Abstract

This project presents the development of a Travel Planner web application designed to streamline the process of planning and managing travel. The application includes features such as user sign-in and sign-up, ticket booking, hotel reservations, landmark exploration, and comprehensive travel planning. Additionally, a "Contact Us" section allows users to communicate directly with the admin for support and inquiries. The platform enables general users to book services and offer their own, enhancing the overall travel experience. The application incorporates standard web functionalities to ensure a seamless user experience. Design patterns have been applied to ensure scalability, maintainability, and efficient operation of the system.

Contents

Introduction

1.1 Project Overview

The Travel Planner web application is designed to provide an all-in-one platform for planning trips, booking tickets, and managing related services. It aims to simplify the travel planning process by integrating various functionalities into a single, cohesive system. Users can sign in or sign up to access personalized features, book tickets for different modes of transportation, reserve hotels, explore landmarks, and create detailed travel itineraries. The application also includes a ”Contact Us” section, allowing users to communicate directly with the admin for support and inquiries. This document outlines the design, implementation, and application of software design patterns in the system, ensuring a robust and efficient architecture.

1.2 Objectives

- Provide a user-friendly interface for planning and managing travel, catering to both novice and experienced travelers.
- Apply design patterns to enhance code quality, reusability, and maintainability, ensuring the system can evolve with user needs.
- Ensure scalability to accommodate a growing user base and expanding feature set.
- Facilitate seamless integration of additional services, such as travel insurance and local guides.
- Enhance user engagement by offering personalized travel suggestions and recommendations.

System Design

2.1 Architecture Overview

The application is based on a client-server architecture, ensuring a clear separation of concerns and facilitating scalability. The frontend is developed using React, providing a dynamic and responsive user interface. It communicates with the backend through REST APIs, which are implemented using Spring Boot. This setup allows for efficient data exchange and seamless integration of new features.

2.2 Modules

The system is divided into several key modules, each responsible for specific functionalities:

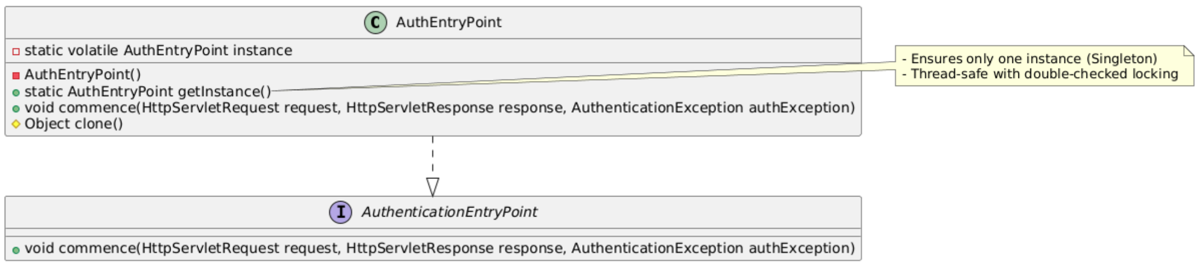
- **Ticket Booking:** This module enables users to book tickets for various modes of transportation, including buses, trains, and flights. It integrates with external APIs to provide real-time availability and pricing.
- **Hotel Booking:** Users can search for and reserve hotels through this module. It offers filtering options based on location, price, and amenities, ensuring a tailored booking experience.
- **Landmarks:** This module provides travel suggestions based on the user's location and preferences. It includes information on popular tourist attractions, historical sites, and local events.
- **Accounts:** Responsible for user authentication and profile management, this module ensures secure access to the application. It supports features like sign-up, sign-in, password recovery, and profile updates.
- **Services:** This module includes additional utilities such as travel insurance, car rentals, and local guides. It aims to enhance the overall travel experience by offering comprehensive service options.
- **Contact Us:** Facilitates communication between users and the admin team. It allows users to submit inquiries, feedback, and support requests, ensuring responsive customer service.

Each module is designed to be modular and extensible, allowing for easy updates and integration of new features as the application evolves.

Design Patterns

3.1 Singleton Design Pattern

The diagram illustrates the implementation of the Singleton design pattern in the ‘AuthEntryPoint‘ class of our travel web application. The pattern ensures that only a single instance of ‘AuthEntryPoint‘ exists throughout the application’s lifecycle, which is crucial for handling authentication consistently. The class uses a thread-safe, double-checked locking mechanism to prevent multiple instantiations in a multi-threaded environment. By extending the ‘AuthenticationEntryPoint‘ interface, it provides a centralized authentication entry point to handle unauthorized access attempts efficiently.



AuthEntryPoint
static volatile AuthEntryPoint instance
AuthEntryPoint()
static AuthEntryPoint getInstance()
void commence(HttpServletRequest request, HttpServletResponse response, AuthenticationException authException)
Object clone()
- Ensures only one instance (Singleton)
- Thread-safe with double-checked locking
AuthenticationEntryPoint
void commence(HttpServletRequest request, HttpServletResponse response, AuthenticationException authException)

Figure 3.1: Singleton Design Pattern in the Travel Planner Application

3.2 Factory Design Pattern

The diagram illustrates the Factory Design Pattern implemented in the ticket booking system of our travel web application. The ‘TicketFactory‘ class is responsible for creating different types of tickets based on the provided type (e.g., ”Bus”, ”Flight”, ”Train”). The factory method ‘createTicket(type: String)‘ returns an instance of the ‘Ticket‘ interface, which is implemented by concrete classes: ‘BusTicket‘, ‘FlightTicket‘, and ‘TrainTicket‘.

Each ticket type provides its own implementation of the ‘book()‘ method to handle the specific booking logic. This design pattern promotes loose coupling by centralizing object creation, making the system more flexible and easier to extend when adding new ticket types in the future.

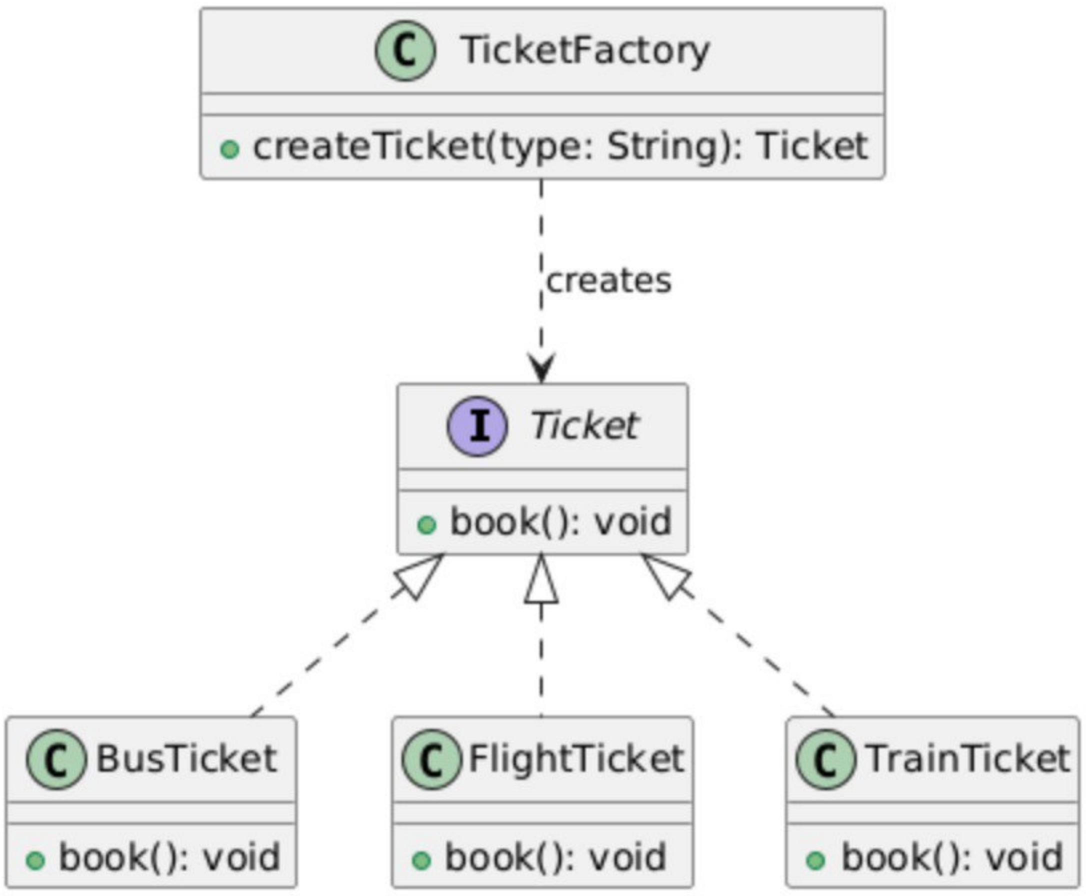


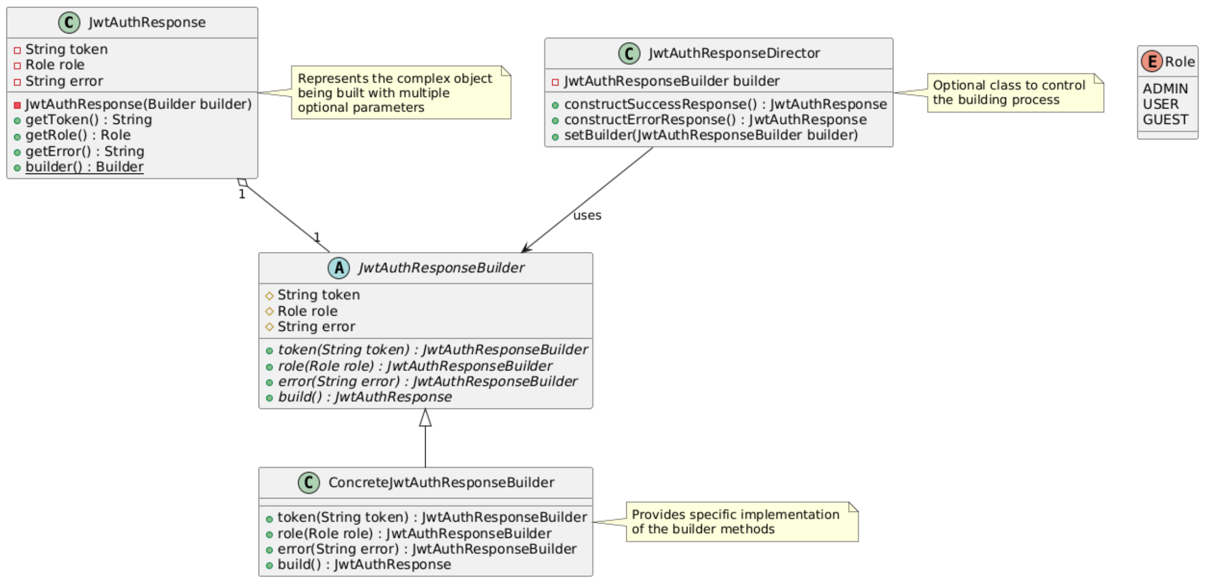
Figure 3.2: Factory Design Pattern in the Travel Planner Application

3.3 Builder Design Pattern for JWT Authentication Response

The diagram represents the Builder Design Pattern used to construct `JwtAuthResponse` objects in the travel web application. This pattern allows the creation of complex authentication responses with optional parameters in a step-by-step manner.

Key Components

- **JwtAuthResponse (Product):** Holds token, role, and error fields, built via a builder.
- **JwtAuthResponseBuilder (Abstract Builder):** Defines methods to set fields and build the response.
- **ConcreteJwtAuthResponseBuilder (Concrete Builder):** Implements the builder logic.
- **JwtAuthResponseDirector (Director):** Controls the building process for different response types (success/error).
- **Role (Enum):** Defines user roles (`ADMIN`, `USER`, `GUEST`).



C JwtAuthResponse
String token
Role role
String error
JwtAuthResponse(Builder builder)
getToken() : String
getRole() : Role
getError() : String
builder() : Builder
Represents the complex object being built with multiple optional parameters
C JwtAuthResponseDirector
JwtAuthResponseBuilder builder
constructSuccessResponse() : JwtAuthResponse
constructErrorResponse() : JwtAuthResponse
setBuilder(JwtAuthResponseBuilder builder)
Optional class to control the building process
E Role
ADMIN
USER
GUEST
1
1
uses
A JwtAuthResponseBuilder
String token
Role role
String error
token(String token) : JwtAuthResponseBuilder
role(Role role) : JwtAuthResponseBuilder
error(String error) : JwtAuthResponseBuilder
build() : JwtAuthResponse
C ConcreteJwtAuthResponseBuilder
token(String token) : JwtAuthResponseBuilder
role(Role role) : JwtAuthResponseBuilder
error(String error) : JwtAuthResponseBuilder
build() : JwtAuthResponse
Provides specific implementation of the builder methods

Figure 3.3: Builder Design Pattern in the Travel Planner Application

3.4 Facade Design Pattern

The facade design pattern in this travel web app simplifies user management by providing a unified interface through the ‘UserFacade‘ class. It acts as an intermediary between the ‘UserController‘ and the complex underlying system, which includes ‘UserServiceImpl‘ and various data transfer objects (DTOs) like ‘UpdateProfileDto‘ and ‘LoginDto‘. The facade offers streamlined methods for updating user profiles, changing passwords, and retrieving user information, encapsulating the complexity of the service layer and enhancing maintainability and scalability.

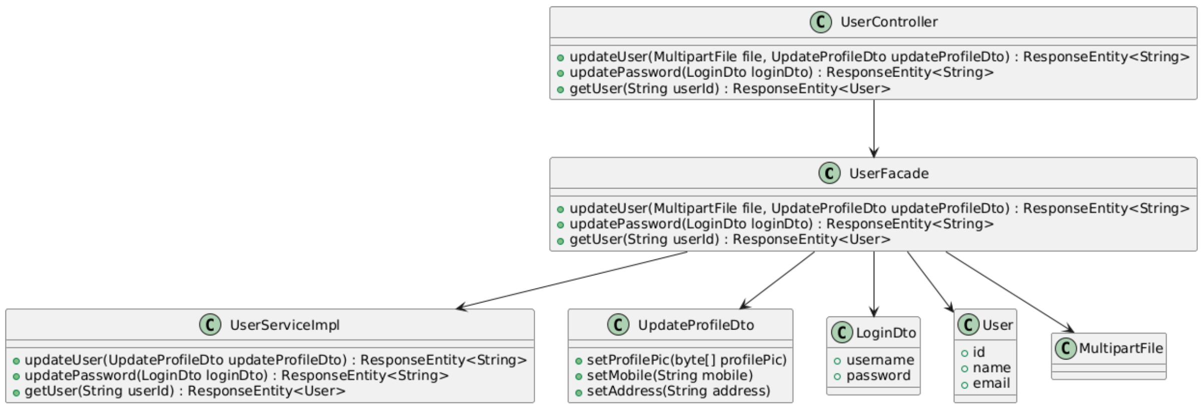

UserController
updateUser(MultipartFile file, UpdateProfileDto updateProfileDto) : ResponseEntity<String>
updatePassword(LoginDto loginDto) : ResponseEntity<String>
getUser(String userId) : ResponseEntity<User>
UserFacade
updateUser(MultipartFile file, UpdateProfileDto updateProfileDto) : ResponseEntity<String>
updatePassword(LoginDto loginDto) : ResponseEntity<String>
getUser(String userId) : ResponseEntity<User>
UserServiceImpl
updateUser(UpdateProfileDto updateProfileDto) : ResponseEntity<String>
updatePassword(LoginDto loginDto) : ResponseEntity<String>
getUser(String userId) : ResponseEntity<User>
UpdateProfileDto
setProfilePic(byte[] profilePic)
setMobile(String mobile)
setAddress(String address)
LoginDto
username
password
User
id
name
email
MultipartFile

Figure 3.4: Facade Design Pattern in the Travel Planner Application

3.5 Adapter Design Pattern

This design utilizes the adapter pattern to integrate payment processing in the travel web app. The ‘StripePaymentServiceAdapter‘ class implements the ‘PaymentService‘ interface, allowing the creation of payment links through Stripe. It uses ‘PaymentLinkCreateParams‘ to configure the payment details and generates a ‘PaymentLink‘ object, which provides the payment URL. This approach abstracts the complexities of Stripe’s API, offering a simple method to create payment links while maintaining flexibility for future payment service integrations.

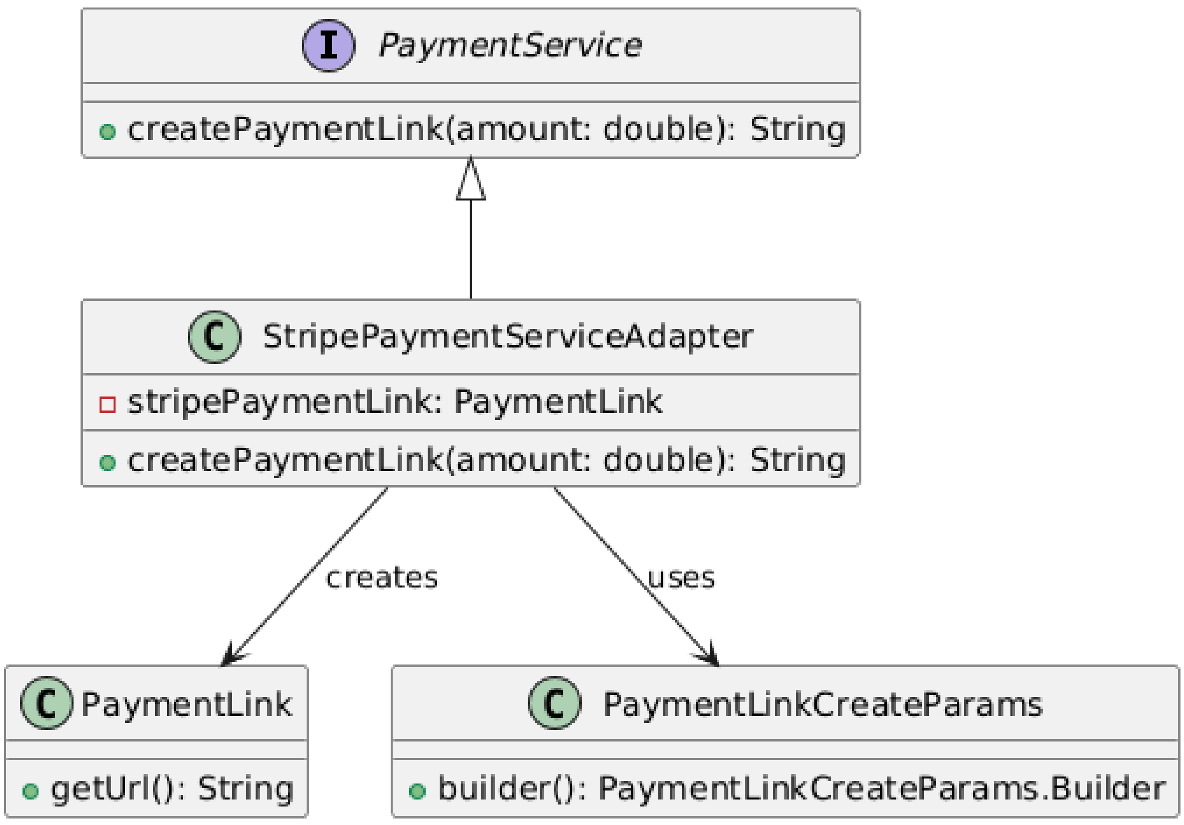

I PaymentService
createPaymentLink(amount: double): String
C StripePaymentServiceAdapter
stripePaymentLink: PaymentLink
createPaymentLink(amount: double): String
creates
uses
C PaymentLink
getUrl(): String
C PaymentLinkCreateParams
builder(): PaymentLinkCreateParams.Builder

Figure 3.5: Adapter Design Pattern in the Travel Planner Application

Implementation

4.1 Frontend (React)

React components are used to create a dynamic and responsive user interface. Key libraries include Axios for API calls and React Router for navigation.

4.2 Backend (Spring Boot)

Spring Boot provides RESTful endpoints for the application. Security is implemented using JWT, and Stripe integration is used for payment. MySQL Database is used to store the data.

Conclusion

The Travel Planner web application demonstrates the effective use of design patterns in building a scalable and maintainable system. Future work may include adding AI-based travel suggestions and more advanced payment options.

References

- Github Repository Link : `https://github.com/NirjharSingha/TravelApp`
- Video demonstration: `https://youtu.be/wxOPHrh79Ps`