**1.Introduction**

**Project Title:**

**TRIPTREK: YOUR ULTIMATE JOURNEY PLANNER**

**Team Members:**

SNEHA GANJI - TEAM LEADER

HARISH BABU ADINA - TEAM MEMBER 1

AARTHI ATTALURI - TEAM MEMBER 2

SHAIK MAHABOOB SUBHANI - TEAM MEMBER 3

**2. PROJECT OVERVIEW**

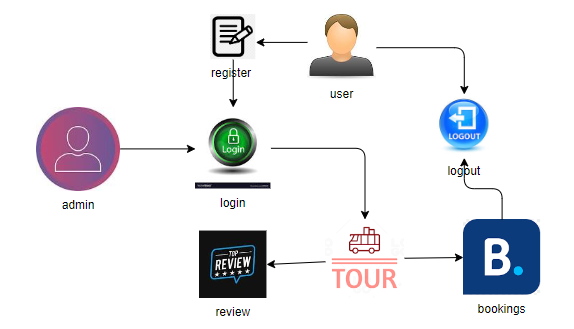
**Purpose:**

TripTrek project provides great chances for users to obtain rich information. Particularly, tourists want to easily get tremendous information about their tourism plans. Therefore, in this paper, we aim to design a tourism resources management system and guide the tourists to plan their travel routes. In general, if any user plans for any tour they must consult any traveling agency for purchasing package. At the same time the user has to do traveling reservations from source to destination, hotel reservations at destination place and other travel reservations from destination to other places, this involves lot of manual work. For all these types of reservations must be done by user by standing lot of time in a queue. To avoid these problems to reduce the manual work to the user new project was developed i.e. TripTrek.

**Key Features:**

* Personalized Itinerary Creation
* Destination Discovery
* Trip Budgeting Tool
* Travel Community
* Real-Time Updates
* Multi-Mode Transportation Options

**3. Architecture**

**Front end:**

Here frontend is responsible for the user interface (UI) and client-side logic. It’s developed using React, which is a popular JavaScript library for building interactive UIs.

**1.Component-based Architecture:**

React follows a component-based architecture, meaning the entire UI is divided into smaller, reusable pieces called components. These components manage their own logic and rendering.

* Container Components: These components are responsible for handling the logic of the application, fetching data from the backend, managing state, and passing down the necessary props to child components.
* Presentational Components: These components focus on the UI part (i.e., how the content is displayed) and are often stateless.

**2.State Management:**

* Local State: Managed using React Hooks like use State and use Effect to control UI elements, handle user input, and manage side effects such as data fetching.
* Global State: Managed using React Context API or third-party tools like Redux to maintain and share data between different components without passing props down multiple levels.

**Back End:**

The backend is developed using Node.js with Express.js, which handles the server-side logic and provides the APIs required by the frontend to interact with data and services.

**Key Components of the Backend Architecture:**

**1.Node.js and Express.js:**

* **Node.js** is a JavaScript runtime that allows JavaScript to be used on the server-side. It handles asynchronous operations like file handling, database queries, and API calls.
* **Express.js** is a lightweight, fast web framework built on top of Node.js. It is used to build RESTful APIs and manage routing.

**2.API Endpoints:**

* **REST API:** The backend provides several endpoints for the frontend to interact with. Examples included
* **GET /api/trips:** Fetch all trips for a particular user.
* **POST /api/trips:** Create a new trip.
* **PUT /api/trips/:id:** Update an existing trip.
* **DELETE /api/trips/:id:** Delete a trip.

**Database:**

* **MongoDB** is used as the database to store user information, trip details, bookings, and other related data.

**Key Components of the Database Architecture:**

**1.MongoDB (NoSQL Database):**

* MongoDB stores data in the form of documents (JSON-like format), which are grouped into collections.
* Mongoose is an ORM (Object-Relational Mapping) library that provides a schema-based solution to model data. It ensures that the data in the database adheres to certain rules and relationships.

**2.CRUD Operations:**

* **Create**: When a user creates a new trip, a new document is added to the Trips collection.
* **Read**: The frontend fetches data from the database via API calls, which query the database using Mongoose.
* **Update:** When a user modifies a trip, the corresponding document is updated.
* **Delete**: If a user deletes a trip, the document is removed from the database.

**4.Setup Instructions**

* **Prerequisites:**

Node.js (v16+)

MongoDB (v4+)

Git

* **Installation:**

1. Clone the repository:

git clone https://github.com/your-repo/triptrek.git

1. Navigate to the project folder:

cd trip trek

1. Install dependencies for both client and server:

npm install

cd client

npm install

1. Set up environment variables:

Create a .env file in the root directory and define variables like

MONGO\_URI, JWT\_SECRET, and PORT.

**5.Folder structure**

**Client:**

client/

│

├── public/

│ ├── index.html

│ ├── favicon.ico

│ ├── manifest. json

│ └── assets/

│ └── images/

│ └── logo.png

│

├── src/

│ ├── assets/

│ │ └── images/

│ ├── components/

│ │ ├── Header.js

│ │ ├── Footer.js

│ │ └── TripList.js

│ │

│ ├── pages/

│ │ ├── Home.js

│ │ ├── Login.js

│ │ └── TripPlanner.js

│ │

│ ├── context/

│ │ └── AuthContext.js

│ │

│ ├── hooks/

│ │ └── useAuth.js

│ │

│ ├── services/

│ │ └── api.js

│ │

│ ├── App.js

│ ├── index.js

│ └── App.css

│

└── package. json

**1.public/**

* Contains static assets that are served directly. It includes:
* **index.html:** The main HTML file where the React app is mounted.
* **favicon.ico:** The favicon for the app.
* **manifest.json:** Used for Progressive Web Apps (PWA) to manage app metadata.
* **assets/:** Contains static resources such as images or logos.

**2.src/ (Source folder):**

* This folder contains the actual source code for the React application.
* **assets/:** Holds images, fonts, or other media files used in the frontend.
* **components/:** Contains reusable components like Header.js, Footer.js, and TripList.js. Components are small, independent UI blocks used throughout the app.
* **pages/:** Holds larger, page-specific components such as Home.js, Login.js, and TripPlanner.js. Each file represents a separate page or view in the app.
* **context/:** Contains the Context API files used for global state management, such as AuthContext.js for user authentication state.
* **hooks/:** Custom hooks used in the app, such as useAuth.js, to handle business logic related to authentication or other functionalities.
* **services/:** Contains files like api.js that handle API requests and interact with the backend server.
* **App.js:** The root component that brings together the entire application.
* **index.js:** The entry point of the React app, where App.js is rendered and mounted to the DOM.
* **App.css:** Global CSS for the entire app.

**3.package.json**

* Lists all dependencies, scripts, and project metadata for the React app.

**Server:**

The server directory holds all the code related to the backend of the application, including the APIs, database interaction, and server logic

config/

server/

│

├── config/

│ └── db.js

│

├── controllers/

│ ├── authController.js

│ ├── tripController.js

│ └── userController.js

│

├── middlewares/

│ ├── authMiddleware.js

**│** ├── errorHandler.js

│ └── validateInput.js

│

├── models/

│ ├── User.js

│ └── Trip.js

│

├── routes/

│ ├── authRoutes.js

│ ├── tripRoutes.js

│ └── userRoutes.js

│

├── utils/

│ └── generateToken.js

│

├── .env

├── server.js

├── package.json

└── README.md

**1.config/**

* Contains configuration files, such as db.js which establishes the connection to the MongoDB database.

**2.controllers/**

* Houses the business logic for handling requests and responses for different routes. Each controller corresponds to different parts of the app.
* **authController.js:** Manages user authentication (login, register, etc.).
* **tripController.js**: Handles CRUD operations for trips.
* **userController.js:** manages user-related actions, such as updating the user profile.

**3.middlewares/**

* Holds middleware functions that are used in various routes to perform tasks before passing the control to the controller:
* **authMiddleware.js:** Ensures that users are authenticated before accessing protected routes.
* **errorHandler.js:** Handles errors and sends appropriate responses to the client.
* **validateInput.js:** Validates incoming data from the client to ensure it follows the expected format.

**4.models/**

* Contains Mongoose models that define the structure of the data in MongoDB:
* **User.js:** Defines the user schema and methods for interacting with user documents in MongoDB.
* **Trip.js:** Defines the schema for storing trip details.

**5.routes/**

* Holds all the API route definitions. Each file defines routes related to specific entities:
* **authRoutes.js:** Manages routes related to authentication (/login, /register).
* **userController.js:** Manages routes related to trip operations (/trips, /trips/:id).
* **userRoutes.js:** Manages routes for user actions (/profile, /update).

**6.utils/**

* Stores utility functions that are shared across the project:
* generateToken.js: A function that generates JWT tokens for authentication.

**7..env**

* Stores environment variables like the MongoDB URI, JWT secret, and other sensitive configuration details.

**8.server.js**

* The entry point for the Node.js server. This file initializes the Express app, connects to the MongoDB database, sets up middleware, and starts listening for incoming HTTP requests.

**9.package.json**

* Lists the dependencies and scripts required for the backend (e.g., Express.js, Mongoose, JWT).

**10.README.md**

* A detailed guide or documentation for setting up and running the backend server.

**6.Running the Application**

**1. Pre-requisites**

Before running the application, ensure that you have installed the following:

* **Node.js** (v16+)
* **MongoDB** (either local or Atlas)
* **Git** (to clone the repository)
* **Code editor/IDE** (e.g., Visual Studio Code)
* Set up **environment variables** in the .env files for both the frontend and backend.

**2. Running the Backend (Node.js) Server**

The backend is responsible for handling API requests, user authentication, trip management, and communication with the MongoDB database.

**Steps to run the backend server:**

1.Navigate to the server directory in your terminal:

cd server

2.Install backend dependencies using npm:

npm install

* This command installs all necessary packages specified in the package.json file (e.g., Express.js, Mongoose, JWT, etc.).

3.Set up environment variables by creating a .env file in the root of the server directory. Add the following details to your .env file:

MONGO\_URI=mongodb://localhost:27017/triptrek

JWT\_SECRET=supersecret key

PORT=5000

* MONGO\_URI: Connection string for MongoDB. If you're using MongoDB Atlas, replace it with your Atlas connection string.
* JWT\_SECRET: A secret key for signing JWT tokens used in user authentication.
* PORT: Port number where the backend server will run (usually 5000).

4.Start the MongoDB server (if running locally):

* If you're using MongoDB locally, ensure that it's running. You can start MongoDB by executing:

Mongod

5.Start the backend server:

npm start

* This will start the Node.js server. It should connect to MongoDB and listen for API requests on the specified port (e.g., <http://localhost:5000>).
* Alternatively, for development, you can use Nodemon to automatically restart the server when files are modified:

nodemon

6.Check the backend server:

* Open a browser or API testing tool like Postman and make a request to http://localhost:5000/api/test to ensure the backend is running properly.

**7.API Documentation**

**Overview of API Endpoints:**

* Base URL: <http://localhost:5000/api>
* All API requests are prefixed with /api and use \*JSON\* as the data format.

**Authentication APIs**

1. **POST** /api/auth/register

* **Description:** Registers a new user.
* **Request Body:**

{

"name": "John Doe",

"email": "john@example.com",

"password": "password123"

}

* **Response:**

{

"token": "JWT\_TOKEN",

"user": {

"id": "12345",

"name": "John Doe",

"email": "john@example.com"

}

}

2. **POST** /api/auth/login

* **Description:** Authenticates an existing user and generates a token.
* **Request Body:**

{

"email": "john@example.com",

"password": "password123"

}

* **Response:**

{

"token": "JWT\_TOKEN",

"user": {

"id": "12345",

"name": "John Doe",

"email": "john@example.com"

}

}

**User APIs**

1. **GET** /api/user/profile

* **Description:** Fetches the authenticated user's profile.
* **Headers:**

{

"Authorization": "Bearer JWT\_TOKEN"

}

* **Response:**

{

"id": "12345",

"name": "John Doe",

"email": "john@example.com"

}

2. **PUT** /api/user/profile

* **Description:** Updates the authenticated user's profile.
* **Headers:**

{

"Authorization": "Bearer JWT\_TOKEN"

}

* **Request Body:**

{

"name": "John Smith",

"email": "johnsmith@example.com"

}

* Response:

{

"message": "Profile updated successfully",

"user": {

"id": "12345",

"name": "John Smith",

"email": "johnsmith@example.com"

}

}

**8.Authentication**

Authentication ensures that only registered users can access and manage their trips. In TripTrek, we use JWT (JSON Web Token) to handle this. Here’s a simpler breakdown of how authentication works:

**1. How Authentication Works**

**1. User Registration:**

- New users sign up by providing their name, email, and password.

- The password is securely encrypted (hashed) before it’s saved in the database to protect it.

- Once registered, the user receives a \*JWT token\* to confirm they’re logged in.

**2. User Login:**

- Existing users log in using their email and password.

- If the login is successful, a \*JWT token\* is generated and sent to the user.

- This token is proof that the user is authenticated and can be used for future actions like viewing trips.

**3. JWT Token:**

- The JWT token is like a key given to the user when they log in.

- It is sent in the headers of every request to access protected parts of the app (like managing trips).

**2. Registration Flow**

* **Endpoint:** POST /api/auth/register
* **What It Does**: Registers a new user by saving their information (name, email, and encrypted password) in the database.

**{**

**"name": "John Doe",**

**"email": "john@example.com",**

"password": "password123"

}

* **Response:**

The user gets a \*JWT token\* which they’ll use in the future to authenticate requests:

json

{

"token": "JWT\_TOKEN",

"user": {

"id": "12345",

"name": "John Doe",

"email": "john@example.com"

}

}

**3. Login Flow**

* Endpoint: POST /api/auth/login.
* What It Does: Authenticates the user with their email and password. If successful, a JWT token is generated.

{

"email": "john@example.com",

"password": "password123"

}

* **Response:**

The response includes the \*JWT token\* to prove the user is logged in:

{

"token": "JWT\_TOKEN",

"user": {

"id": "12345",

"name": "John Doe",

"email": "john@example.com"

}

}

**4. Protecting Routes with JWT Token**

* JWT Token is used to access protected parts of the app (like viewing trips or managing the user profile).
* Every request to these protected routes must include the token in the Authorization header.

To access a protected route (like fetching the user’s profile), the JWT token must be included in the request headers:

{

"Authorization": "Bearer JWT\_TOKEN"

}

* The server checks if the token is valid. If it is, the user is allowed to access the resource; if not, they get an error.

**5. Token Expiration**

* JWT tokens can expire after a set time (like 1 hour). After expiration, the user must log in again to get a new token.

**6. Storing JWT Tokens**

* On the frontend, the \*JWT token\* is stored (usually in \*localStorage\* or \*sessionStorage\*) so the user stays logged in even after refreshing the page.

**Storing the Token:**

javascript

localStorage.setItem('token', token);

Using the Token in Requests:

When making requests, the token is retrieved from localStorage and added to the request headers:

```javascript

const token = localStorage.getItem('token');

const config = {

headers: {

Authorization: Bearer ${token},

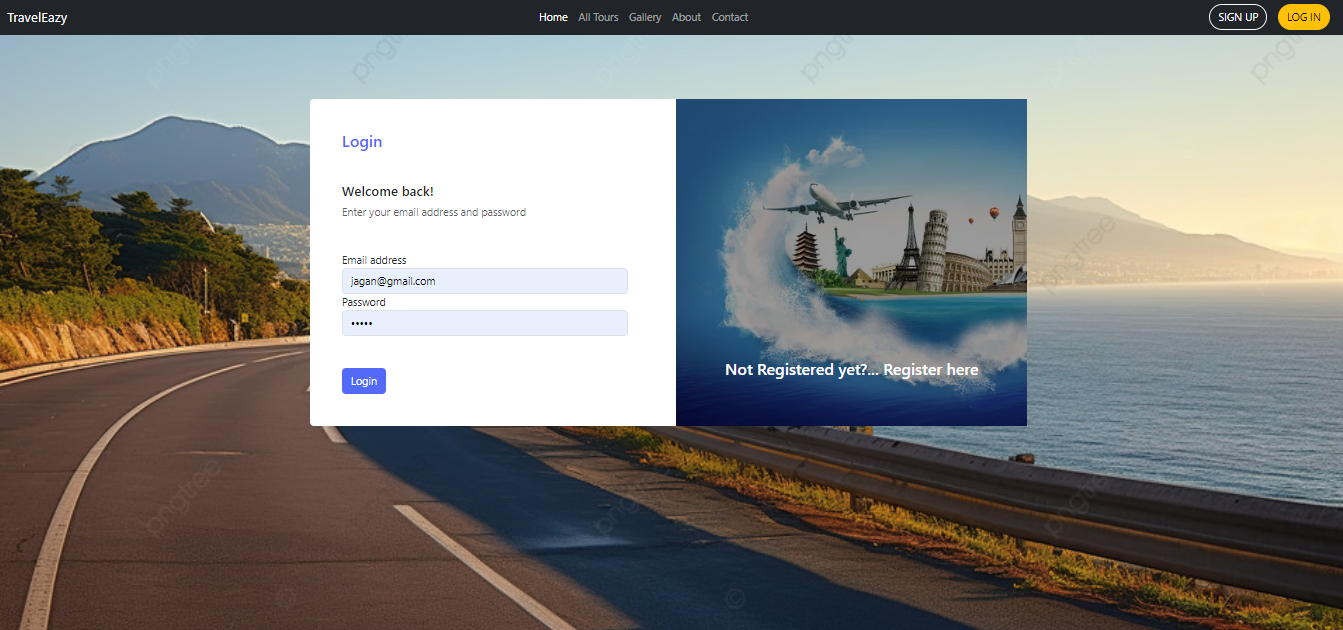
},

}

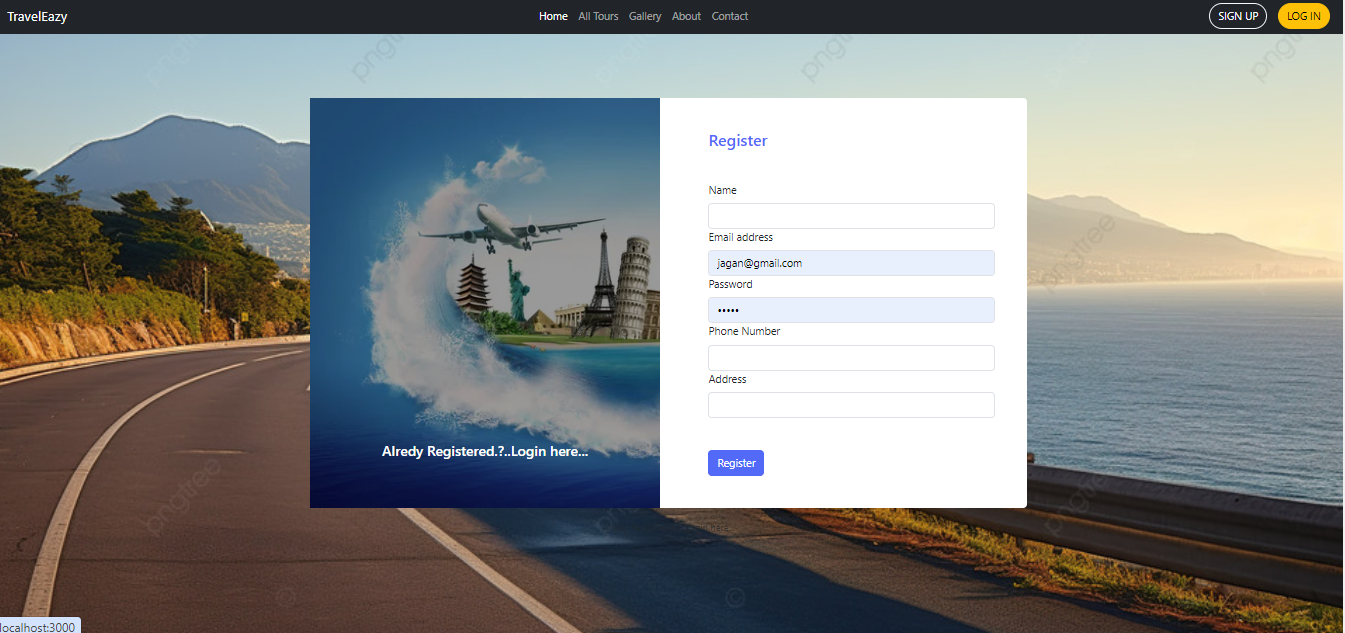
9.User Interface

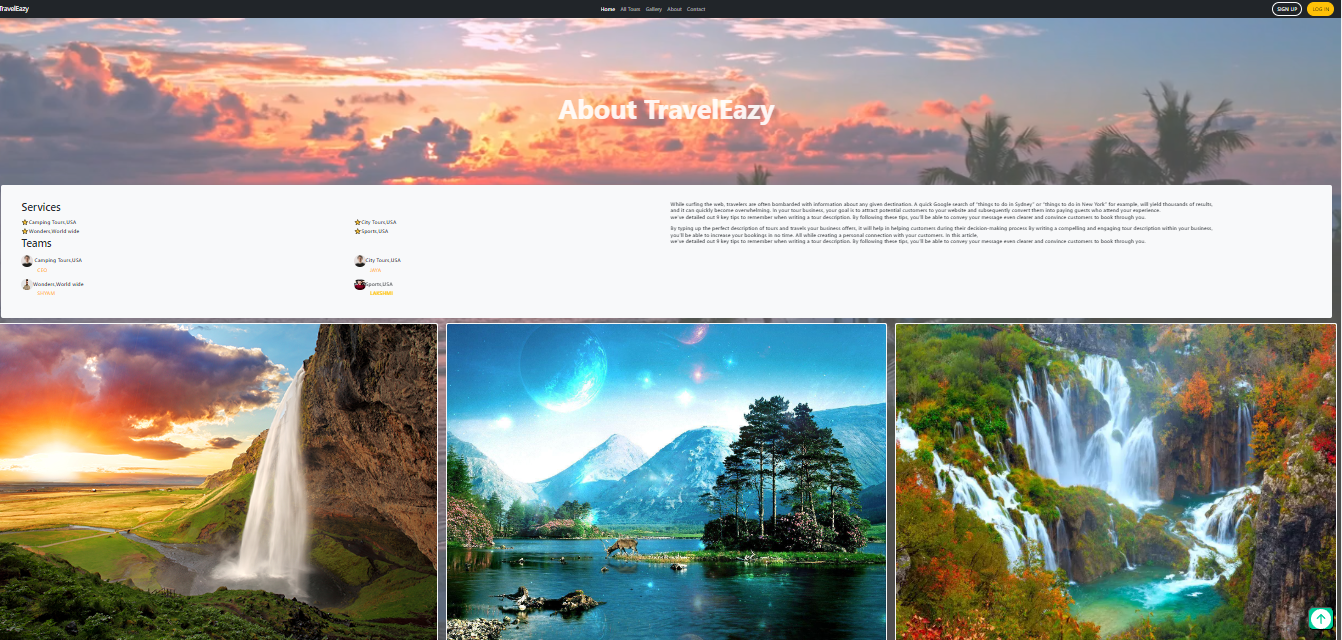
**HomPage:** 

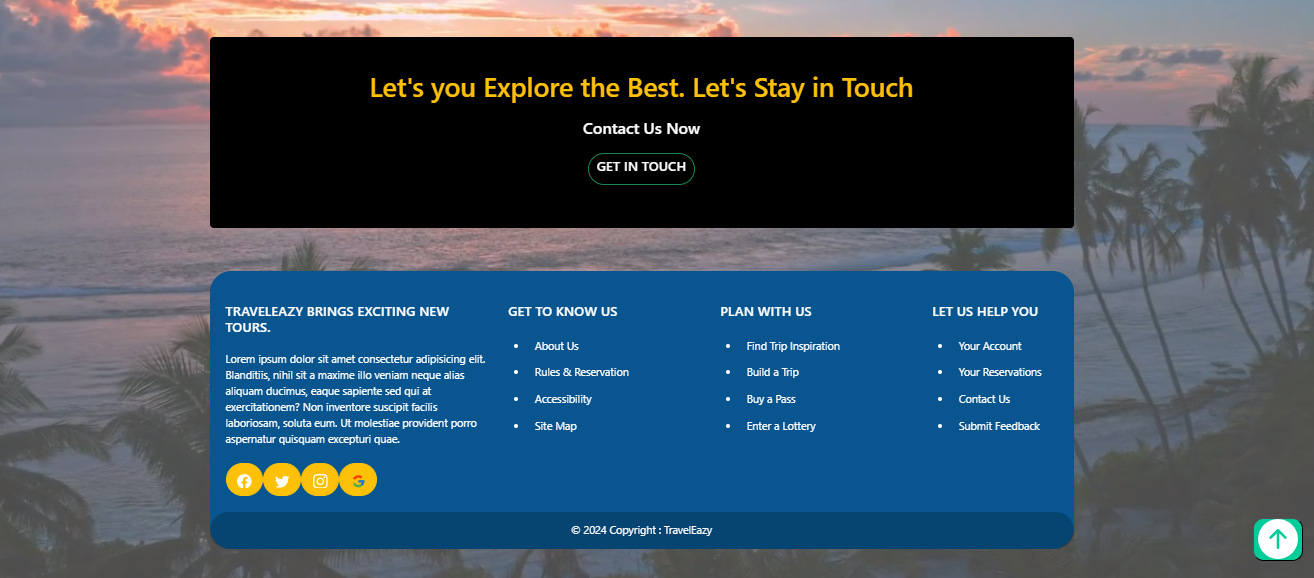
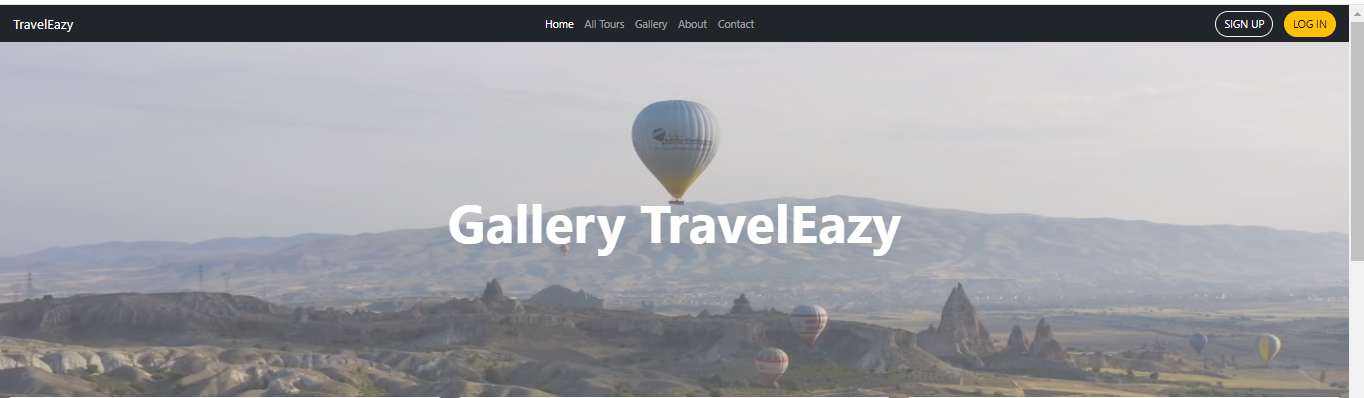
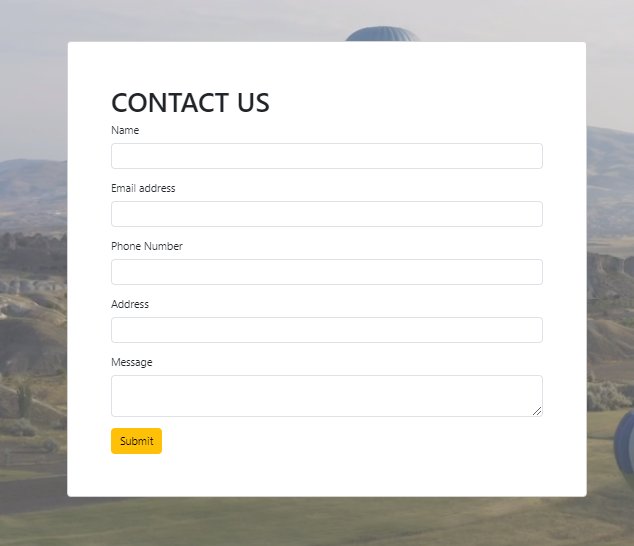
**Sign in**:



**Register:**



Gallery:

Footer gallery:contacexploreplaces

**10.Testing**

**System Testing**

* The purpose of testing is to discover errors. Testing is the process of trying to discover every conceivable fault or weakness in a work product. It provides a way to check the functionality of components, sub-assemblies, assemblies and/or a finished product It is the process of exercising software with the intent of ensuring that the
* Software system meets its requirements and user expectations and does not fail in an unacceptable manner. There are various types of test. Each test type addresses a specific testing requirement.

**Types of Tests**

**1.Unit testing**

* Unit testing involves the design of test cases that validate that the internal program logic is functioning properly, and that program inputs produce valid outputs. All decision branches and internal code flow should be validated. It is the testing of individual software units of the application .it is done after the completion of an individual unit before integration. This is a structural testing, that relies on knowledge of its construction and is invasive. Unit tests perform basic tests at component level and test a specific business process, application, and/or system configuration. Unit tests ensure that each unique path of a business process performs accurately to the documented specifications and contains clearly defined inputs and expected results.

**2.Integration testing**

* Integration tests are designed to test integrated software components to determine if they actually run as one program. Testing is event driven and is more concerned with the basic outcome of screens or fields. Integration tests demonstrate that although the components were individually satisfaction, as shown by successfully unit testing, the combination of components is correct and consistent. Integration testing is specifically aimed at exposing the problems that arise from the combination of components.
* Software integration testing is the incremental integration testing of two or more integrated software components on a single platform to produce failures caused by interface defects.
* The task of the integration test is to check that components or software applications, e.g. components in a software system or – one step up – software applications at the company level – interact without error.

**Test Results**: All the test cases mentioned above passed successfully. No defects encountered.

3.**Acceptance Testing**

* User Acceptance Testing is a critical phase of any project and requires significant participation by the end user. It also ensures that the system meets the functional requirements.

**Test Results:** All the test cases mentioned above passed successfully. No defects encountered.

**4.Functional testing**

* Functional tests provide systematic demonstrations that functions tested are available as specified by the business and technical requirements, system documentation, and user manuals.

Functional testing is centered on the following items:

**Valid** **Input : identified classes of valid input must be accepted.**

**Invalid Input : identified classes of invalid input must be rejected.**

**Functions : identified functions must be exercised.**

**Output : identified classes of application outputs must be exercised.**

**Systems/Procedures: interfacing systems or procedures must be invoked.**

* Organization and preparation of functional tests is focused on requirements, key functions, or special test cases. In addition, systematic coverage pertaining to identify Business process flows; data fields, predefined processes, and successive processes must be considered for testing. Before functional testing is complete, additional tests are identified and the effective value of current tests is determined.

**5.White Box Testing**

* White Box Testing is a testing in which in which the software tester has knowledge of the inner workings, structure and language of the software, or at least its purpose. It is purpose. It is used to test areas that cannot be reached from a black box level.

**6.Black Box Testing**

* Black Box Testing is testing the software without any knowledge of the inner workings, structure or language of the module being tested. Black box tests, as most other kinds of tests, must be written from a definitive source document, such as specification or requirements document, such as specification or requirements document. It is a testing in which the software under test is treated, as a black box .you cannot “see” into it. The test provides inputs and responds to outputs without considering how the software works.

**11. Known Issues**

In any project, some issues or bugs may still exist, and it's important to document them for developers and users. Below are the known issues in the current version of TripTrek:

**1. Token Expiry Handling**

* Issue: If the JWT token expires while a user is interacting with the app (e.g., creating or viewing trips), they aren't notified immediately. They may encounter failed requests without understanding why.
* Impact\*: Users may lose data or actions may not go through as expected.
* Workaround\*: Users have to log out manually and log back in to refresh the token.
* Solution\*: Implement an automatic token refresh system or a user-friendly notification prompting them to log in again when the token expires.

**2. Slow Page Load Times on Trip Details Page**

* Issue: The Trip Details page can load slowly when a user has a large number of trips or complex trip data.
* Impact: Users experience delays in viewing their trip details, leading to frustration.
* Workaround: Users with fewer trips experience no delays, but for those with many trips, the page can take extra time to render.
* Solution: Optimize data fetching and pagination so that only a limited number of trips are loaded at once, rather than all at once.

**3. No Offline Support**

* Issue: The application doesn’t currently support offline mode, meaning users cannot plan or view trips without an internet connection.
* Impact: Users cannot use the app when traveling to places with limited or no internet.
* Workaround: None.
* Solution: Implement service workers or caching mechanisms (such as Progressive Web App (PWA) features) to provide offline access to trips already viewed or saved.

**4. Inconsistent UI on Smaller Screens**

* Issue: Some pages (like the Trip Overview page) do not fully adapt to mobile screens or smaller devices, leading to a cluttered and difficult-to-navigate interface.
* Impact: Mobile and tablet users face challenges in navigating or using some features.

**5. Limited Search Functionality**

* Issue: The search feature for finding trips is very basic. Users can only search by trip title, but they cannot filter by date, location, or other trip details.
* Impact: It can be difficult for users to find a specific trip if they have many saved trips.
* Workaround: Users must scroll manually to find the desired trip.
* Solution: Enhance the search feature to include filters for trip date, destination, or type.

**6. Error Messages Not Always Clear**

* Issue: When there’s a server-side error (e.g., a database connection issue), users sometimes receive generic error messages like "Something went wrong," which doesn't help them understand what the problem is.
* Impact: Users can be confused and may not know how to resolve the issue or report it accurately.
* Workaround: Users must try reloading the page or contacting support.
* Solution: Provide more specific and helpful error messages to guide users on what went wrong and potential fixes.

**12.Future Enhancements**

In addition to resolving the known issues, the following enhancements are planned to improve user experience and add value to the TripTrek application:

**1. Real-Time Collaboration on Trip Planning**

* Enhancement: Allow users to invite friends or family to collaborate on trip planning in real time. For example, multiple people can add to or modify a shared trip itinerary.
* Benefit: This would make group trips easier to organize, with everyone able to contribute to the trip details.
* Implementation: Use WebSockets or a similar real-time technology to allow live collaboration and updates to trip plans.

**2. Push Notifications for Trip Updates:**

* Enhancement: Implement a notification system that alerts users about important updates, like approaching trip dates, new suggestions for trips, or changes to flight schedules.
* Benefit: Helps users stay informed and updated without having to log into the app frequently.
* Implementation: Use Push Notifications via browser or mobile devices for reminders, updates, and important alerts.

**3. Advanced Trip Search and Filtering**

* Enhancement: Improve the search and filtering functionality so that users can find trips by destination, date, type of activities, or other trip-specific details.
* Benefit\*: Makes it easier for users to navigate through large amounts of trip data and find specific trips quickly.
* Implementation: Add a robust search bar with filtering options using \*\*MongoDB query\* capabilities and UI refinements.

**4. Trip Recommendations and AI-Powered Suggestions**

* Enhancement: Integrate AI-based trip recommendations based on user preferences and past trips. For example, if a user often visits beach destinations, the app could suggest similar trips.
* Benefit\*: Offers a personalized experience, making trip planning more efficient and enjoyable for users.
* Implementation\*: Use machine learning models to analyze user data and suggest personalized trip options based on travel history and preferences.

**5. Offline Mode Support\***

- \*Enhancement\*: Add offline support for trip planning, so users can create or view their trips even without an internet connection.

- \*Benefit\*: Makes the app more reliable for travelers who may be in areas with poor or no internet connectivity.

- \*Implementation: Implement \*\*Progressive Web App (PWA)\* features and caching strategies to store critical data offline.

**6. Social Media Integration\***

* Enhancement\*: Add social media sharing options that allow users to share their trips or itineraries directly to platforms like Facebook, Instagram, and Twitter.

- \*Benefit\*: Makes it easy for users to share trip details with friends and family, potentially growing the app’s user base.

- \*Implementation\*: Use APIs from popular social media platforms to integrate sharing features into the app.

---

#### \*7. Multiple Language Support\*

- \*Enhancement\*: Add support for multiple languages to make the app accessible to users worldwide.

- \*Benefit\*: Expands the user base and makes the app usable for non-English speaking users.

- \*Implementation: Use \*\*i18n (internationalization)\* libraries to translate the app’s content and allow users to switch between languages.

---

#### \*8. Trip Budget Tracking and Expense Sharing\*

- \*Enhancement\*: Add a budgeting feature that lets users track their trip expenses and share costs with other travelers. Users can set a budget for their trip and record expenses as they go.

- \*Benefit\*: Makes trip planning more comprehensive by including financial tracking, which is crucial for many travelers.

- \*Implementation\*: Implement a trip expense tracker in the UI and allow users to export or share the details with fellow travelers.

---

#### \*9. Integration with Flight and Hotel Booking APIs\*

- \*Enhancement\*: Allow users to book flights, hotels, or other travel arrangements directly from the app using third-party booking services like Skyscanner or Booking.com.

- \*Benefit\*: Provides a seamless experience where users can plan and book their entire trip within the app, rather than needing separate apps or websites.

- \*Implementation\*: Integrate with APIs from popular travel booking platforms to fetch and display flight/hotel data in the app.