© January 2026 | IJIRT | Volume 12 Issue 8 | ISSN: 2349-6002

Wanderlust Travel Planner

Mohit Kumar Choudhary[1], Yogesh Singh Rajawat[2], Shreya Agarwal[3], Charul Chhipa[4]
Ronak Sharma[5]
[1,2,4,5]*Department of Computer Science and Engineering JECRC University, Jaipur, India*
[3]*Assistant Professor–II, Department of Computer Science and Engineering JECRC University, Jaipur, India*

***Abstract*—Planning a trip is often difficult because people need to search for places, collect information from different websites, and manually create a day-wise plan. To solve this common problem, we built Wanderlust, a MERN-based travel planner web application. Wanderlust provides users a single platform where they can explore tourist destinations, read descriptions, check images, save their favorite places, and create a complete itinerary for their trip. The design of the application is simple and easy to understand, making it useful even for first-time travelers or students who have little experience in planning trips.**

The Wanderlust system is built using the MERN stack: MongoDB for storing user and destination data, Express.js and Node.js for backend APIs, and React.js for the frontend interface. Admin users can add new destinations, edit existing entries, or remove outdated information. This makes the system dynamic and expandable.

This research paper explains the complete development life cycle of Wanderlust, including requirement analysis, architecture design, database modeling, frontend and backend development, testing, results, and future improvements. Testing was performed manually as well as through Postman, and all major features worked successfully. Wanderlust can be further improved by integrating maps, AI-based recommendations, and budget estimation. Overall, the system provides a smooth and user-friendly experience, helping users plan their trips in a more organized way.

***Index Terms*—Travel Planning, MERN Stack, Itinerary Man- agreement, Full-Stack Web Application, Tourism Technology**

I. INTRODUCTION

Travelling is an enjoyable experience, but planning a trip can be stressful for many people. A traveler needs to find good places to visit, search for important details, save useful information, and then plan activities for each day. This involves visiting multiple websites, noting down information manually, and managing everything separately. Because of this, the planning phase becomes time-consuming and confusing.

To solve this problem, we created Wanderlust — a travel planner web application that makes the entire trip planning

process simple and organized. Wanderlust helps users explore destinations, view descriptions, check images, and prepare a day-wise itinerary using a clean and easy-to-use interface. The objective is to reduce the effort required in planning and provide all necessary features in one place.

Wanderlust uses the MERN stack, which includes MongoDB, Express.js, React.js, and Node.js. MERN is widely used because it supports full-stack JavaScript development, making the system fast and efficient. The application includes both user and admin panels. While normal users can plan their trips and explore places, the admin can add or update destination data.

The main goals of Wanderlust are:

- Provide a simple platform for exploring travel destinations.
- Help users plan trips using a day-wise itinerary.
- Offer a user-friendly interface for all age groups.
- Ensure smooth performance using the MERN stack.
- Allow administrators to manage travel data easily. Overall, Wanderlust aims to make trip planning more enjoyable by reducing the stress and time involved in the planning process.

II. SYSTEM ARCHITECTURE

The system architecture of Wanderlust is based on the MERN stack, which connects the frontend and backend efficiently. Each technology in the MERN stack has a specific role:

A. MongoDB

MongoDB is used to store all data, including user details, destination data, and itinerary plans. It stores data in JSON format, which makes it easy to update and retrieve.

Collections include:

- Users: name, email, password
- Destinations: title, image, description, cost, location
- Itineraries: user ID, day-wise plan
- Admin: admin login data

B. Express.js

Express.js handles all backend routes. It receives requests from the frontend, processes them, and sends responses in JSON format.

C. React.js

React.js is used for the user interface. It updates only the required parts of a page, reducing loading time and improving user experience.

D. Node.js

Node.js runs the server and connects the frontend with the database. It ensures fast processing of user requests.

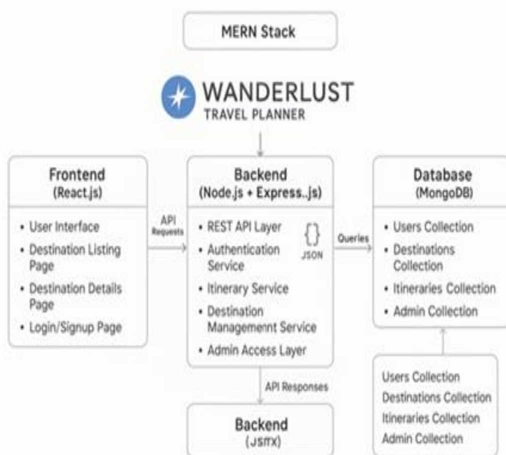

Fig. 1. Overall System Architecture of Wanderlust

III. METHODOLOGY

The development of Wanderlust followed a structured, step- by-step methodology to ensure that the system was not only functional but also simple, user-friendly, and efficient. Each stage of development was planned carefully so that the final product could meet real user needs. The methodology explains how requirements were collected, how the system was designed, how the user interface was planned, and how the backend and database were developed. It also explains the testing approach used to verify the final system. This section describes each part of the methodology in detail.

A. Requirement Analysis

Requirement analysis was the first and most important step because it helped us understand what problems users face while planning trips. We interacted with students, frequent travelers, and people who often face confusion during travel planning. These interactions were informal interviews and simple surveys that asked users about their typical planning steps, the tools they currently use, and the problems they encounter. We noted down common pain points such as scattered information, difficulty in creating a day-wise plan, and the need for a single platform to store and edit plans.

After collecting ideas and challenges, we converted them into clear functional and non-functional requirements. Functional requirements included: destination browsing, viewing details (images, description, cost), user registration and login, itinerary creation and editing, bookmarking favorites, and admin controls to add or update destinations. Non-functional requirements included: fast loading times, mobile responsive- ness, ease of use, and secure handling of user credentials. These requirements were prioritized based on user feedback; for example, the itinerary builder and simple UI were given high priority because users repeatedly mentioned them as essential.

B. System Design

In this stage, we created diagrams to visualize how the system would work internally. These diagrams acted as blueprints for developers and helped avoid confusion later. The main design artifacts were: use case diagrams, entity-relationship diagrams (ERD),

wireframes, and simple data flow diagrams.

1) Use Case Diagram: We prepared a use case diagram for both user and admin roles. For regular users, the primary use cases included: browse destinations, view details, save favorite, create itinerary, edit itinerary, and delete itinerary. For the admin user, the use cases included: add destination, edit destination, remove destination, and view user activity (basic logs). The use case diagram ensured that all interactions were captured and provided clarity on user roles.

2) Entity-Relationship Diagram (ERD): An ERD was created to model the database structure. The three main entities were Users, Destinations, and Itineraries. The relationships were straightforward: one user can have many itineraries, and each itinerary may contain multiple destination references. Destinations stand alone as records containing fields like title, images, description, and tags. ERD helped design MongoDB schemas in a structured manner without being constrained by rigid table schemas.

3) Wireframes and UI Mockups: Wireframes were created for key screens: homepage, search/listing page, destination detail page, itinerary builder page, and admin dashboard. Wireframes focused on layout decisions such as where to show images, how to place action buttons (save, add to itinerary), and how to present day-wise planning in a clear way. Mobile- first wireframes were prioritized to ensure good usability on phones since many users plan trips on mobile devices.

C. Frontend Development

Frontend development was done using React.js because it allows building component-based user interfaces that update efficiently. We divided the frontend into reusable components such as Header, Footer, Destination Card, Destination Detail, Itinerary Editor, and Login Form. Using components reduced duplication and made it easier to maintain the UI. State management was handled using Reacts built-in hooks; for
shared state like the current user or a temporary itinerary, we used React Context to avoid prop-drilling.

The Itinerary Editor component allows users to add days, attach destinations, add short notes for each day, and save or export the itinerary. Form validation was implemented to ensure mandatory fields are filled. Accessibility and responsiveness were checked using simple CSS breakpoints and testing on small and medium screen sizes.

D. Backend Development

The backend was implemented with Node.js and Express.js. The main responsibilities of the server were: user authentication, destination CRUD (Create, Read, Update, Delete) operations, itinerary CRUD operations, and serving static files in production. The backend used RESTful API design with JSON payloads for easy consumption by the React frontend. Important design choices included:

- Authentication: We used JWT (JSON Web Tokens) for stateless authentication. During login, the server issues a token signed with a secret that the frontend stores in local Storage (or in-memory for safety). Protected routes check the token on each request.
- Input Validation: Both server-side and client-side validation were implemented. Server-side validation used middleware to check required fields and proper formats to avoid bad data entering the database.
- Error Handling: Express middleware was used for centralized error handling that returns consistent JSON error responses with HTTP status codes.

APIs included:

- `POST /api/auth/signup` — register a new user (hashed password).
- `POST /api/auth/login` — authenticate and return JWT.
- `GET /api/destinations` — fetch destination list with optional filters.
- `GET /api/destinations/:id` — fetch details of a specific destination.
- `POST /api/itineraries` — create or save an itinerary.
- `GET /api/itineraries/: userid` — fetch all itineraries for a user.
- `PUT /api/destinations/:id` — admin updates a destination.
- `DELETE /api/destinations/:id` — admin re- moves a destination.

E. Database Structure

MongoDB was selected because it stores data in a flexible JSON-like format, which fits well for

varying destination content (different destinations may have different numbers of images, tags, or fields). We designed three main schemas using Mongoose (a MongoDB ODM for Node.js):

1) User Schema: Fields:
- name: `String`
- `email: String (unique)`
- `password: String (hashed)`
- `created At: Date`

Passwords are hashed using bcrypt before saving. The server never stores plaintext passwords.

2) Destination Schema: Fields:
- `title: String`
- `description: String`
- `images: [String] (URLs)`
- `location: String`
- `tags: [String]`
- `estimated Cost: Object (min, max)`
- `created At: Date`

Images are saved as URLs (hosted on a static server or cloud storage). Tags help in filtering and search.

3) Itinerary Schema: Fields:
- `userId: ObjectId (reference to Users)`
- `title: String`
- `days: [dayNumber: Number, activities: [time, title, notes, destination]]`
- `created At: Date`

The itinerary structure supports multiple days, each with many activities. Activities can reference a destination by its ID, allowing the app to display rich details for each activity.

F. Workflow

The workflow explains how the entire system works from a user's perspective and how data flows through the system components:

1) The user opens the website; React renders the homepage and optionally fetches featured destinations using `GET /api/destinations`.
2) The user searches or browses destinations; the frontend calls the API with optional query filters (location, tag, cost).
3) When the user clicks on a destination, the frontend fetches full details via GET `/api/destinations/:id` and displays images, description, and suggestions.
4) To create an itinerary, the user signs in. The login flow calls `POST /api/auth/login`, receives a JWT, and stores it for subsequent requests.
5) The Itinerary Editor component allows adding days and activities; on save the frontend posts the itinerary to `POST /api/itineraries` with the JWT for authentication.
6) The server validates the JWT, checks the payload, saves the itinerary in MongoDB, and returns the saved itinerary ID.
7) Users can later fetch or update itineraries using `GET /api/itineraries/:userid` or `PUT /api/itineraries/:id`.
8) Admin users access the admin dashboard, perform destination CRUD operations that call the protected end- points; these endpoints are validated to ensure the user has admin rights.

This workflow ensures a clean separation between frontend actions and backend processing while keeping user data se- cure.

IV. ALGORITHMS AND WORKFLOWS

A. User Login Algorithm

A secure and reliable login process is essential. The steps are:

1) The frontend collects email and password from the login form.
2) The frontend sends a `POST /api/auth/login` re- quest with the credentials.
3) On the server, the backend searches the Users collection for the email.
4) If a user is found, the server compares the provided password with the stored hashed password using bcrypt's compare function.
5) If the password matches, the server generates a JWT token containing the user ID and role, signed with the server's secret key, and returns it to the client.
6) The frontend stores the token (preferably in memory or HttpOnly cookie) and uses it for authenticated API calls.
7) If the password does not match or the user is not found, the server returns a 401 Unauthorized error with a helpful message.

B. Itinerary Creation Algorithm

Itinerary creation focuses on flexibility and easy editing:

1) The user starts an itinerary and adds a title and the number of days.
2) For each day, the user adds activities. Each activity contains a time (optional), a short title, notes, and an optional reference to a destination.
3) When saving, the frontend composes a JSON Object representing the itinerary and sends it to `POST /api/itineraries`.
4) The server validates the structure, ensures the user is authenticated, and stores the itinerary in MongoDB.
5) The server returns the saved itinerary ID so the frontend can link it to the user's profile and show a success message.
6) Users can edit or delete itineraries using the appropriate API endpoints; the server checks ownership before allowing modifications.

V. Testing

A. Manual Testing

Manual testing was performed to ensure the user interface and flows work as expected. The following test scenarios were executed:

- User Signup and Login: Create accounts with valid and invalid data; verify error messages and successful login flows.
- Destination Browsing: Search and filter destinations; check that images and details render correctly.
- Itinerary Editing: Create, update, and delete itineraries; confirm changes persist in the database.
- Responsiveness: Test UI layout across desktop and mobile screen sizes to ensure usability.
- Admin Actions: Add, update, and remove destinations from the admin dashboard; verify immediate effect on public listing.

Bugs found during manual testing (for example, missing form validations or wrong button states) were tracked and fixed. Each fix was retested to ensure no regression occurred.

B. API Testing

Postman was used to test all REST endpoints. The testing approach included:

- Sending requests with valid and invalid payloads to verify server-side validation.
- Testing protected routes with and without JWT tokens to enforce authentication checks.
- Verifying correct HTTP status codes for success and error cases.
- Inspecting responses to ensure required fields are returned (for example, itinerary IDs and destination objects).

Example API tests:

- `POST /api/auth/login` with incorrect password should return 401 Unauthorized.
- `GET /api/destinations? Tag=beach` should re- turn a filtered list.
- `POST /api/itineraries` with missing day entries should return 400 Bad Request.

C. Performance and Load Considerations

While the project is primarily a prototype, we considered basic performance improvements:

- Pagination for destination lists to avoid large payloads.
- Indexing commonly searched fields in MongoDB (like location and tags).
- Minifying frontend assets for faster load times in production.

VI. RESULTS

After development and testing, the Wanderlust application provides a smooth and usable experience. The main outcomes include:

- A clear user interface that allows easy browsing of destinations and quick creation of itineraries.
- A secure authentication module that uses hashed pass- words and JWT.
- A flexible itinerary structure stored in MongoDB that supports multiple days and activities.
- Admin controls to manage destination data without modifying code.
- Fast API responses for typical user actions in the development environment.

User feedback (from informal testing with peers) high- lighted that the itinerary editor and the simple save/load features are especially useful. Some users suggested adding maps and suggested timing for activities, which we have listed in future work.

VII. LIMITATIONS

Although Wanderlust covers the core features of travel planning, several limitations remain:

- No map integration: The app currently does not show map routes or location pins.
- No AI recommendations: There is no machine learning module to suggest places or plan optimized itineraries.
- No booking features: Wanderlust does not support direct hotel or flight bookings.
- Basic admin analytics: Admin panel does not include detailed analytics or bulk upload for destinations.
- Prototype-level deployment: The app is tested in development or staging; full production hardening (rate limiting, advanced logging, backups) requires additional work.

VIII. FUTURE ENHANCEMENTS

Future work will focus on features that increase usability and automation:

- Map Integration: Add Google Maps or OpenStreetMap to show routes and distance estimates.
- AI-based Suggestions: Use simple recommender systems based on user history to suggest destinations and activities.
- Budget Planner: Provide budget estimates per day and overall trip cost calculations.
- Booking Integrations: Integrate with hotel or flight APIs to offer booking options.
- Mobile App: Build a native or cross-platform app (React Native) for offline access to saved itineraries.

IX. CONCLUSION

Wanderlust provides an integrated and user-friendly environment for travel planning. By combining destination browsing detailed information, and an easy-to-use itinerary editor, the system removes the need to search and save data across multiple platforms. The MERN stack allowed us to build a full-stack JavaScript application that is fast, maintainable, and scalable. Testing confirmed functional correctness for core features. With further enhancements like maps, AI recommendations, and booking support, Wanderlust can evolve into a full travel assistant.

REFERENCES

[1] MERN Stack Documentation. https://www.mongodb.com/mern-stack

[2] React.js Official Documentation. https://reactjs.org/docs/getting- started.html

[3] Node.js and Express.js Documentation. https://nodejs.org/ and https://expressjs.com/

[4] MongoDB Official Documentation. https://www.mongodb.com/docs

[5] Open-source travel applications and tourism technology reports (2022– 2024). https://github.com/topics/travel-app