CSC 431

# Wanderlust Travel Recommender

# System Architecture Specification (SAS)

**Team 07**


| | |
|---|---|
| Cecile Darwiche | Scrum Master |
| Michael Sampino | System Architect |
| Sydney Divozzo | Requirements Engineer |

# Version History

| Version | Date | Author(s) | Change Comments |
|---|---|---|---|
| **1.0** | 4/3/2024 | Cecilce Darwiche, Michael Sampino, Sydney Divozzo | First draft |
| **1.1** | 5/1/24 | Cecilce Darwiche, Michael Sampino, Sydney Divozzo | Final draft - Made changes per TA's comments |
| | | | |
| | | | |

# Table of Contents

# Table of Tables

# Table of Figures

# 1. System Analysis

## 1.1 System Overview

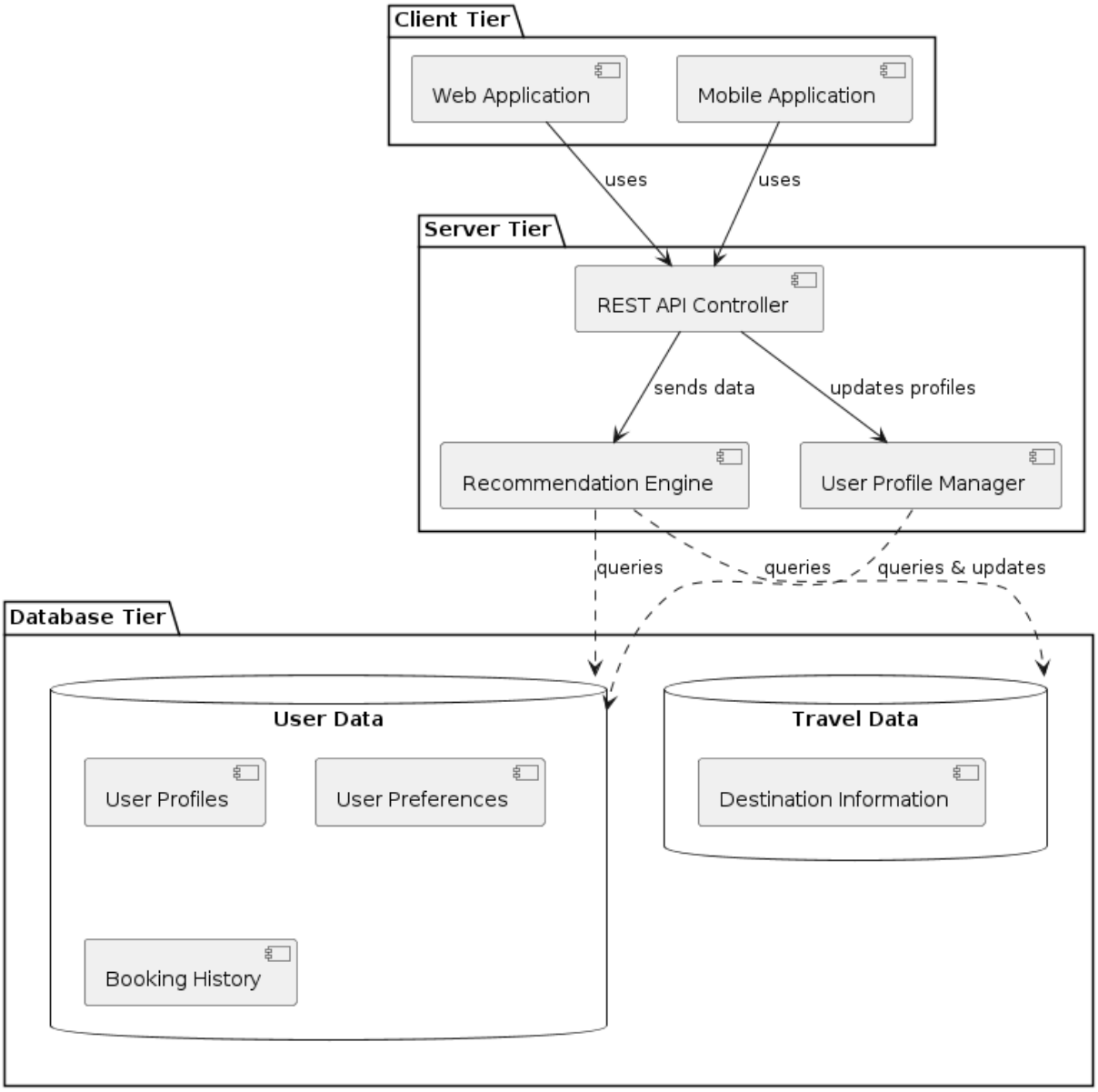
The Wanderlust Travel Recommender is a comprehensive web and mobile application designed to provide personalized travel recommendations and itinerary planning services to users worldwide. Built upon a three-tier architecture, the system seamlessly integrates client-side interfaces, server-side logic, and backend data storage to deliver a dynamic and intuitive user experience.

Leveraging a three-tier architecture, the system distinguishes client, server, and database tiers. Web and mobile applications at the client tier connect to the server tier, where application logic resides. The database tier manages crucial travel data, ensuring efficient storage and retrieval.

This three-tier architecture is crucial for supporting the robust needs of the Wanderlust Travel Recommender System, ensuring that it can handle large volumes of user interactions and data processing with high efficiency and reliability.

## 1.2 System Diagram

Figure 1: System Diagram




## 1.3 Actor Identification

Table 1: Actor Identification

| | |
|---|---|
| New users | Individuals who wish to create an account. |
| Register users | Individuals looking for personalized travel recommendations. |

| | |
|---|---|
| Administrators | System operators responsible for managing the application's backend and ensuring data integrity. |

## 1.4 Design Rationale

### 1.4.1 Architectural Style

Wanderlust Travel Recommender will use a three-tier architecture in this form:

- Client Tier: The client tier comprises web and mobile applications used by travelers to access the system's functionalities. These applications communicate with the server tier via RESTful API endpoints to request travel recommendations and submit user preferences.
- Server Tier: The server tier hosts the application logic and REST API controller responsible for processing client requests, generating recommendations, and orchestrating system operations. It leverages event-driven architecture to handle asynchronous events and decouple system components.
- Database Tier: The database tier manages the storage and retrieval of travel-related data, including user profiles, preferences, booking history, and destination information. It supports relational and non-relational databases to accommodate various data types and structures.

Benefits of the three-tier architecture:

- Scalability: Each tier can be scaled independently, allowing for more precise resource management and improved handling of increased loads without affecting the other tiers.
- Security: By separating the business logic from the data storage and client interface, the system enhances security measures by restricting direct access to the database and the business logic layer.
- Maintainability: Updates and maintenance can be performed on one tier without significant downtime or disruption to the other tiers, facilitating easier updates and bug fixes.

### 1.4.2 Design Pattern(s)

These are the design patterns that we have deemed applicable for this architecture format:

- The **Factory Method Pattern** allows for the creation of complex objects, like recommendation engines or data processing modules, by defining an interface for object creation and enabling subclasses to override the instantiation process. For example, Wanderlust could utilize the Factory Method Pattern to instantiate different types of recommendation engines or data processing modules based on user preferences or system requirements. By defining an interface for object creation and allowing

subclasses to override the instantiation process, Wanderlust can achieve flexibility and extensibility in creating and customizing recommendation algorithms or data processing components tailored to different user needs and preferences.

- The **Strategy Pattern** enables interchangeable algorithms within the recommendation engine, facilitating dynamic selection and switching of algorithms at runtime based on user preferences or system requirements. This enhances flexibility and maintainability in handling recommendation logic for Wanderlust.
  Wanderlust could utilize the Strategy Pattern to implement interchangeable algorithms within its recommendation engine. These algorithms could vary based on factors such as user preferences, travel history, budget constraints, and destination interests. By encapsulating each algorithm as a separate strategy, Wanderlust gains the flexibility to dynamically select and switch between strategies at runtime, depending on the user's input or system requirements.

- The **Facade Pattern** simplifies interactions with a complex subsystem by providing a unified interface.It promotes loose coupling, enhances maintainability, and improves readability by encapsulating the subsystem's intricacies behind a single facade.
  For example, Wanderlust could implement a facade to provide a simplified interface for complex operations, such as interacting with the database or coordinating multiple subsystems. By encapsulating the complexity behind a facade, Wanderlust promotes loose coupling between classes and modules, enhances maintainability, and improves readability within the system's implementation.

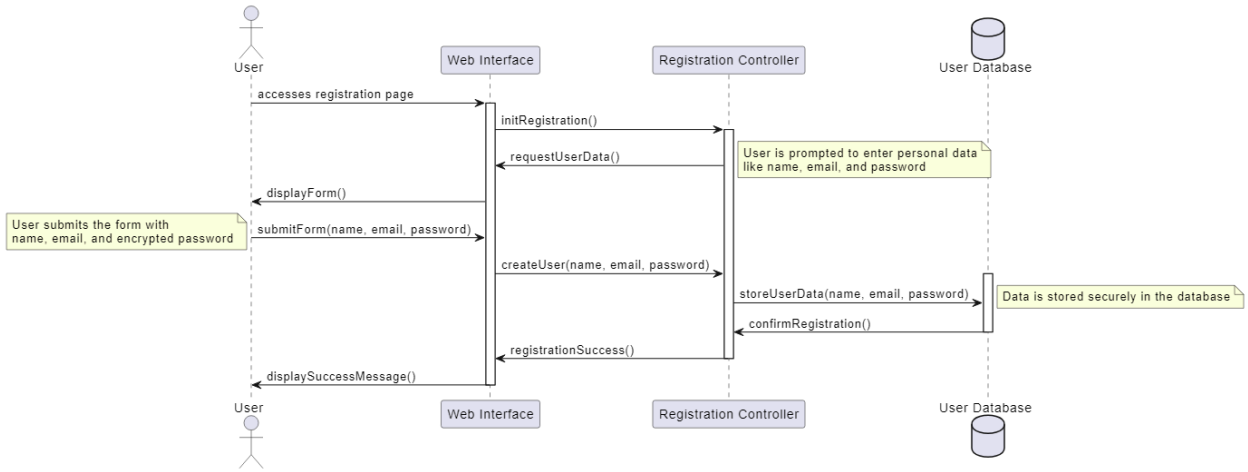
### 1.4.3 Framework

We're leveraging Node.js on the backend, complemented by Express.js for its impressive ability  to manage asynchronous requests and its expansive middleware capabilities. This setup ensures our server-side is both robust and flexible. On the frontend, React keeps our user interface responsive, seamlessly updating and rendering exactly what's needed for an engaging user experience. We've anchored our system with PostgreSQL for data storage, chosen for its reliability and efficiency in managing complex queries, ensuring our application is not just powerful but also scalable. Together, these frameworks and technologies form the foundation of our platform, ensuring a smooth and dynamic experience for every traveler.

# 2. Functional Design

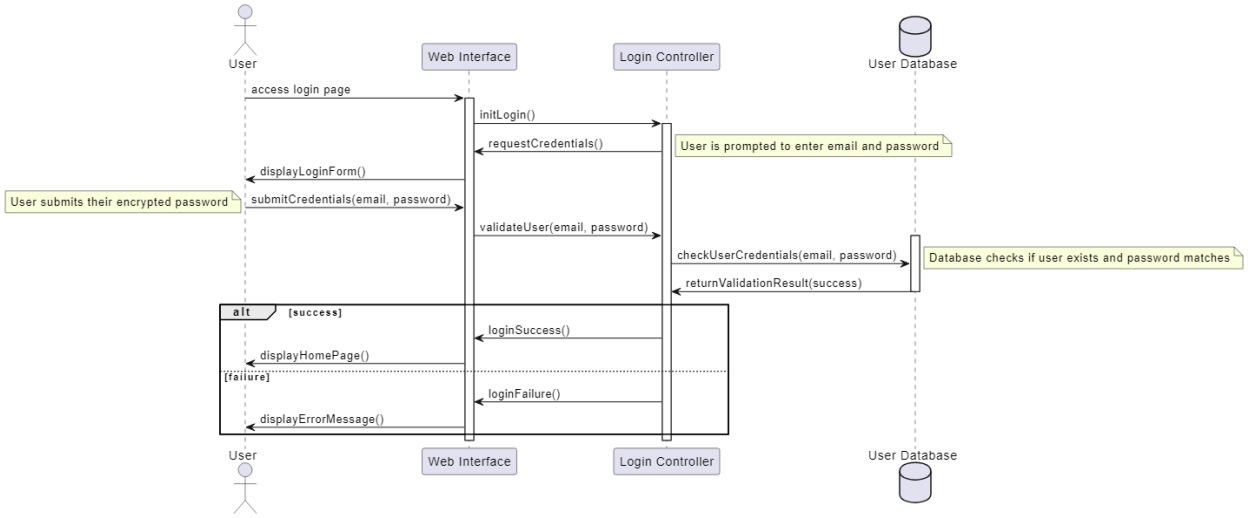
## 2.1 User Registration Sequence Diagram

Figure 2: User Registration Diagram



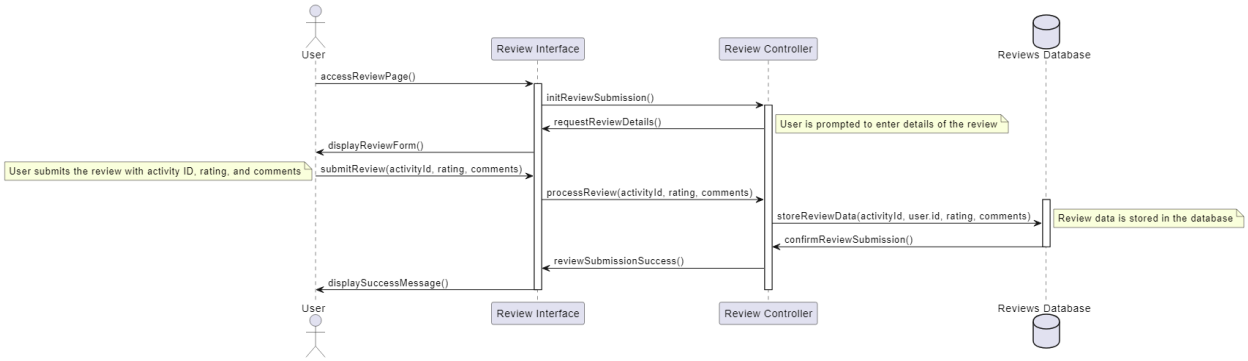

## 2.2 User Login Sequence Diagram

Figure 3: User Login Sequence Diagram



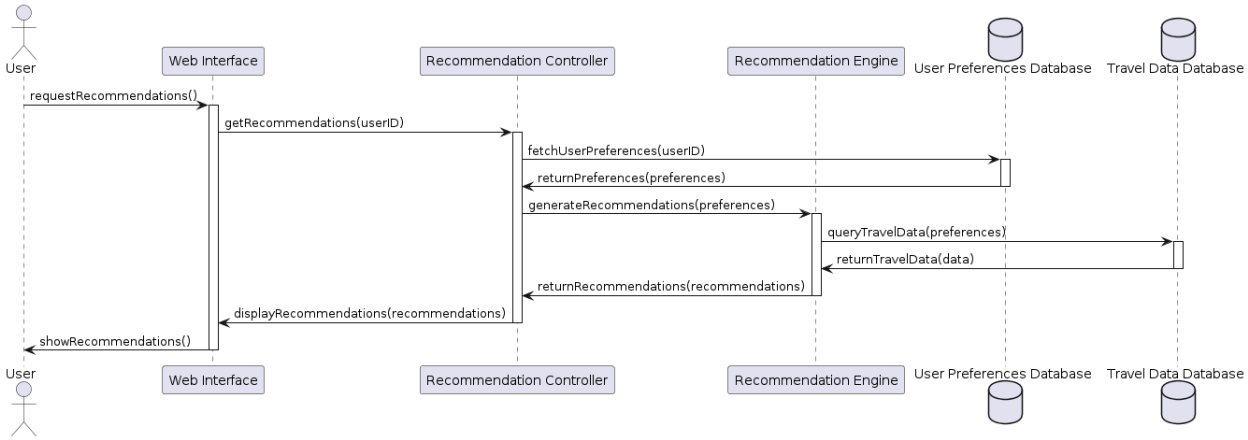
## 2.3 Submit Review Sequence Diagram

Figure 3: Submit Review Sequence Diagram




## 2.4 Recommendation Engine Sequence Diagram

Figure 4: Recommendation Engine Sequence Diagram



# 3. Structural Design - Class Diagram

Figure 4: Structural Design - Class Diagram