**Design Document**

**Table of Contents**

1. Class Diagram (CRC Cards)
2. System Interaction with the Environment
3. High Level Architecture Overview
4. System Decomposition and Error/Exception Strategy
5. **Class Diagram (CRC Cards)**

|  |
| --- |
| Class Name: **User** |
| Parent Class: None  Subclass: None |
| Responsibilities:   * Store and manage user information (username, email, password, phone, etc.). * Maintain lists of visited countries. * Maintain passport information if needed (e.g., user could have multiple passports). * Provide methods or properties to access/update user’s travel data. |
| Collaborators:   * **AuthService**: for handling registration/login. * **TravelService**: for marking visited countries and fetching visited percentage. * **VisaService**: for looking up visa requirements based on user’s passports. |

|  |
| --- |
| Class Name: **AuthService** |
| Parent Class: None  Subclass: None |
| Responsibilities:   * Handle user registration (POST /register). * Handle user login (POST /login). * Verify credentials (username/email and password). * Manage session data or tokens once a user is authenticated. |
| Collaborators:   * **User**: AuthService creates and validates User objects. |

|  |
| --- |
| Class Name: **VisaService** |
| Parent Class: None  Subclass: None |
| Responsibilities:   * Process visa requirements (POST /visa-requirements) based on the user’s passports. * Determine the “best visa option” for each destination. * Return relevant visa info (visa required, visa on arrival, etc.). |
| Collaborators:   * **User**: for accessing user’s passport list. * **Passport:** For detailed passport data. * **Country**: for retrieving country-specific visa rules. |

|  |
| --- |
| Class Name: **TravelService** |
| Parent Class: None  Subclass: None |
| Responsibilities:   * Generate travel suggestions (POST /travel-suggestions) given a user’s budget. * Track user’s visited countries:   + Add or remove a visited country (POST /markVisitedCountries).   + Calculate the percentage of visited countries (GET /visitedCountriesPercent). |
| Collaborators:   * **User**: for retrieving and updating visited-country data and budget info. * **Country**: for storing or looking up country details (e.g., codes, names, region). |

|  |
| --- |
| Class Name: **Passport** (not sure if we need) |
| Parent Class: None  Subclass: None |
| Responsibilities:   * Hold passport-specific data (passport number, type, issuance country, expiry). * Provide methods to verify passport validity or type. |
| Collaborators:   * **User**: each user may have one or more Passports. * **VisaService**: to evaluate visa requirements based on passport details. |

|  |
| --- |
| Class Name: **Country** (not sure if we need) |
| Parent Class: None  Subclass: None |
| Responsibilities:   * Store country data (country code, name, region, etc.). * Track whether a user has marked it as visited (if you keep a per-country record). * Provide data for other services (VisaService, TravelService). |
| Collaborators:   * **TravelService**: to check and update visited status. * **VisaService**: to confirm visa requirements. |

1. **System Interaction with the Environment**

**Operating Environment and Dependencies:**

* **Operating System**: The application is assumed to run on a Unix-like environment (e.g., Linux) or a Windows server with Node.js installed.
* **Programming Language / Framework**: Node.js (v16+) with Express.js (for REST endpoints).
* **Database**: A relational (PostgreSQL) database for persistent storage of user accounts, country data, and visited-country records.
* **Virtual Machines / Containers**: The application may be deployed in Docker containers.
* **Network Configuration**: Standard HTTP/HTTPS traffic; the service exposes ports 4000 (configurable) and 3000 (configurable).
* **External APIs / Services**: The system requires external data (e.g., real-time visa rules or travel cost info), it will connect via secure API requests.

**Assumptions:**

* Adequate network reliability and bandwidth for communicating with the database or any external APIs.
* A modern web browser for the front-end.
* Node.js environment must have all required dependencies installed (Express, body-parsers, session libraries, etc.).

1. **High Level Architecture Overview**

Frontend (REACT)

REST API (Express)

AuthService, VisaService, TravelService

Database (postgreSQL)

External APIs

openAI, Expedia (later)

**Assumptions:**

* **Frontend**: A web or mobile client that interacts via REST calls.
* **REST API / Express:**
  + **AuthService** manages user registration, login, and session handling.
  + **VisaService** processes requests for visa requirements using user passport data.
  + **TravelService** handles travel suggestions, marking visited countries, and calculating visited-country percentages.
* **Database**: Stores user records, passport data, visited country lists, and pre-defined country/visa information.
* **External Services:** Could include third-party APIs for real-time travel advisories, flight/hotel info.

1. **System Decomposition and Error/Exception Strategy**

**System Decomposition:**

* **AuthService**:
  + **Role**: Validates credentials, handles new user registration and login, and maintains session or token data.
  + **Interactions**: Communicates with the database (e.g., Users table/collection) to create accounts and verify passwords.
* **VisaService**:
  + **Role**: Receives a list of user passports and returns visa requirements for each destination.
  + **Interactions**: May consult an internal (local) data store or external API to fetch country-specific visa rules.
* **TravelService**:
  + **Role**: Generates travel suggestions based on user’s budget; tracks and calculates visited countries.
  + **Interactions**: Reads/writes user data in the database to update visited-country lists; calculates visited-percent metrics.
* **Database Layer**:
  + **Role**: Provides persistent storage for user profiles, countries, visited records, and any needed reference data.
  + **Interactions**: Exposed through repositories/ORM or direct queries from each service.

**Error and Exception Handling:**

* **Invalid User Input**:
  + **Strategy**: Validate incoming JSON payloads (e.g., username/email format, password length, budget must be numeric).
  + **Response**: Return HTTP 400 (Bad Request) with a descriptive error message object ({ errors: [{ msg: ... }] }).
* **Authentication/Authorization Failures**:
  + **Strategy**: If a user is not logged in or a session is invalid, respond with HTTP 401 (Unauthorized).
  + **Response**: Provide a clear error message and instructions to log in again.
* **External System Failures** (e.g., third-party visa API unavailable):
  + **Strategy**: Implement graceful degradation—return partial results or a fallback response if external data is temporarily unavailable.
  + **Response**: Return HTTP 500 if the error is unrecoverable, or a 503 (Service Unavailable) if an external dependency is down.
* **Database Connectivity Issues**:
  + **Strategy**: Use retry logic or circuit breakers, depending on severity.
  + **Response**: Return HTTP 500 (Internal Server Error) and log the event for troubleshooting.
* **Unexpected Exceptions**:
  + **Strategy**: Catch unhandled exceptions at a global error handler (e.g., app.use(...) in Express). Log details for debugging.
  + **Response**: Return a generic HTTP 500 (Internal Server Error) message to the user, without exposing sensitive stack traces.