, the client ensures that all UI components can consistently read and react to authentication changes without duplicating login/logout logic.

User: This class represents the authenticated end user on the client side. It stores the user's unique identifier, email, display name, optional avatar URL, travel persona preferences, and onboarding completion status. User is the primary identity model consumed by all client-side components that need to personalize the experience, such as displaying the user's name in the

header, routing new signups to onboarding, or tailoring chat context with the user's travel preferences.

TravelPersona: This class represents the user's travel preference profile, captured during the onboarding process. It stores arrays of preferred travel styles (adventure, cultural, relaxation) and interests (history, food, nature), along with travel frequency and preferred pace. TravelPersona enables the system to provide personalized recommendations from the very first interaction by giving the LLM and route generation modules an understanding of the user's general travel tendencies before any explicit trip request is made.

ChatState: This class represents the global state governing the chat interface and its associated view modes. It maintains the list of all chat sessions, the currently active chat, loading and error states, sidebar visibility, map fullscreen toggle, resizable panel width, and mode flags for switching between Chat, Explore, and Saved views. ChatState also tracks saved destination IDs for backward-compatible bookmark functionality. By centralizing these UI concerns in a single Redux slice, the application ensures that view transitions (switching from Explore mode to Chat mode) are atomic and consistent, preventing conflicting UI states.

Chat: This class represents a single conversation session between the user and the AI assistant. It contains a unique identifier, a human-readable title (auto-generated from the first query), an optional trip duration label, the ordered list of messages exchanged, and creation/update timestamps. Chat instances are persisted to a database hosted on servers so that users can resume previous conversations across sessions. The design of maintaining multiple independent Chat objects allows the user to have separate planning conversations for different trips without mixing context.

Message: This class represents a single message within a Chat conversation. It stores a unique identifier, the sender role (user or assistant), the text content, a timestamp, and an optional LocationCard attachment for place recommendations. Message is the main unit of the conversation history and is used both for rendering the chat UI and for constructing the context window sent to the LLM. The optional LocationCard field enables interactive content within the chat stream without requiring a separate message type.

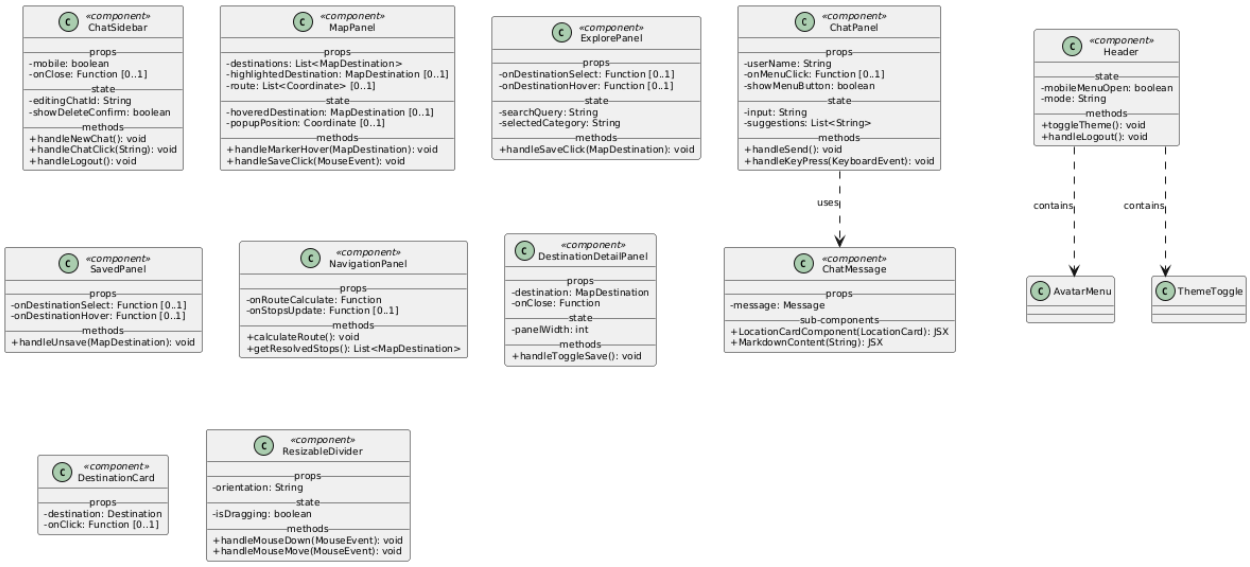
LocationCard: This class represents an embedded place recommendation card within a chat message. It stores the place name, type/category, rating, price level, optional image URL, and geographic coordinates. LocationCard enables the AI assistant's responses to include structured POI cards inline with text responses. This way, users can interact with both conversational output and actionable travel data (view on map, save to bookmarks).

SavedState: This class represents the collection of destinations that the user has bookmarked for future reference. It stores an array of MapDestination objects and provides actions for saving, unsaving, and toggling the saved status of individual destinations. SavedState persists to a persistent database ensuring bookmarks survive page refreshes. By maintaining a state slice for saved destinations, the system separates bookmark management from the chat and navigation concerns, allowing multiple UI surfaces (map popups, explore panel, destination detail panel) to independently read and modify the saved collection.

NavigationState: This class represents the state of the turn-by-turn navigation planning interface. It manages an ordered list of stops (destination IDs or null for empty slots), the selected transportation mode (driving or walking), computed route information (duration and

distance), the road-following route coordinates returned from the routing service, and a loading flag for asynchronous route computation. NavigationState supports dynamic stop management (adding, removing, reordering up to 10 stops) when any input changes, ensuring that displayed route metrics always match the current configuration.

2.2.2. UI Components



ChatPanel: This component is the primary conversational interface where users interact with the AI travel assistant. It renders the message history of the active chat, provides a text input with send functionality, and manages the send-receive cycle with the LLM service. ChatPanel handles both existing chat sessions and the new-chat creation flow, automatically generating chat titles and dispatching messages to the Redux store. It may also call agent functions from the LLM server.

ChatMessage: This component renders a single message bubble within the chat panel. It differentiates between user messages (right-aligned, primary-colored bubbles) and assistant messages (left-aligned, with an AI avatar icon). For assistant messages, it renders the content through a Markdown processor with support for tables, code blocks, lists, blockquotes, and other rich formatting. When a message includes an attached LocationCard, ChatMessage renders an interactive LocationCardComponent below the text, displaying the place's image, name, rating, and type.

ChatSidebar: This component provides the navigation sidebar visible on the chat interface. It displays the list of past chat sessions with options to rename or delete them, navigation links to Explore, Saved, and Navigation views, a theme toggle, and user account actions (profile, settings, logout). ChatSidebar supports both desktop and mobile layouts and manages routing transitions between different application views.

MapPanel: This component renders the interactive Leaflet-based map displaying destination markers on a map. It supports custom-colored markers with numeric labels for ordered stops, hover-based popup cards showing destination details (image, name, rating, price, save button), route rendering for navigation paths, and fullscreen toggle. MapPanel integrates with SavedState for bookmark toggling from map popups, and communicates destination selection and hover events to parent components.

ExplorePanel: This component provides a browsable catalog of all available destinations. It features a text search bar, category filter chips (Historical, Museum, Park, Culture, Nature, Landmark, Cafe), and a scrollable grid of destination cards with images, ratings, and save/bookmark toggles. ExplorePanel filters destinations client-side using the searchDestinations utility and communicates hover and selection events to the MapPanel for synchronized map-list interaction.

SavedPanel: This component displays the user's bookmarked destinations in a card grid layout. It reads from SavedState and renders each saved destination as an interactive card with image, rating, type, and an unsave button. SavedPanel provides hover and selection callbacks for map synchronization, enabling users to see their bookmarked places highlighted on the map. When no destinations are saved, it displays an empty-state message encouraging users to explore and save places.

NavigationPanel: This component provides the route planning interface where users can add, remove, and reorder stops, select a transportation mode, and compute optimized routes. It renders a dynamic list of stop selectors, add/remove stop controls, a mode selector, and a "Calculate Route" button. Upon route calculation, it displays the total duration and distance, and passes the computed route coordinates to the MapPanel for rendering via the OSRM routing service.

DestinationDetailPanel: This component renders a detailed, resizable side panel for a single selected destination. It displays the destination's image, name, location, category, rating, review count, price level, description, opening hours, phone number, website link, and a save/unsave toggle button. The panel is horizontally resizable via mouse or touch drag, with configurable minimum and maximum width constraints, allowing users to adjust the layout to their preference.

Header: This component renders the top navigation bar of the application, visible on public pages (Landing, Auth, Profile, etc.). It displays the RoadRunner logo, navigation links (Explore, Saved, Chat), a theme toggle button, and conditional authentication controls, showing Login/Sign Up buttons for unauthenticated users or an AvatarMenu for authenticated users.

Footer: This component renders the application footer with the RoadRunner branding, and necessary links. It is displayed on public pages below the main content.

AvatarMenu: This component provides a dropdown menu triggered by clicking the user's avatar in the header. It displays the user's name and email at the top, followed by navigation options for Profile and Settings, and a logout action. AvatarMenu reads the current user from AuthState and generates fallback initials when no avatar image is available.

DestinationCard: This component renders a single destination as a styled card with an image, category chip, name, location, rating, and price level indicator. It is used as the display unit in ExplorePanel and LandingPage grids, supporting click-to-select interaction and hover animations. The card uses the Destination interface for its data model.

ThemeToggle: This component provides a simple icon button that toggles between light and dark color schemes using MUI Joy's useColorScheme hook. It renders a sun icon in dark mode and a moon icon in light mode, with a rotation animation on hover.

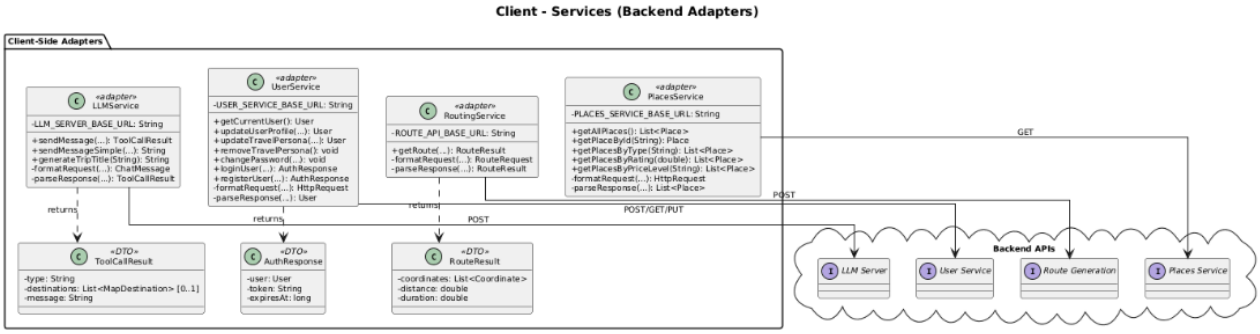
2.2.3. Data & Models

Destination: This class represents the base data model for a Point of Interest displayed on the client side. It stores the destination's unique identifier, name, display location string, image URL, rating, price level (1–4 scale), category, and optional extended metadata including description, opening hours, phone number, website, and review count. Destination serves as the shared interface consumed by DestinationCard, ExplorePanel, and all other UI components that display place information.

MapDestination: This class extends Destination with geographic coordinates (latitude, longitude) required for map display. MapDestination is the primary data model used by MapPanel for marker placement, by NavigationPanel for stop selection, and by SavedState for persisting bookmarked places with their locations. The coordinates field allows distance calculations, map centering, and integration with the RoutingService for route computation.

DestinationData: This module provides the static dataset of destinations and utility functions for client-side search and retrieval. It exports pre-defined arrays of MapDestination objects , a searchDestinations function that filters by text query and/or category, a getDestinationById function for direct lookup, and a categories array defining the available filter categories. This module serves as the local data source consumed by ExplorePanel, MapPanel, NavigationPanel, and the LLMService's tool call execution.

2.2.4. Services



LLM Service: This service acts as the client-side adapter for the backend LLM Server. It is responsible for formatting user messages and conversation history into the request structure expected by the LLM Server API, sending requests over HTTP, and parsing the server's responses back into client-side data models. When the LLM Server returns tool call results (e.g., destination search results or saved destination confirmations), LLMService transforms them into structured ToolCallResult objects containing text and optional LocationCard data that ChatPanel can render. It also exposes a utility method that requests the LLM Server to generate a concise trip title from the user's initial query.

Routing Service: This service acts as the client-side adapter for the backend Route Generation package. It accepts an array of geographic waypoints and a transportation mode (driving or walking) from the NavigationPanel, forwards the request to the Route Generation API, and returns the computed route coordinates along with distance and duration metrics as a RouteResult object. The Route Generation module handles the actual pathfinding using OSRM and applies any necessary optimizations or fallback strategies on the server side.

User Service: This service acts as the client-side adapter for the backend User Service package. It handles user authentication operations, profile management, and travel persona configuration. UserService accepts credentials or profile updates from authentication and settings pages, formats them into the request structure expected by the User Service REST API, sends requests over HTTP, and parses the server's JSON responses back into User objects and AuthResponse tokens. The service also manages session token storage in localStorage and includes the bearer token in all authenticated requests.

POI Service: This service acts as the client-side adapter for the backend Places Service package. It provides read-only access to the centralized POI database maintained by the Places Service, supporting queries by ID, type/category, rating score, and price level. PlacesService

accepts filter criteria from ExplorePanel and DestinationDetailPanel, constructs the appropriate REST API endpoint URL with query parameters, sends GET requests to the Places Service, and parses the JSON array responses into MapDestination objects with coordinates.

3. Class Interfaces

3.1. Server

3.1.1. User

class User
Attributes
private String Id
private String email
private String name
private String avatar
private List<TravelPersona> travelPersona
private List<TravelPlan> travelPlans
private Map<String, Chat> chats
Methods
public Chat giveFeedback(String feedback)
public TravelPlan createTravelPlan(Place[] places)
Getter and setter methods

class TravelPlan
Attributes
private String id
private List<String> selectedPlacesIds
Methods
Getter and setter methods

class TravelPersona
Attributes
private String id
private String[] travelStyles
private String[] interests
private String[] travelFrequency
private String[] preferredPace
Methods
Getter and setter methods

class UserController
Attributes
private User[] users
Methods
public User[] getAllUsers()
public User getUserById(int ID)
public User getUserByName(String name)
public void setUserName(String name)
public void setUserEmail(String email)
public void addTravelPersona(String userId, TravelPersona persona)
public void removeTravelPersona(String userId, String travelPersonaId)
public void setPassword(String password)
public void addSelectedPlace(String userId, Place place)

class Chat
Attributes
private String id
private String title
private String duration
private List<Message> messages
private long createdAt
private long updatedAt
private User user
Methods

Getter and setter methods

class Message
Attributes
private String id
private String role
private String content
private long timestamp
Methods
Getter and setter methods

3.1.2. POI (Places)

class Place
Attributes
private double latitude
private double longtitude
private String id
private String name
private String formattedAddress
private String[] types
private int ratingCount
private double ratingScore
private String priceLevel

private String businessStatus
Methods
Getter and setter methods

class POIController
Attributes
private List<Place> places
Methods
public Place[] getAllPlaces()
public Place[] getPlacesByType(String type)
public Place[] getPlacesByRatingScore(double minScore)
public Place[] getPlaceByName(String name)
Public Place[] getPlacesByPriceLevel(String priceLevel)
Place getPlaceById(long id)
Place[] getPlacesByIds(long[] ids)

3.1.3. Route Generation

class TripRequest
Attributes
private String requestId
private int maxBudgetMin
private int maxStops
private Set<String> mandatoryTypes

private Map<String, Float> overrideWeights
Methods
public List<Route> generateRoutes(Map<String, String> userVector, int k)
public void parseUserVector(Map<String, String> userVector)
public boolean validate()
public Route rerollRoutePoint(Route route, int index, Map<String, String> indexParams)
private Set<String> lockUnchangedPOIs(Route route, int index)
Getter and setter methods

class Route
Attributes
private String routeId
private List<RoutePoint> points
private List<RouteSegment> segments
private int totalDurationSec
private double totalDistanceM
private boolean feasible
Methods
public void rerollPoint(int index, Map<String, String> indexParams, TripRequest req, Set<String> lockedPOIIds)
public Route insertManualPOI(Route route, int index, String poiId)
public Route removePoint(Route route, int index)
public Route reorderPOIs(Route route, int[] index)
private void recomputeLocalSegments(int index)
private void recomputeTotals()
Getter and setter methods

class RoutePoint
Attributes
private int index
private Place poi
private int plannedVisitMin
Methods

public void assignPOI(Place poi)
Getter and setter methods

class RouteSegment
Attributes
private int fromIndex
private int toIndex
private int durationSec
private double distanceM
Methods
public boolean hasGeometry()
Getter and setter methods

3.1.4. LLM Agent

class llm_server
Attributes
TOOL_REGISTRY
CURRENT_LLM_MODEL
TOOLS_DEFINITION
SYSTEM_PROMPT
Methods
handle_chat(): json
index() : rendered_template
invoke_action(str, str) : str
get_tools_list() : json

class chatbot
Attributes
SYSTEM_PROMPT
TOOLS_DEFINITON
Methods
load_model():
extract_json_from_text(): json
ask_question(): json
get_tools_list(): json

class BaseAgent
Attributes
tool_template
Methods
__call__(): str

class UserProfileAgent_SetInfo
Attributes
tool_template
Methods
__call__(user_id, user_data_update_info, travel_id): str

class UserFeedbackAgent
Attributes
tool_template
Methods
__call__(trip_id, user_feedback)): str

class XAIJustificationAgent
Attributes
tool_template

Methods
__call__(recommendation_id, user_data): str

class Route_search_agent
Attributes
tool_template
Methods
__call__(POI_data, user_data, travelPersona): str

class POI_suggest_agent
Attributes
tool_template
Methods
__call__(user_data, current_route_info): str

class ItineraryModificationAgent
Attributes
tool_template
Methods
__call__(trip_id, action_type, poi_object): str

class weatherAgent
Attributes
tool_template
Methods
__call__(Location: str, unit: str): str

3.2. Client

3.2.1. State Management

class AuthState
Attributes
private User | null user
private boolean isAuthenticated
private boolean isLoading
private String | null error
private boolean isNewSignup
Methods
public void setLoading(boolean loading)
public void setError(String | null error)
public void loginSuccess(User user)
public void logout()
public void signupSuccess(User user)
public void clearNewSignup()
public void updateUser(Partial<User> updates)
public void updateTravelPersona(TravelPersona persona)
public void deleteAccount()

public void loginUser(String email, String password)
public void signupUser(String name, String email, String password)
private AuthState loadAuthState()

type User
Attributes
private String id
private String email
private String name
private String | undefined avatar
private TravelPersona | undefined travelPersona
private boolean hasCompletedOnboarding

type TravelPersona
Attributes
Private String id
private List<String> travelStyles
private List<String> interests
private String travelFrequency
private String preferredPace

class ChatState
Attributes
private List<Chat> chats
private Chat | null activeChat

private boolean isLoading
private String | null error
private boolean sidebarOpen
private boolean mapFullscreen
private int chatPanelWidth
private boolean exploreMode
private boolean savedMode
private List<String> savedDestinations
Methods
public void createChat(String id, String title, String initialQuery)
public void setActiveChat(String chatId)
public void addMessage(String chatId, Message message)
public void updateChatTitle(String chatId, String title)
public void deleteChat(String chatId)
public void setLoading(boolean loading)
public void setError(String | null error)
public void toggleSidebar()
public void setSidebarOpen(boolean open)
public void toggleMapFullscreen()
public void setChatPanelWidth(int percentage)
public void toggleExploreMode()
public void setExploreMode(boolean active)
public void toggleSavedMode()
public void setSavedMode(boolean active)
public void toggleSavedDestination(String destinationId)

public void addSavedDestination(String destinationId)
private List<Chat> loadChats()
private void saveChats(List<Chat> chats)

type Chat
Attributes
String id
String title
String | undefined duration
List<Message> messages
long createdAt
long updatedAt
User user

type Message
Attributes
String id
‘user’ | ‘assistant’ role
String content
long timestamp
LocationCard | undefined locationCard

type LocationCard
Attributes
private String name

private String type
private float rating
private String priceLevel
private String | undefined image
private double lat
private double lng

class SavedState
Attributes
private List<MapDestination> destinations
Methods
public void saveDestination(MapDestination destination)
public void unsaveDestination(String destinationId)
public void toggleSaveDestination(MapDestination destination)
private List<MapDestination> loadSaved()
private void saveSaved(List<MapDestination> destinations)

class NavigationState
Attributes
private List<String | null> stops
private 'driving' | 'walking' mode
private RouteInfo | null routeInfo
private List<Coordinate> | null routeCoordinates
private boolean isLoadingRoute
Methods

public void setStops(List<String | null> stops)
public void addStop()
public void toggleSaveDestination(MapDestination destination)
public void removeStop(int index)
public void updateStop(int index, String | null value)
public void setMode(String mode)
public void setRouteInfo(RouteInfo | null info)
public void setRouteCoordinates(List<Coordinate> | null coords)
public void setLoadingRoute(boolean loading)
public void clearRoute()

3.2.2. UI Component

class ChatPanel
Props
private String userName
private Function | undefined onMenuClick
private boolean showMenuButton
private boolean isNewChatMode
private Function onCreateChat
State
private String input
private List<String> suggestions
Methods
private void scrollToBottom()

private void handleSend()
private void handleKeyPress(KeyboardEvent e)

class ChatMessage
Props
private Message message
Methods
LocationCardComponent(LocationCard location) : JSX (React.FC)
MarkdownContent(String content) : JSX (React)

class ChatSidebar
Props
private boolean mobile
private Function | undefined onClose
State
private String editingChatId
private String editTitle
private boolean showDeleteConfirm
private String deleteTargetId
Methods
private void handleNewChat()
private void handleChatClick(String chatId)
private void handleNavClick(String path)
private void handleLogoClick()
private void handleRenameChat(String chatId, String currentTitle)

private void handleConfirmRename()
private void handleDeleteChat(String chatId)
private void handleLogout()
private void handleViewMapToggle()
private String getInitials(String name)

class MapPanel
Props
private List<MapDestination> destinations
private MapDestination | null highlightedDestination
private Function onDestinationSelect (optional)
private List<Coordinate> | null route
private List<MapDestination> orderedDestinations
State
private MapDestination | null hoveredDestination
private boolean isPopupHovered
private Coordinate | null popupPosition
Methods
private boolean isDestinationSaved(String destinationId)
private void handleMarkerHover(MapDestination destination, LeafletMouseEvent event)
private void handleMarkerLeave()
private void handlePopupMouseEnter()
private void handlePopupMouseLeave()
private void handleMapClick()
private void handleSaveClick(MouseEvent e)

private void handleDestinationClick()
createCustomIcon(String color, int number) : LeafletIcon
MapResizeHandler(boolean fullscreen) : JSX
MapClickHandler(Function onMapClick) : JSX

class ExplorePanel
Props
private Function | undefined onDestinationSelect
private Function | undefined onDestinationHover
private Function | undefined onMenuClick
private boolean showMenuButton
State
private String searchQuery
private String selectedCategory
Methods
private boolean isDestinationSaved(String destinationId)
private void handleSaveClick(MapDestination destination, MouseEvent e)
private void handleMenuClick()

class SavedPanel
Props
private Function | undefined onDestinationSelect
private Function | undefined onDestinationHover
private Function | undefined onMenuClick
private boolean isNewChatMode

private Function onCreateChat
State
private String searchQuery
private String selectedCategory
Methods
private boolean isDestinationSaved(String destinationId)
private void handleSaveClick(MapDestination destination, MouseEvent e)
private void handleMenuClick()

class NavigationPanel
Props
private MapDestination destination
private Function onClose
State
private int panelWidth
Methods
private void handleToggleSave()
private void handleMouseDown(MouseEvent e)

class ResizableDivider
Props
private ‘vertical’ | ‘horizontal’ orientation
State
private boolean isDragging
Methods

private void handleMouseDown(MouseEvent e)
private void handleMouseMove(MouseEvent e)
private void handleMouseUp()
private void handleTouchStart(TouchEvent e)
private void handleTouchMove(TouchEvent e)
private void handleTouchEnd()

class Header
Props
private boolean mobileMenuOpen
private String mode // current color scheme
Methods
private void toggleTheme()
private void handleLogout()
private String getInitials(String name)

class Footer

class AvatarMenu
State
private boolean open
Methods
private void handleLogout()
private void handleProfileClick()
private void handleSettingsClick()

private String getInitials(String name)

type DestinationCard
State
private Destination destination
private Function | undefined onClick

class ThemeToggle
Props
private ‘sm’ | ‘md’ | ‘lg’ size
private ‘plain’ | ‘outlined’ | ‘soft’ | ‘solid’ variant
Methods
private void toggleTheme()

3.2.3. Data & Models

class Destination
Attributes
private String id
private String name
private String location
private String image
private float rating
private int priceLevel
private String category

private String | undefined description
private String | undefined openingHours
private String | undefined phone
private String | undefined website
private int | undefined reviewCount

class MapDestination extends Destination
Attributes
private Coordinate coordinates // [lat, lng]

class Destination
Attributes
public static List<MapDestination> ankaraDestinations
public static List<MapDestination> cafeDestinations
public static List<MapDestination> allDestinations
public static List<String> categories
Methods
public static List<MapDestination> searchDestinations(String query, String category)
public static MapDestination getDestinationById(String id)

class Destination
Attributes
public static List<MapDestination> ankaraDestinations
public static List<MapDestination> cafeDestinations
public static List<MapDestination> allDestinations

public static List<String> categories
Methods
public static List<MapDestination> searchDestinations(String query, String category)
public static MapDestination getDestinationById(String id)

3.2.4. Services

class LLMService
Attributes
private static String LLM_SERVER_BASE_URL
Types
ChatMessage { String role, String content }
ToolCallResult { String type, List<MapDestination> destinations, MapDestination savedDestination, String message }
Methods
public static ToolCallResult sendMessage(String message, List<ChatMessage> history)
public static String sendMessageSimple(String message, List<ChatMessage> history)
public static String generateTripTitle(String query)
private static ChatMessage formatRequest(String message, List<ChatMessage> history)
private static ToolCallResult parseResponse(LLMServerResponse response)

class RoutingService
Attributes
private static String ROUTE_API_BASE_URL
Types

RouteResult { List<Coordinate> coordinates, double distance, double duration }
Methods
public static RouteResult getRoute(List<Coordinate> waypoints, String mode)
public static String sendMessageSimple(String message, List<ChatMessage> history)
private static RouteRequest formatRequest(List<Coordinate> waypoints, String mode)
private static RouteResult parseResponse(RouteGenerationResponse response)

class UserService
Attributes
private static String USER_SERVICE_BASE_URL
private static String authToken
Types
AuthResponse { User user, String token, long expiresAt }
Methods
public static User getCurrentUser()
public static User updateUserProfile(String name, String email, String avatar)
public static User updateTravelPersona(TravelPersona persona)
public static void removeTravelPersona()
public static void changePassword(String oldPassword, String newPassword)
public static AuthResponse loginUser(String email, String password)
public static AuthResponse registerUser(String name, String email, String password)
private static HttpRequest formatRequest(Object data)
private static User parseResponse(UserServiceResponse response)

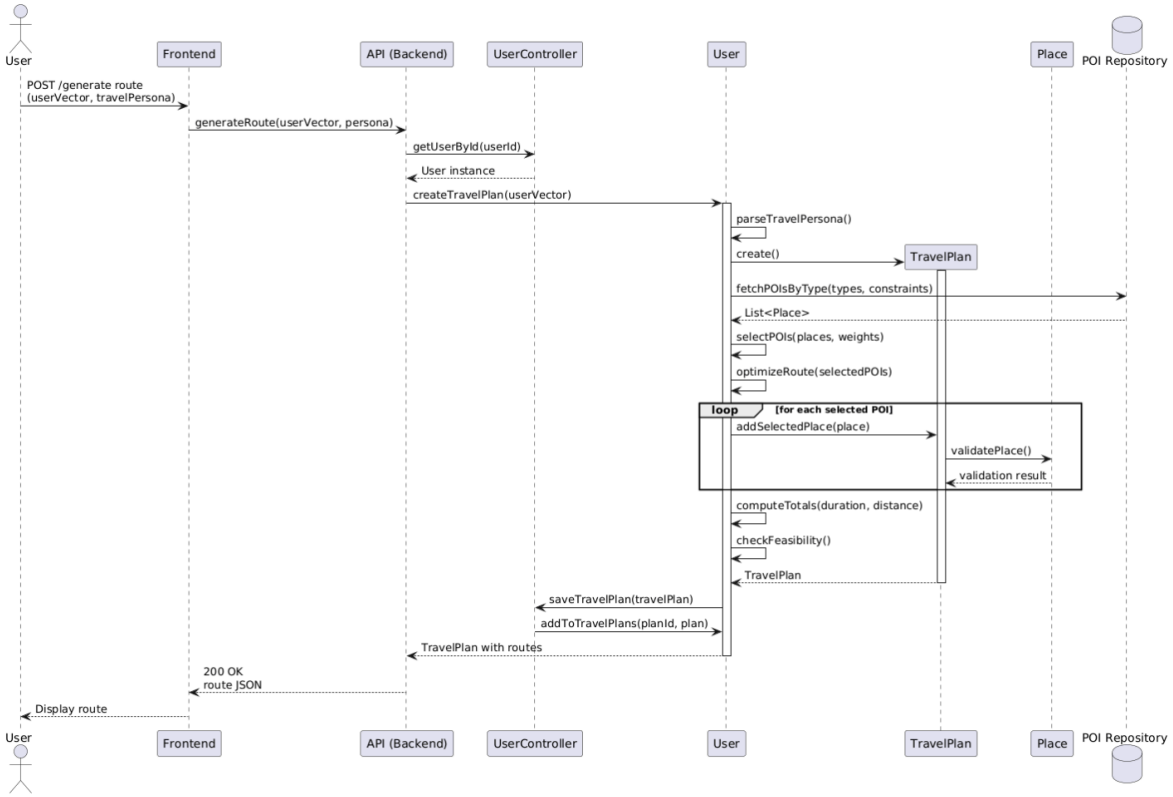
private static void storeAuthToken(String token)
private static String getStoredAuthToken()

class PlacesService
Attributes
private static String PLACES_SERVICE_BASE_URL
private static String authToken
Methods
public static List<Place> getAllPlaces()
public static Place getPlaceById(String id)
public static List<Place> getPlacesByType(String type)
public static List<Place> getPlacesByRating(double minScore)
public static List<Place> getPlacesByPriceLevel(String priceLevel)
private static HttpRequest formatRequest(Map<String, Object> params)
private static List<Place> parseResponse(PlacesServiceResponse response)

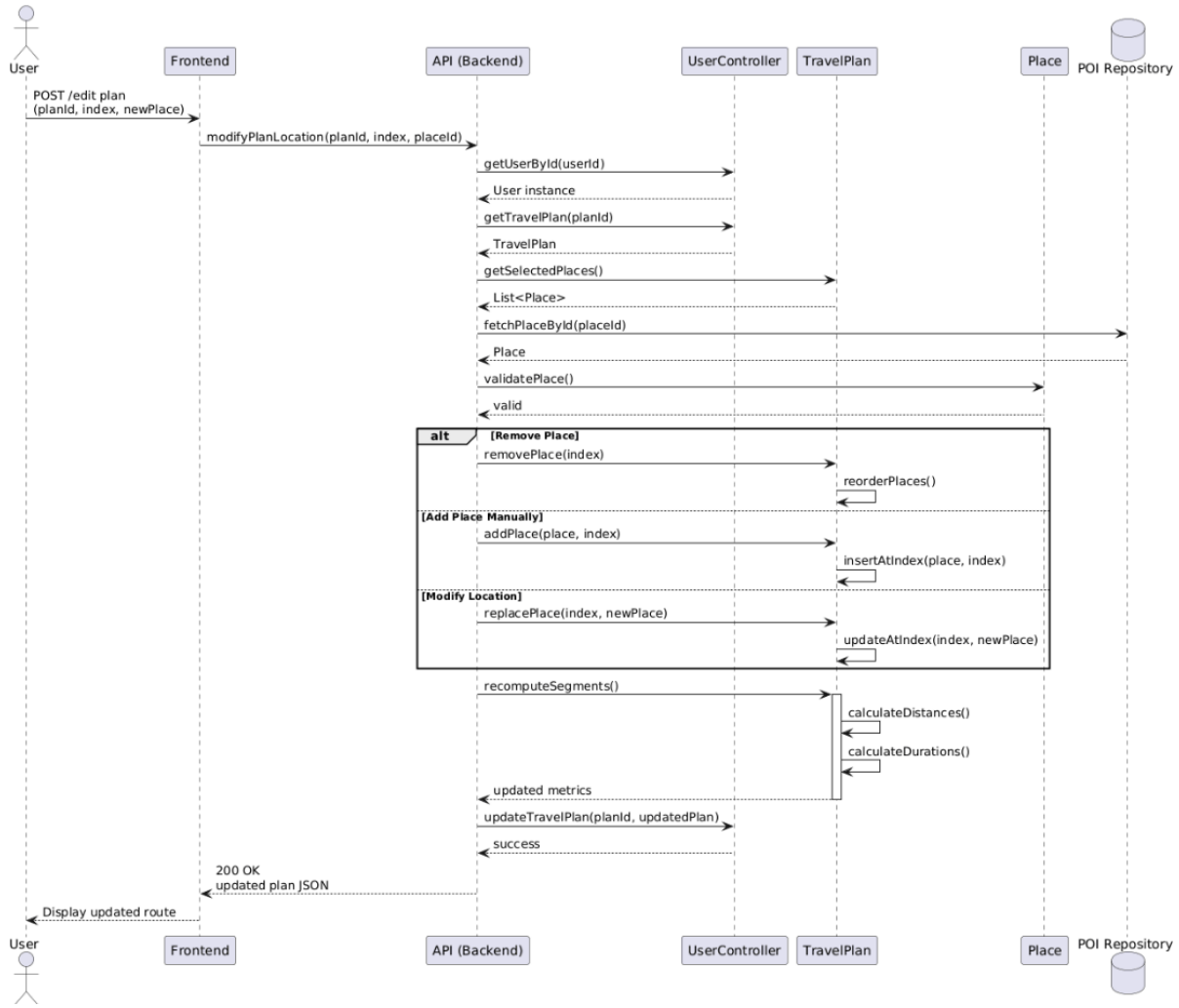
4. Sequence & Use-Case Diagrams

4.1. User

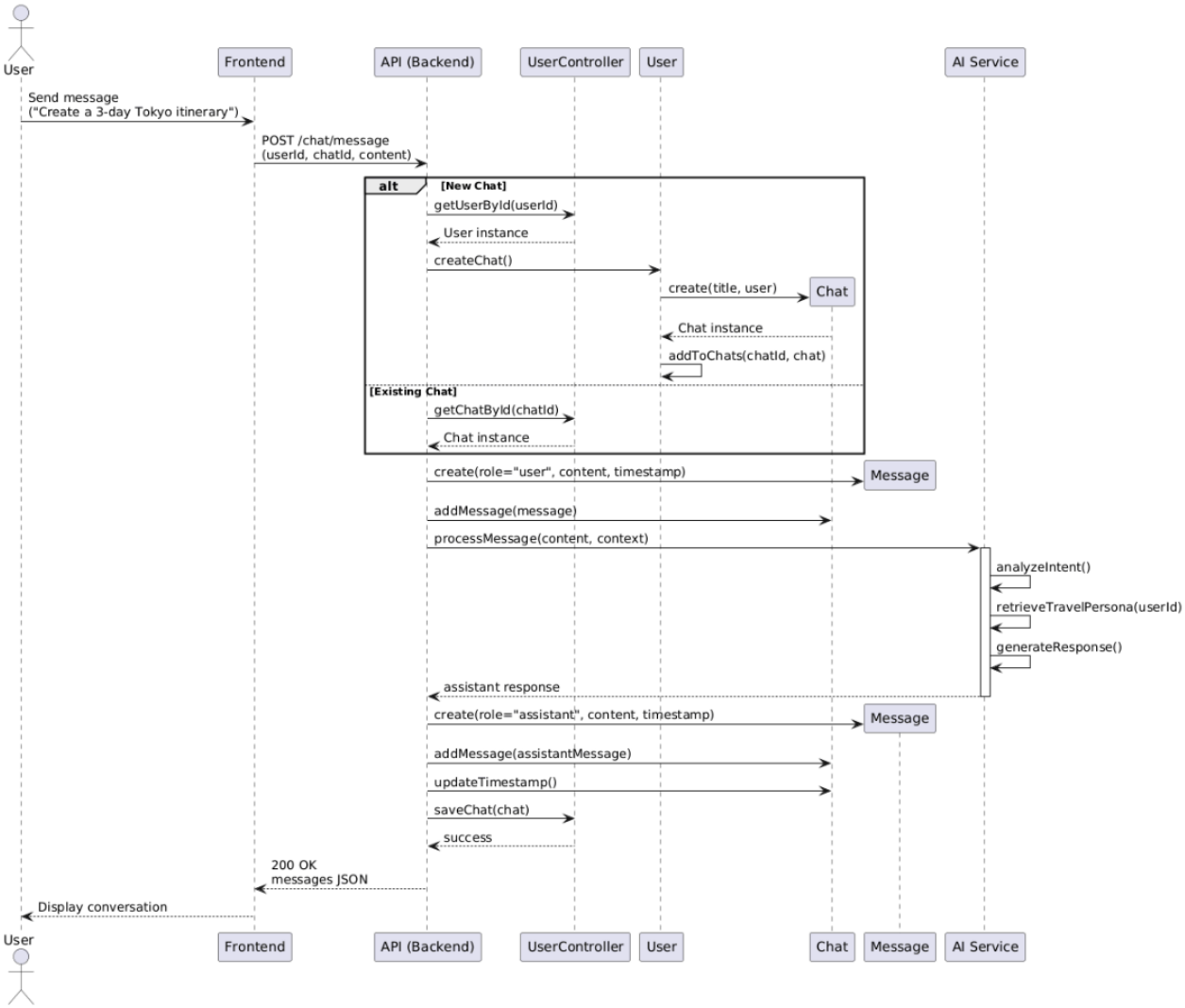
4.1.1. Sequence Diagrams



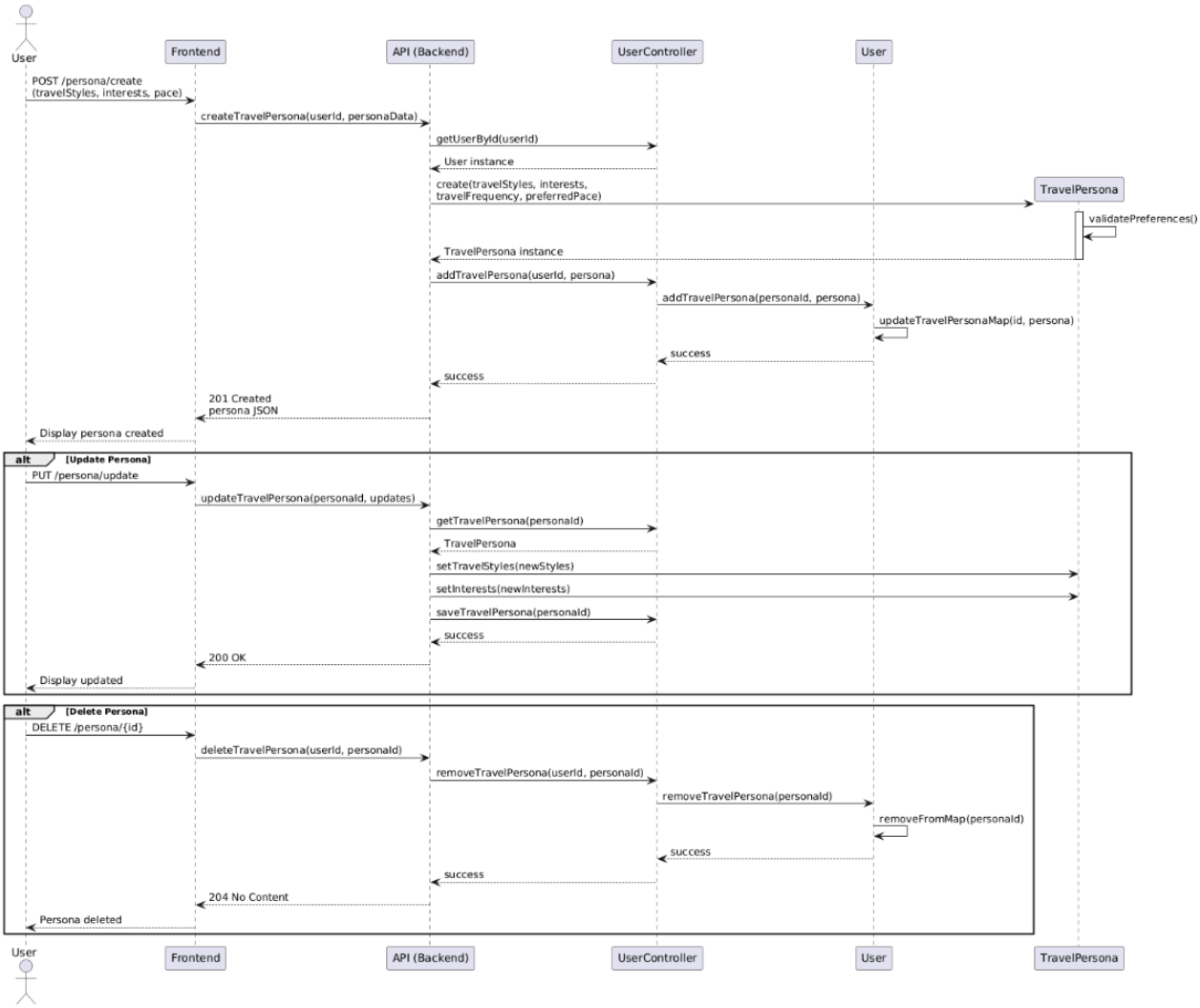
This diagram shows the create travel plan workflow where a user submits route generation parameters through the frontend. The API Backend's UserController retrieves the user instance, which then initiates travel plan creation by parsing the travel persona, instantiating a TravelPlan, and fetching relevant POIs from the repository based on type and constraints. The system selects and optimizes POIs according to preference weights, performs a feasibility check, and validates each place before adding it to the plan. Finally, the completed TravelPlan (containing only place IDs) is saved to the user's collection and returned with route details to display in the frontend.



This diagram shows how a user modifies an existing travel plan by adding, removing, or replacing places. The frontend sends the modification request to UserController, which retrieves the TravelPlan and validates the target place through the POI Repository. Depending on the operation (remove, add manually, or modify location), the system updates the place list at the specified index, recomputes affected route segments (calculating distances and durations), and saves the updated plan before returning the modified route to the frontend.

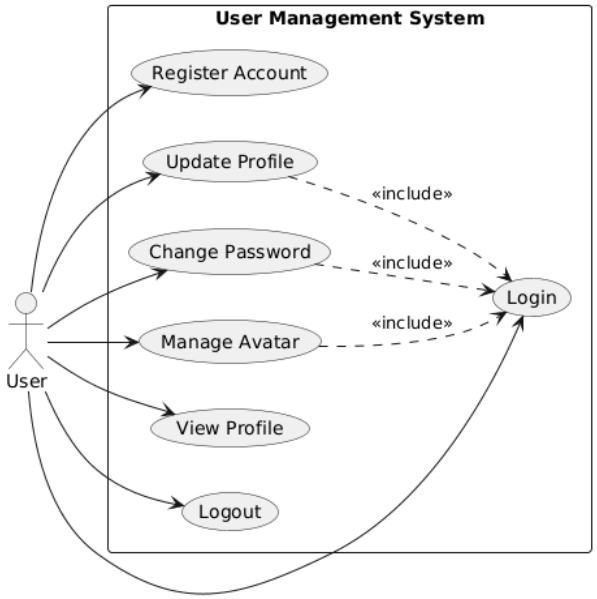


This diagram shows a user initiating a chat conversation to create a travel itinerary. The frontend posts the message to the API Backend, which either creates a new Chat or retrieves an existing one through UserController. The user's message is added to the Chat, then forwarded to the AI Service for processing. The AI Service analyzes the intent, retrieves the user's TravelPersona, and generates a response. The assistant's message is added to the Chat history, timestamps are updated, and the complete conversation is returned to display in the frontend.

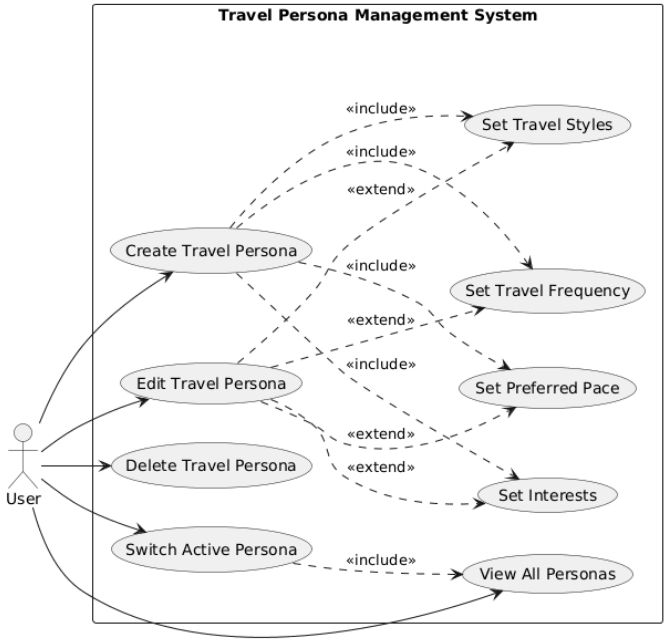


This diagram shows the creation, update, and deletion of travel personas. When creating a persona, the frontend sends travel style and preference data to UserController, which instantiates a new TravelPersona, validates preferences, updates the User's persona map, and returns success. For updates, the system retrieves the existing persona, modifies specified attributes (travel styles, interests, frequency, pace), and saves changes. For deletion, UserController removes the persona from the user's map and confirms removal, enabling users to manage multiple distinct travel preference profiles.

4.1.2. Use Case Diagrams

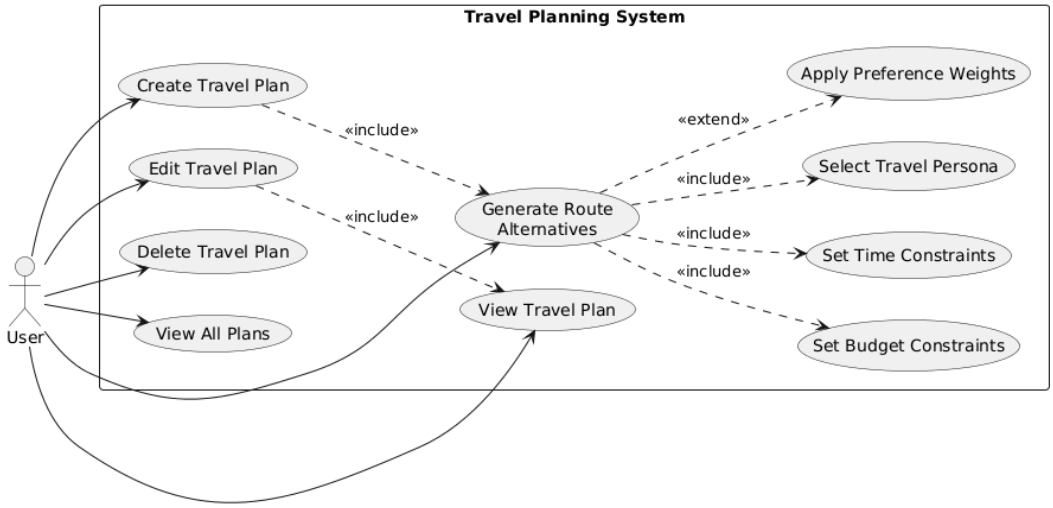


This diagram shows core user account management functionality. Users can register new accounts, update their profile information, change passwords, and manage avatars. The Update Profile, Change Password, and Manage Avatar use cases all include a Login prerequisite, ensuring authenticated access. Users can also view their profile and logout from the system.

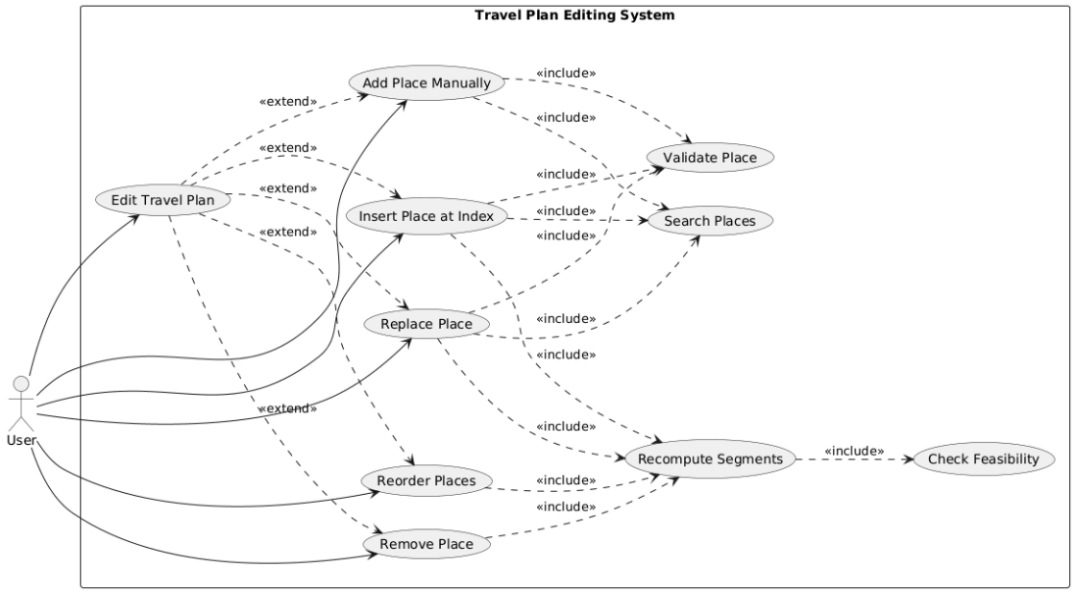


This diagram illustrates how users manage multiple travel preference profiles. Users can create new personas, edit existing ones, delete personas, and switch between active personas. The Create Travel Persona use case extends to specialized configuration actions: Set Travel Styles, Set Travel Frequency, Set Preferred Pace, and Set Interests. Users can also view all their saved

personas. Edit Travel Persona similarly extends to the same configuration actions for modifying existing preferences.



This diagram depicts the itinerary planning workflow. Users can create, edit, delete, and view travel plans. The central Generate Route Alternatives use case includes selecting a travel persona and extends to Apply Preference Weights, Set Time Constraints, and Set Budget Constraints. These planning parameters inform the route generation algorithm. The View Travel Plan use case provides access to saved itineraries.

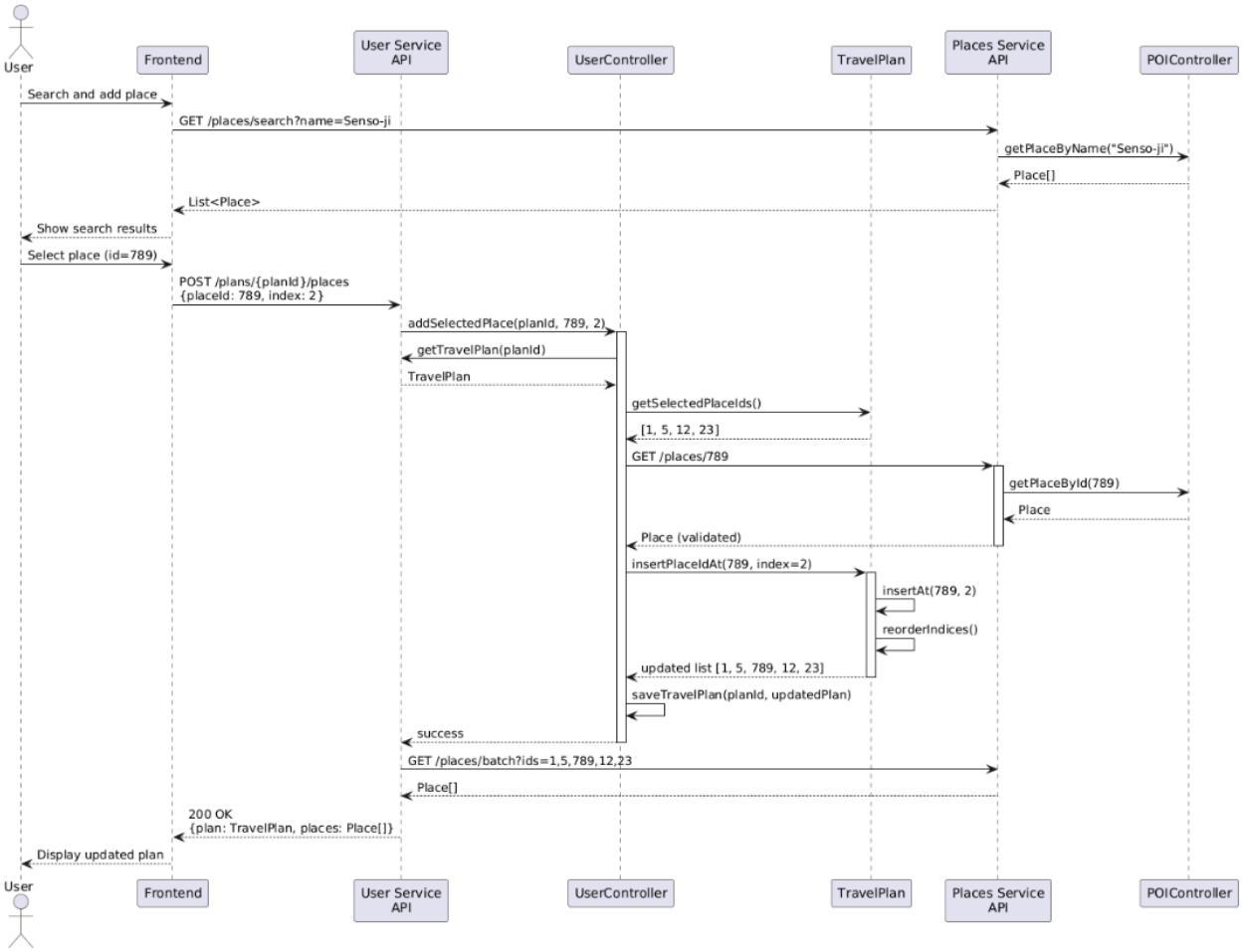


This diagram shows itinerary modification capabilities. The Edit Travel Plan use case extends to multiple operations: Add Place Manually (with Validate Place and Search Places includes), Insert Place at Index (includes Search Places), Replace Place (includes Search Places), Reorder Places, and Remove Place. All modification operations that alter the route structure include

Recompute Segments, which in turn includes Check Feasibility to ensure the updated itinerary remains valid against time and budget constraints.

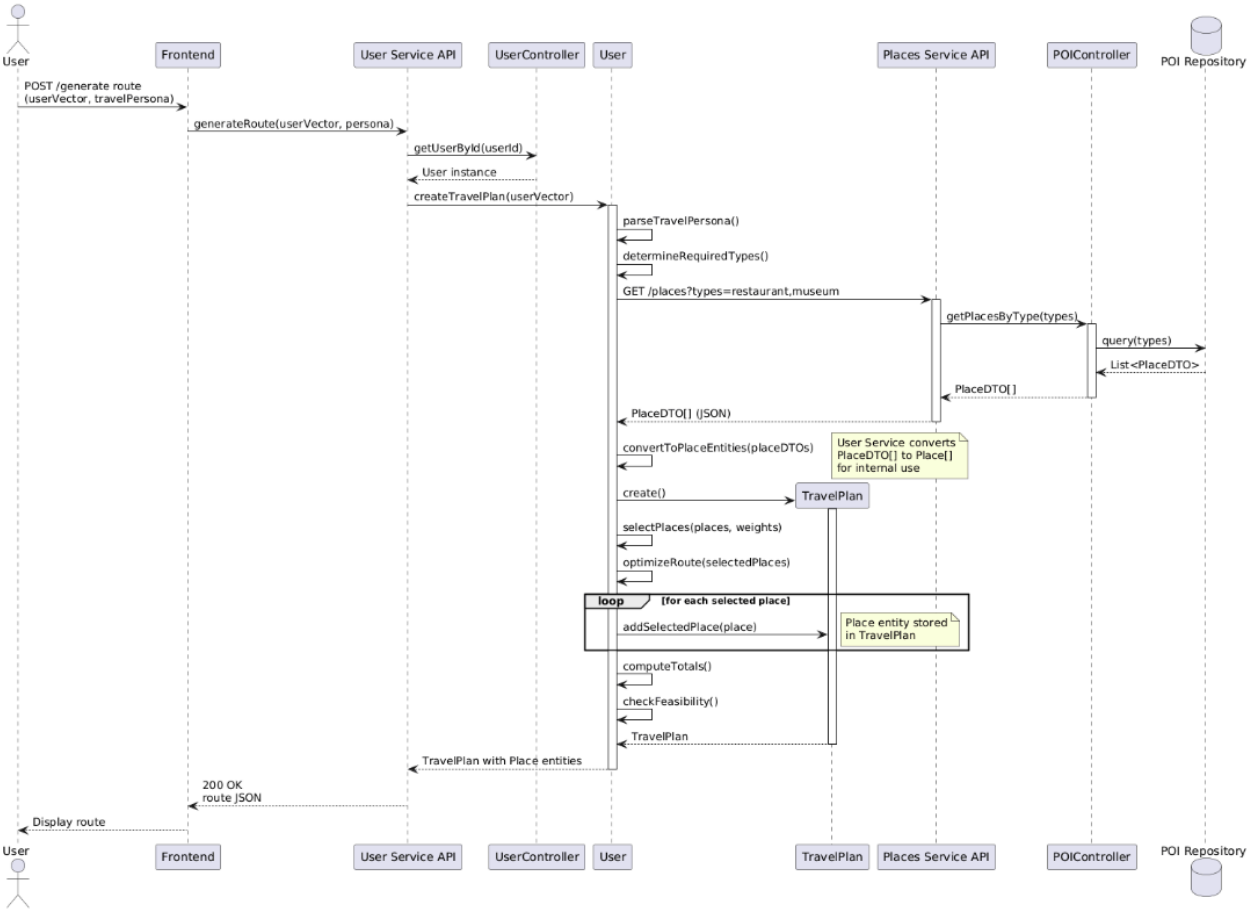
4.2. POI (Places)

4.2.1. Sequence Diagrams



This diagram shows how users search for and add places to their travel plan. The user initiates a search through the frontend, which queries the Places Service API by name (e.g., "Senso-ji"). The POIController returns matching places, which are displayed to the user. Upon selecting a place, the frontend sends an add request to UserController with the place ID and index. UserController retrieves the TravelPlan, fetches the selected place IDs, validates the new place through the Places Service, and inserts it at the specified index. The system then recomputes

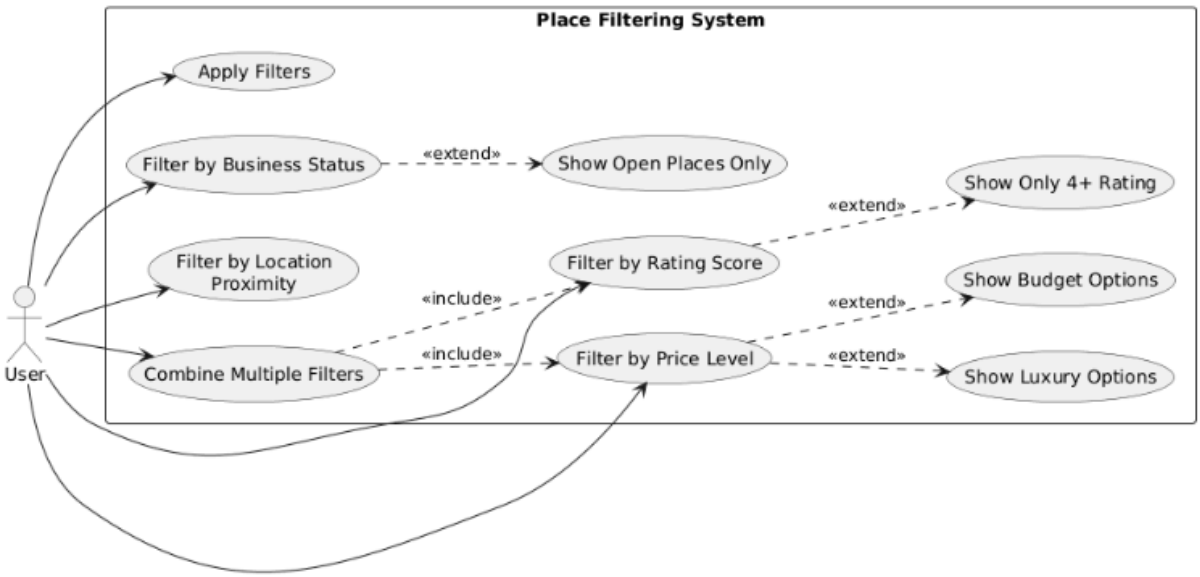
route segments and feasibility before returning the updated plan with full place details to the frontend.



This diagram shows automated route generation based on user preferences. The user submits route parameters and travel persona through the frontend to UserController, which retrieves the User instance and initiates travel plan creation. The system parses the TravelPersona, queries the Places Service for POIs matching the specified types (e.g., restaurant, museum), and receives a filtered list. The route generation algorithm selects POIs based on preference weights, optimizes the route order, and stores selected place entities in TravelPlan. After computing totals and checking feasibility, the completed plan is saved and returned with route details to display in the frontend.

4.2.2. Use Case Diagrams

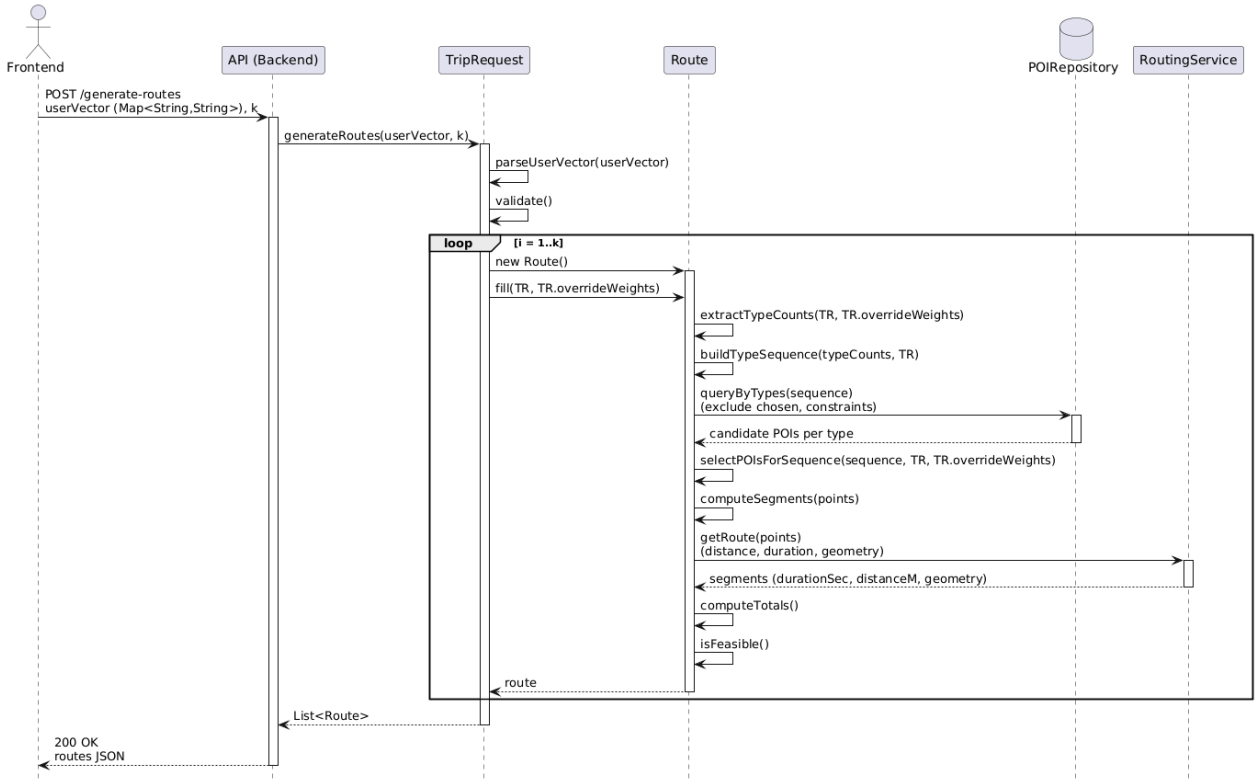
This diagram shows place discovery and filtering capabilities available to users and external systems via API. Users can search places by type (extending to specific categories like Search Restaurants, Search Attractions, Search Hotels, Search Museums), search by name, filter by rating score, and filter by price level. The View Place Details use case includes Search All Places, providing access to comprehensive place information. External systems can also access these search capabilities through API endpoints for integration purposes.



This diagram shows advanced filtering options for narrowing place search results. Users can apply single filters or combine multiple filters simultaneously. Filter by Business Status extends to Show Open Places Only, while Filter by Rating Score extends to Show Only 4+ Rating. Filter by Price Level extends to both Show Budget Options and Show Luxury Options. The Combine Multiple Filters use case includes both Filter by Rating Score and Filter by Price Level, enabling users to create complex queries that match specific criteria (e.g., high-rated budget restaurants).

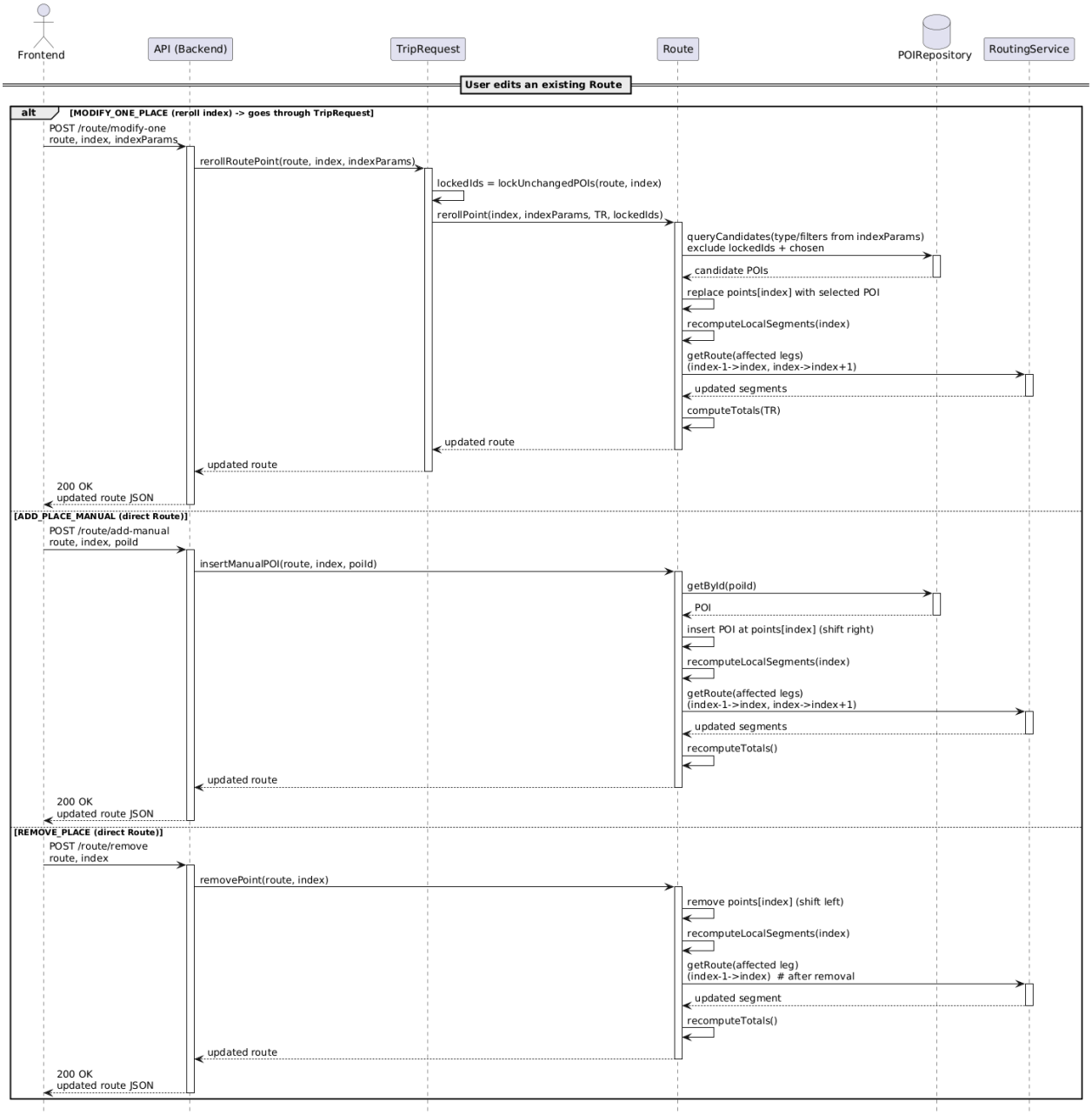
4.3. Route Generation

4.3.1. Sequence Diagrams



The sequence diagram presents the overall workflow of the route generation process initiated by a client request. The process begins when the client submits a planning request that includes preference parameters and the desired number of alternative itineraries. The backend component responsible for handling this request first interprets the incoming preference data and extracts the relevant constraints and weight distributions. A validation step ensures that the provided inputs are consistent and suitable for route construction.

After validation, the system generates multiple route alternatives in an iterative manner. For each alternative, a new itinerary object is created and constructed through a series of internal steps. First, the system determines how many locations of each category should appear based on user-defined priorities and constraints. It then establishes an ordered structure for these categories. Suitable location candidates are retrieved from the data repository while respecting exclusion rules and hard constraints. Once concrete locations are selected and assigned in sequence, travel distances and durations between consecutive visits are computed through a routing component. Finally, aggregate metrics such as total duration and total distance are calculated, and overall feasibility is evaluated. When all alternatives are completed, the backend returns the resulting list of routes to the client.



This sequence diagram illustrates the interaction flow when a user edits an already generated route. The diagram covers three distinct scenarios: modifying a single location, manually adding a new location, and removing an existing location.

In the first scenario, where a single location is modified, the request is routed through the trip-level coordination component to ensure consistency with the original constraints. The system locks all unchanged locations to prevent unintended modifications and rerolls only the selected index. A new candidate location is retrieved from the data repository while excluding locked and

already chosen entries. After replacing the targeted visit, only the affected travel segments are recomputed using the routing service. The total duration, distance, and feasibility values are then updated before returning the revised route to the client.

In the second scenario, where a user manually adds a location, the request is handled directly at the route level. The specified location is retrieved from the repository and inserted at the given index, shifting subsequent visits accordingly. The system recalculates only the locally affected segments and updates aggregate metrics to reflect the structural change. The updated route is then returned.

In the third scenario, where a location is removed, the route removes the visit at the specified index and shifts remaining visits to maintain ordering. As in the other scenarios, only the adjacent travel segments are recomputed, followed by recalculation of total metrics. The updated route is then sent back to the client.

4.3.2. Use Case Diagrams

This use case diagram presents the functional scope of the Travel Planner Backend from the perspective of the user. The primary actor is the User, who interacts with the system to generate route alternatives and to modify an existing route through three main operations: modifying a single location, manually adding a place, and removing a place.

The “Generate Route Alternatives” use case represents the core functionality of the system. It includes request validation, parsing of the user preference vector, selection of POIs according to constraints and weights, computation of route segments, and calculation of total metrics and feasibility. These included use cases represent internal responsibilities required to construct a complete and valid itinerary.

The “Modify One Location” use case allows the user to reroll a specific indexed POI in an existing route. This process includes locking unchanged POIs, selecting a replacement location, recomputing only the affected travel segments, and updating totals and feasibility. The inclusion relationships indicate that modification reuses core route-construction behaviors while limiting the scope of recalculation.

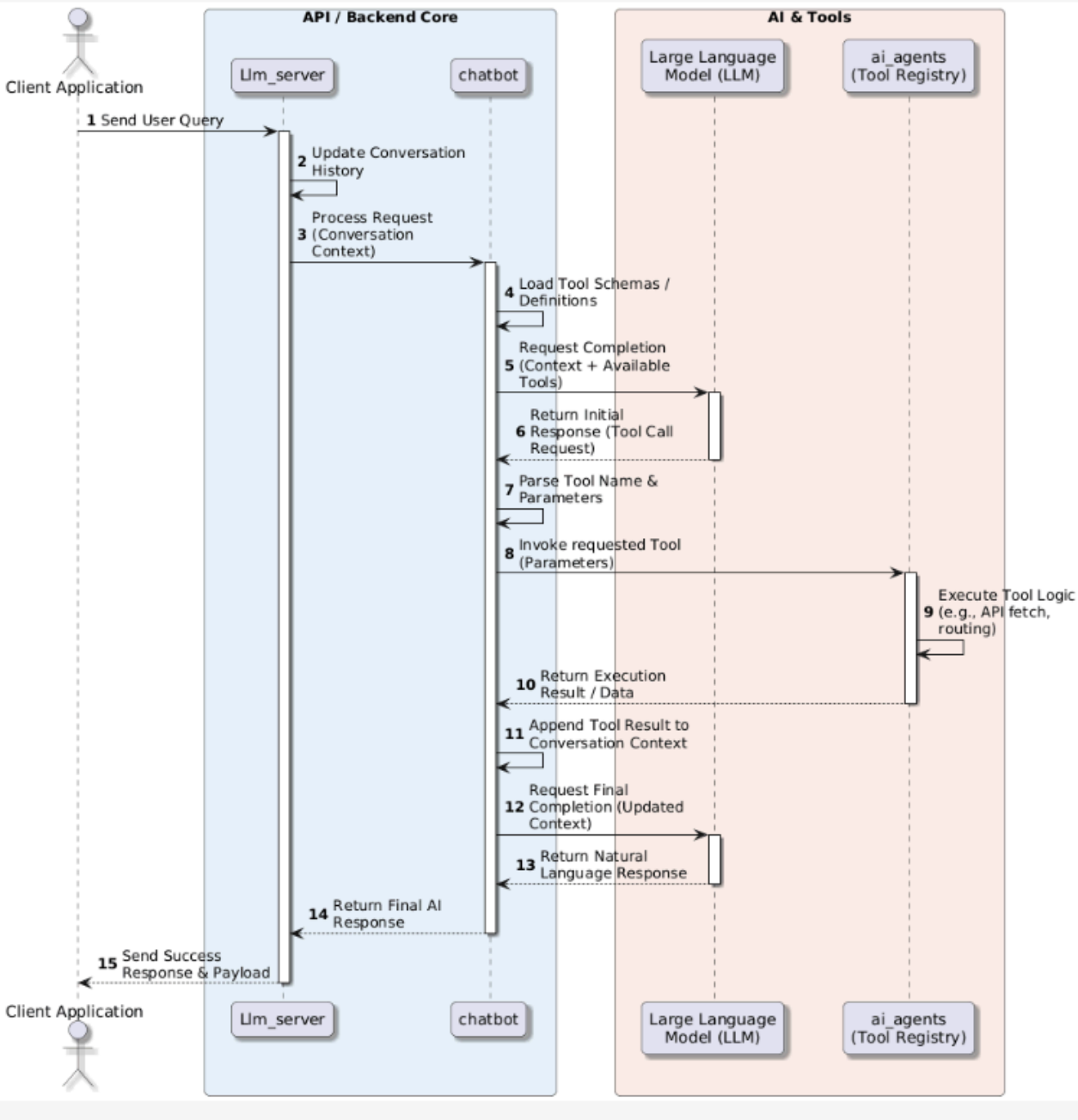
The “Add Place Manually” use case enables insertion of a new POI at a specified position within the route. It includes selecting or retrieving the specified POI, inserting it at the given index, recomputing adjacent segments, and updating aggregated metrics.

The “Remove Place” use case allows deletion of a visit at a specific index. It includes removing the POI, recomputing affected segments, and recalculating totals to maintain route consistency.

The diagram also shows interactions with two external system components: the POI Repository, which provides candidate or specific POI data, and the Routing Service, which computes travel distance and duration between consecutive visits. These external actors support selection and routing computations but do not initiate use cases themselves.

4.4. LLM Agent

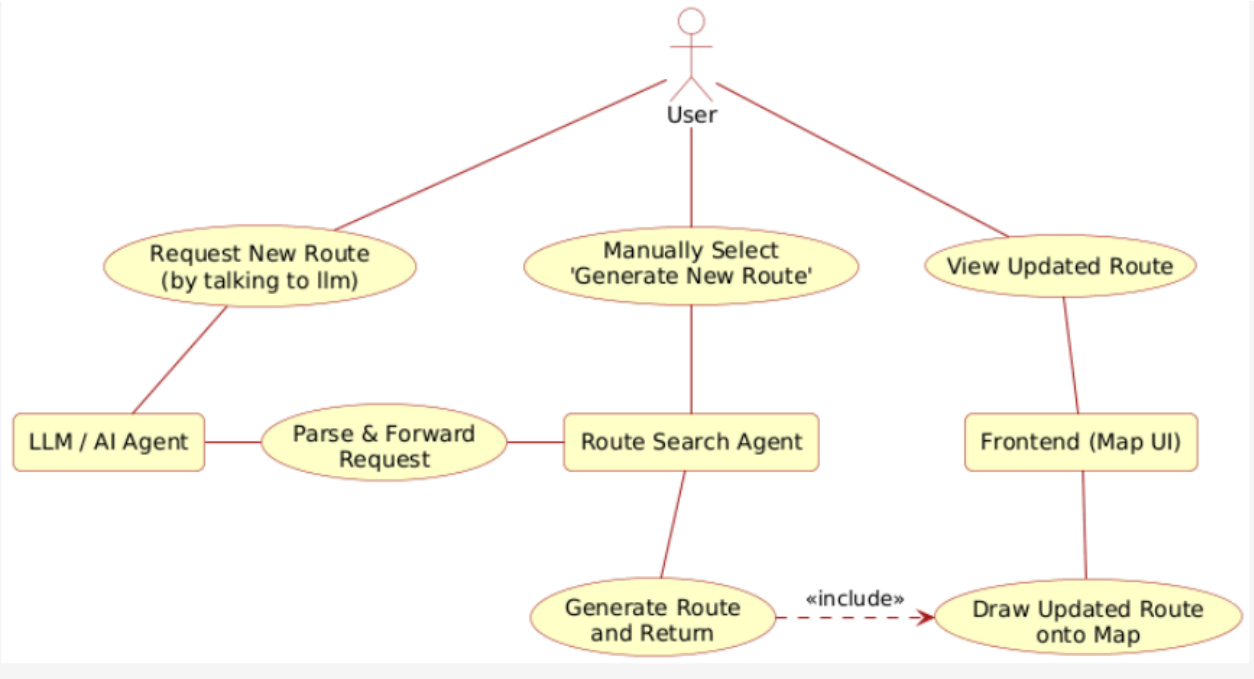
4.4.1. Sequence Diagrams



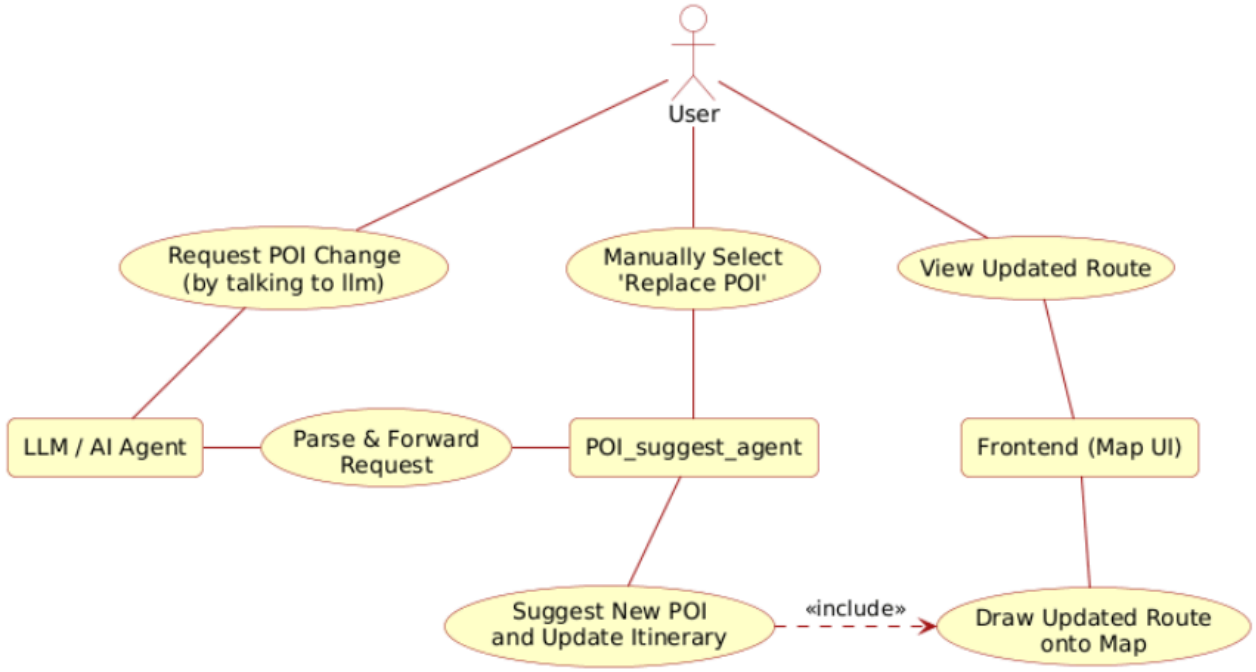
The client application submits a natural language query to the backend server, which updates the conversation history and forwards it to the chatbot orchestrator. The orchestrator consults the Large Language Model (LLM), which evaluates the context and either provides a direct text response or dynamically requests specific data from a registered tool. If a tool is invoked, its execution results are appended to the conversation and sent back to the LLM for a final processing pass, ensuring the ultimate response returned to the user is accurate and

comprehensive. Based on the tool calls, an action is either taken, or is asked to be taken by the client-side. This allows the chatbot - user integration to be seamless and very powerful.

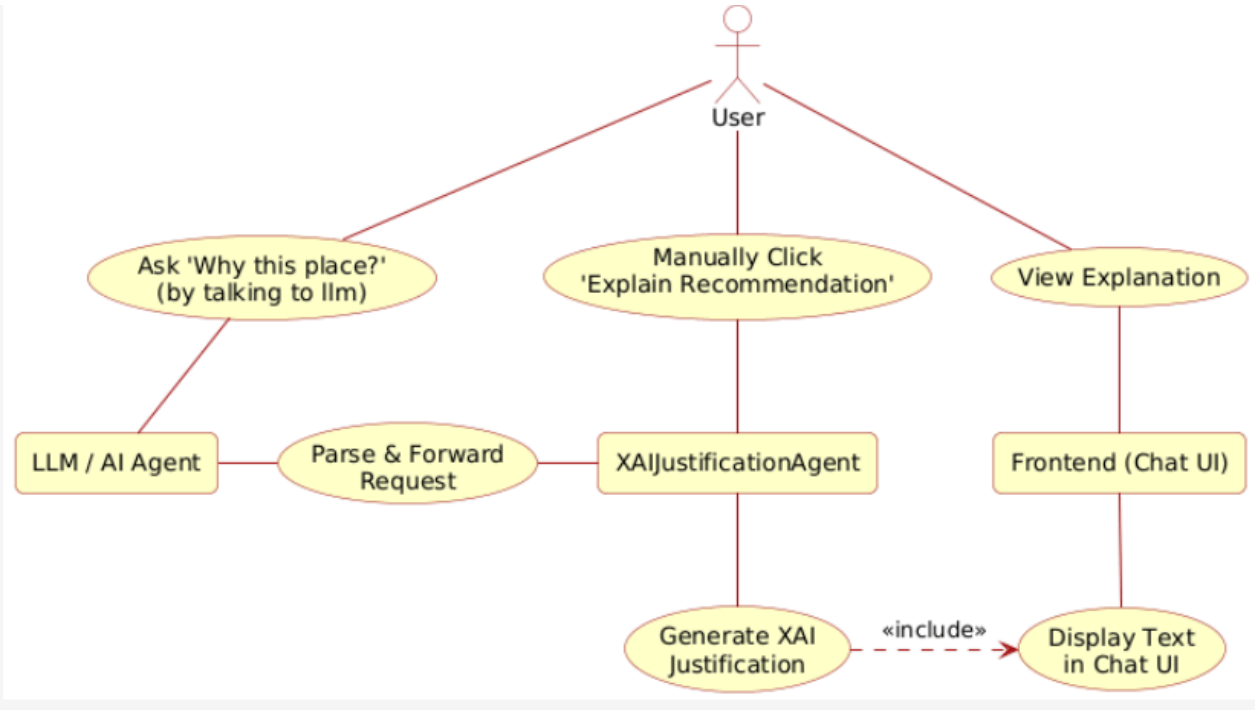
4.4.2. Use Case Diagrams



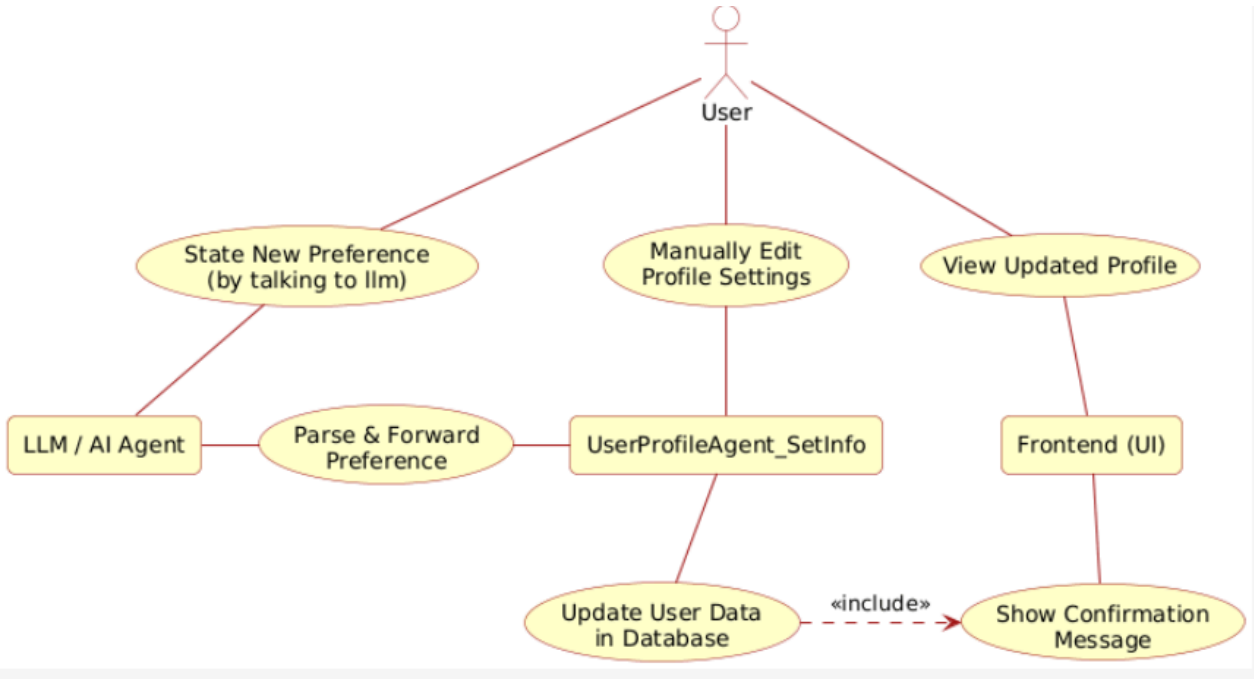
The user requests the creation of a brand-new travel itinerary from scratch, utilizing either natural language prompts sent to the AI chatbot or manual interface selections. These inputs are processed by the central Route Search Agent, which calculates an optimized sequence of destinations and commands the frontend to render the complete journey on the interactive map.



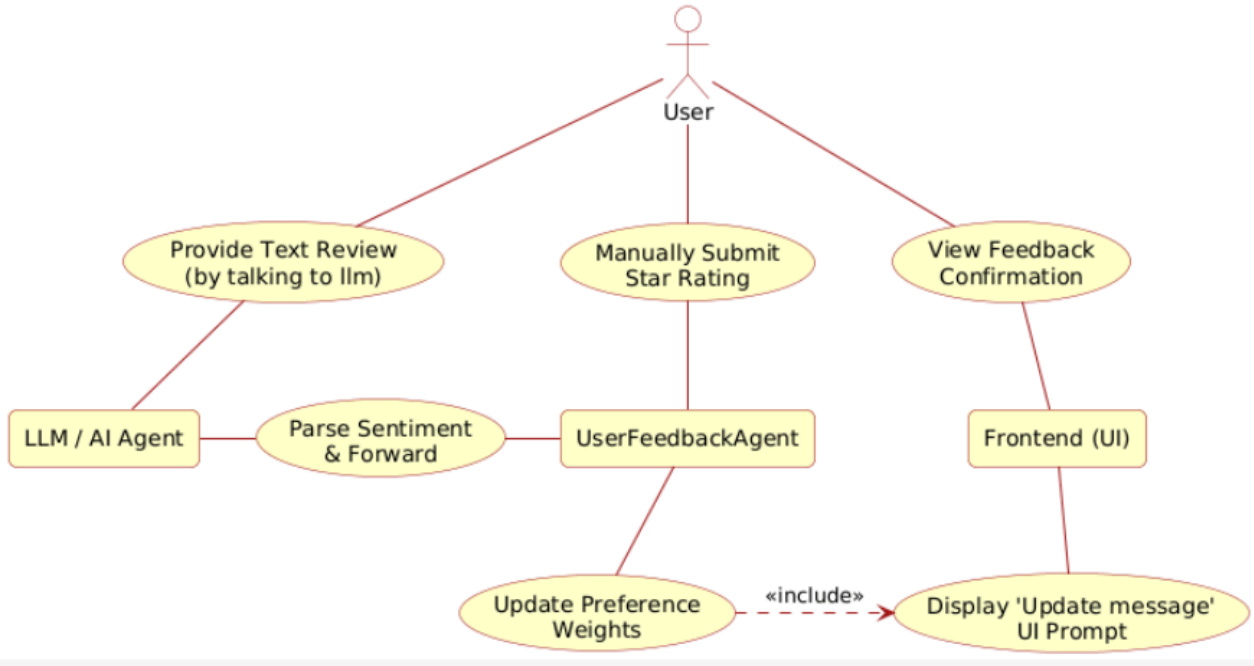
The traveler initiates a request to an update of a specific location in their existing itinerary, either by conversing with the AI chatbot or by clicking a manual replacement button on the interface. The system routes this request to the POI_suggest_agent, which intelligently selects a new waypoint and prompts the map UI to immediately display the updated route.



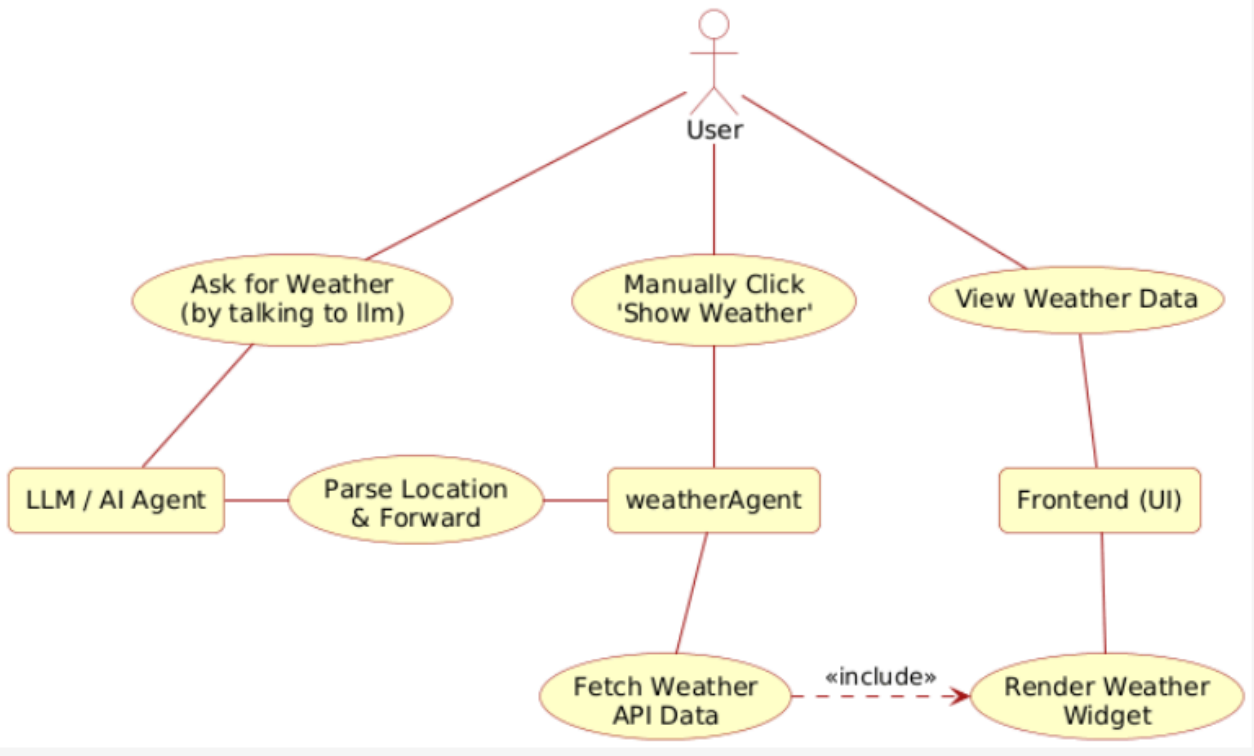
The traveler asks the chatbot why a specific Point of Interest was included in their itinerary, or clicks an "info" button next to a recommendation. The system uses the XAIJustificationAgent to fetch the contextual reasoning based on the user's profile and displays the explanation.



The user wants to change the parameters of the current trip - for example, how adventurous or cheap they want the trip to be. They can talk to the LLM about it and have the LLM update the settings, or can manually change their user settings.



After completing a trip, the user submits a text review about their experience or manually clicks a star rating for a specific route. The UserFeedbackAgent processes this sentiment to automatically adjust the user's preference weights for future trips.



The user queries the chatbot about the current or forecasted weather for a specific waypoint, or clicks a weather icon on the UI. The AI utilizes the weatherAgent to retrieve the live data and passes it to the frontend to render a weather widget.

5. Glossary

- **Feasible Route:** A route calculated by the system whose total cost, duration, and composition satisfy the hard constraints defined in the TripRequest (e.g., staying within maxBudgetMin, not exceeding maxStops, and including all mandatoryTypes)
- **Mandatory Types:** A specific set of POI categories (e.g., "Museum," "Restaurant") that the user has explicitly requested. The route generation algorithm treats these as hard constraints that must appear in the final itinerary.
- **Reroll (Indexed POI):** An operation that allows a user to reject a specific location in a generated itinerary. The system replaces exactly one POI at the specified index while keeping all other locked POIs fixed, recomputing only the affected travel segments and totals to maintain route feasibility.
- **User Vector:** A data structure received from the frontend that encodes the user's current session preferences, including derived signals for pace, social preference, interest

priorities, and budget constraints. This serves as the input for the route generation algorithms.

- **XAI (Explainable AI):** A system capability and ethical requirement handled by the XAIJustificationAgent to generate text-based justifications linking suggested POIs to user preferences to build trust and transparency.
- **Tool Calling:** A paradigm where the system uses pretested, statically defined function templates (agents) and allows the AI model to dynamically decide which function should be invoked with specific parameters. This approach fuses statically tested codes into dynamic routing capabilities.
- **Microservices Architecture:** The backend structural design where responsibilities are divided into isolated, highly reliable services (e.g., User Service, Places Service) communicating via REST-like HTTP endpoints.
- **Agent**: A specialized, functional module that empowers an AI to take concrete actions rather than simply generating conversational text. Enables the LLM to make use of tool-calls through the logic implemented within each AI agent. Agents are typically built on a unified template.
- **JWT (JSON Web Token):** A standard used by the system's authentication state to securely manage token-based, persistent user sessions across the microservices.
- **OSRM (Open Source Routing Machine):** The external service used by the Route Generation module to perform fast, heuristic routing calculations (computing travel distances and durations) over mathematically perfect but slower "Traveling Salesperson" calculations.
- **Leaflet:** An open-source JavaScript library utilized by the MapPanel component to render interactive maps, custom-colored destination markers, and route paths on the client interface.
- **Travel Persona:** A reusable travel preference profile stored in the database that defines a user's general travel tendencies (e.g., travel styles, interests, frequency, and pace). This informs AI recommendations before explicit, session-specific trip constraints are set.
- **POI (Point of Interest):** A specific geographical location (such as a museum, restaurant, or park) that a user might want to visit. In the system, it serves as the core, immutable data entity managed by the Places Service.
- **TripRequest:** A session-specific internal data snapshot constructed from the User Vector. It stores numeric limits (e.g., budget, maximum stops) and mandatory category requirements used to initiate and constrain the route generation algorithm.
- **Location Card:** An embedded, interactive UI component rendered within an AI chat message that displays structured POI data (image, name, rating, price level) and allows the user to interact with the destination (e.g., view on map, save to bookmarks) directly from the conversation stream.

- **Route Segment:** The computed travel path and metadata between two consecutive ordered visits (RoutePoints) in a generated itinerary. It stores travel attributes such as duration, distance in meters, and optional path geometry for map rendering.

6. References

- [1] IBM, "UML - Basics," June 2003. [Online]. Available: http://www.ibm.com/developerworks/rational/library/769.html. [Accessed: 15-Feb-2026].
- [2] Koç University, "IEEE Style - Referencing & Citation Styles," December 2023. [Online]. Available: https://libguides.ku.edu.tr/c.php?g=715112&p=5177877. [Accessed: 15-Feb-2026].
- [3] Project OSRM, "Open Source Routing Machine," 2024. [Online]. Available: http://project-osrm.org/. [Accessed: 15-Feb-2026].
- [4] OpenAI, "Chat Completions API Guide," 2024. [Online]. Available: https://platform.openai.com/docs/guides/chat. [Accessed: 15-Feb-2026].
- [5] Leaflet, "Leaflet - an open-source JavaScript library for mobile-friendly interactive maps," 2024. [Online]. Available: https://leafletjs.com/. [Accessed: 15-Feb-2026].
- [6] Spring, "Spring Boot," 2024. [Online]. Available: https://spring.io/projects/spring-boot. [Accessed: 15-Feb-2026].
- [7] Musync, Low Level Design Project Sample, Low Level Design Report Sample (Senior Design Project - Musync). [Accessed: 15-Feb-2026]
- [8] PlantUML, Ashley's PlantUML Documentation, 2025. [Online]. Available: https://plantuml-documentation.readthedocs.io/en/latest/. [Accessed: 15-Feb-2026]
- [9] A. Bandi, B. Kongari, R. Naguru, S. Pasnoor, and S. V. Vilipala, "The Rise of Agentic AI: A Review of Definitions, Frameworks, Architectures, Applications, Evaluation Metrics, and Challenges," *Future Internet*, vol. 17, no. 9, p. 404, 2025. [Online]. Available: https://doi.org/10.3390/fi17090404. [Accessed: 15-Feb-2026].