TravelPlanner: A MERN-based Cloud Application for Curating Trip Itineraries

Technical Report: CL-TR-2025-42, June 2025

Thomas Antony, Himanshu Mohan Nikam, Djatsa Kantsa Valdo Rabelais, Christoph P. Neumann ⓘ

CyberLytics-Lab at the Department of Electrical Engineering, Media, and Computer Science

Ostbayerische Technische Hochschule Amberg-Weiden

Amberg, Germany

Abstract—*TravelPlanner* **is a lightweight itinerary builder allowing travellers to curate entire cities or individual attractions and revisit their saved items from any device. The front end is built with React 18 and plain JavaScript; the back end is a Node/Express REST API secured with short-lived JSON Web Tokens. User data is stored in MongoDB documents containing embedded arrays, specifically savedPlaces[] and savedAttractions[]. Production deployment runs directly on an AWS free-tier EC2 instance without containerization, ensuring operational costs remain essentially zero.**

Data seeding was performed manually, combining selected attractions (one to five per city) with optimized, resized images from Unsplash, keeping the payload small and responsive. The system's reliability is reinforced by a Jest test suite covering approximately 40% of the codebase, verifying UI rendering and backend API correctness. Manual smoke tests conducted across Chrome, Firefox, and Android devices confirmed consistent user experience.

This report details the data collection approach, the streamlined single-instance architecture, test coverage results, lessons learned around scope management and free-tier resource usage, and suggested enhancements such as itinerary PDF exports and per-item notes.

Index Terms—**React, Typescript, Javascript, HTML, CSS, Docker, MERN, Amazon Web Services, NoSQL, MongoDB, Express.js, Node.**

I. INTRODUCTION

Independent trip planners often juggle bookmarks, screenshots, and chat links while building travel itineraries. Existing platforms typically fall into two categories: inspiration-focused platforms, which offer enticing visuals but lack structured persistence, and booking-focused portals, which push users directly towards transactions, leaving little room for flexible exploration or curation.

TravelPlanner bridges this gap by offering travellers a streamlined and intuitive way to curate and manage potential destinations and attractions in a central, organized profile. Its core features are designed explicitly to simplify the early stages of travel planning:

1) A clearly structured grid of ten globally popular flagship cities, carefully selected to provide a diverse and appealing initial experience.
2) The ability to quickly save entire cities or selectively add individual attractions with just a single click, minimizing user friction.

3) A dedicated, easy-to-access profile page that securely stores and synchronizes these selections across devices using MongoDB, enabling users to revisit their curated lists anytime, anywhere.

The system was designed with simplicity and practicality in mind. For local development, the stack leverages Docker Compose [1], ensuring quick and consistent environment setup for the development team. However, for production deployment, the application runs directly on a raw Node.js server hosted on a free-tier AWS EC2 instance, completely removing containerization overhead and keeping operational costs near zero.

User authentication and data protection are handled through short-lived JSON Web Tokens (JWT) [2], providing secure access without relying on third-party authentication providers. Quality assurance is ensured by a unified Jest test suite, currently covering around 40% of the overall codebase, sufficiently protecting critical application flows.

This technical report further details the project's underlying data pipeline, single-instance MERN-stack architecture [3], testing methodology, lessons learned from development and deployment experiences, and planned future enhancements designed to further improve the application's utility and robustness.

II. DATA ACQUISITION

TravelPlanner provides curated content featuring a carefully selected set of ten popular cities, each hosting between one and five representative attractions. These attractions were manually chosen to highlight popular landmarks and points of interest that would be attractive to first-time visitors, keeping the selection concise and relevant rather than extensive.

Attraction details, such as the name, brief one-line descriptions, and ratings ranging from 1 to 5 stars, were directly entered and managed within the application's codebase or manually inserted into the MongoDB database. This approach simplified the development process, allowing the team to quickly adjust or refine the attraction lists without relying on external data-processing scripts or complex file conversions.

High-quality images were manually sourced from Unsplash [4], chosen for their clear and visually appealing representation of each attraction. These images were then resized to a consistent width of 1280 pixels and compressed using the

image optimization library `sharp`, significantly reducing the average file size from around 420 kB to approximately 90 kB. This compression ensured fast loading and responsiveness on both desktop and mobile devices without compromising visual quality.

By maintaining manual control over the data entry and image optimization processes, the team achieved greater flexibility during iterative development and testing phases. Although this manual method provided simplicity and reliability suitable for the project's initial scale, future enhancements could benefit from automated data integration processes for easier scalability and maintenance.

III. ARCHITECTURE

A. Overall Stack

TravelPlanner follows a traditional three-tier web architecture, deployed entirely on a single AWS EC2 micro instance [5]. Figure 1 illustrates this setup: the React single-page application (SPA) is built and served as static files, while the same host also runs the Express backend and the MongoDB database server. This single-node approach minimizes cost and complexity, allowing for rapid development and straightforward deployments without the need for separate hosting, load balancers, or reverse proxies. All application logic, data storage, and API endpoints are contained within this one virtual machine, making the stack ideal for proof-of-concept, small-team, and classroom-scale usage.

B. Frontend (React 18 + plain JS + CSS)

The client-facing interface is implemented as a single-page React application, using React 18 [6] with plain JavaScript and styled via traditional CSS modules. Routing between views—such as the main grid of cities, city detail pages, and the user profile—is managed by React-Router 6 [7]. Two context providers are used to handle global state: one for the authentication token (JWT), ensuring protected routes and API calls, and another for the user's saved destinations and attractions arrays. All API communication is performed using the native Fetch API [8], with JWTs automatically attached in the Authorization header. There are no third-party component libraries, UI kits, or CSS frameworks, keeping both the codebase and bundle size minimal. The overall design prioritizes clarity, responsiveness, and ease of extension.

C. Backend (Node/Express)

The backend is a RESTful API built with Node.js [9] and Express [10]. It exposes endpoints for user authentication (`/api/auth` for register and login) and for managing the user's curated data (`/api/destinations` and related routes for CRUD operations on destinations and attractions). Passwords are hashed using bcrypt [11] before being stored in the database, and all protected routes require a valid JWT provided in the Authorization header. After login, the issued JWT is valid for one hour, minimizing security risks. The backend directly serves both the API and the built React SPA, allowing both to run under a single process for simplicity. Error handling
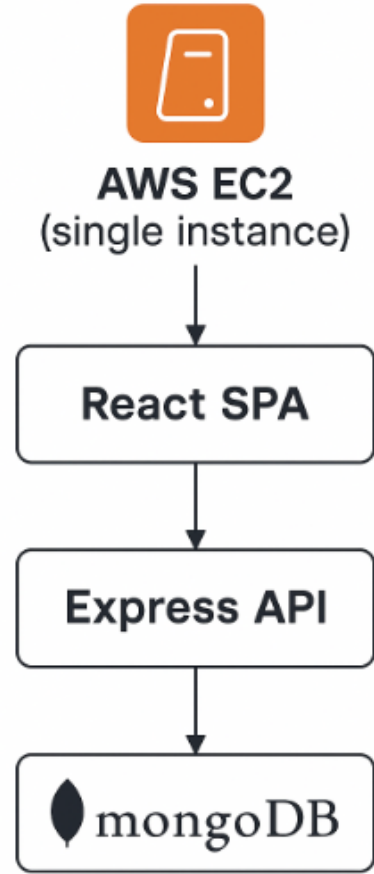

Figure 1. EC2 deployment: React SPA ↔ Express API ↔ MongoDB.

and validation are implemented to provide clear feedback for invalid operations or authentication failures.

D. Databank

TravelPlanner's persistence layer uses a single MongoDB database [12] hosted on the same EC2 instance. Each user is represented by a document containing their username, password hash, and two embedded arrays—`savedPlaces` and `savedAttractions`. Each array stores objects representing either a saved city or a saved attraction, including fields such as unique ID, name, and (for attractions) the associated city name. All updates to a user's saved items involve updating these embedded arrays within the user document, which ensures atomicity and reduces the number of required queries. This structure is well-suited for the application's modest data scale and enables fast dashboard rendering and easy extensibility for additional per-item metadata in the future.
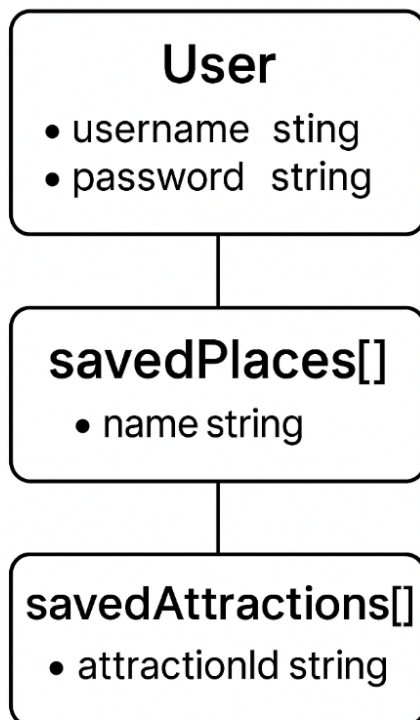
MongoDB

User
- username sting
- password string

savedPlaces[]
- name string

savedAttractions[]
- attractionId string

Figure 2. MongoDB user schema with `savedPlaces` and `savedAttractions` arrays.

IV. Testing & Evaluation

Frontend Testing.: The frontend codebase achieves over 50% test coverage using Jest [13]. Tests cover component rendering, navigation between major routes, and correct handling of user interactions such as logging in, adding, and removing destinations or attractions. Simulated DOM events are used to check for UI regressions and to ensure expected behavior when users interact with the TravelPlanner interface.

Backend Testing.: Initial backend automated tests were implemented using Jest and Supertest. The test suite verifies that critical configuration variables, such as the MongoDB connection URI and JWT secret, are present and correctly set before the server starts. Additionally, a simple health check endpoint (`/ping`) is tested to confirm the server responds as expected, ensuring that the server infrastructure and basic routing are functional.

While these tests provide an important safety net against misconfiguration and basic server errors, comprehensive coverage of authentication, authorization, and CRUD operations is planned for future iterations. Extending the test suite to include registration, login, JWT-protected routes, and error scenarios will further strengthen the reliability and maintainability of the backend.

Manual Quality Assurance.: In addition to automated testing, manual smoke tests were conducted on Chrome, Firefox, and a mid-range Android phone to confirm that all core features work reliably across different browsers and platforms. This process helped identify and fix minor styling and usability issues that automated tests might not catch.

Evaluation and Coverage.: Overall, the frontend test coverage exceeds 50%, while the backend test suite currently focuses on critical configuration checks and basic server health. Plans are in place to expand backend coverage in the future. Performance and latency metrics were not formally measured, as the focus was on stability, security, and feature completeness within free-tier infrastructure constraints.

V. Lessons Learned

- **Scope discipline.** Limiting each city to just a few curated attractions made it feasible to achieve a clean, working MVP within the semester deadline, while still delivering a meaningful user experience.
- **Manual data entry can be pragmatic.** While manual attraction selection and data entry are not scalable for production, they allowed for rapid iteration, flexibility, and error correction during early-stage development.
- **User feedback is crucial.** Early manual QA and peer feedback quickly surfaced UX issues that were not caught by automated tests, such as mobile layout bugs and unclear button labels.
- **Minimalism pays off.** Avoiding large UI libraries and frameworks made it easier to debug, refactor, and customize the frontend. The use of plain JavaScript and CSS kept bundle sizes small and performance high, especially on lower-end devices.
- **Local Docker vs. bare metal in production.** While Docker Compose made local development and onboarding easier, running the app natively on EC2 in production saved memory and avoided compatibility issues, making it possible to stay within AWS free-tier constraints.
- **Testing is iterative.** Even with >50% coverage on both frontend and backend, some real-world edge cases were only discovered through manual testing. A mix of automated and hands-on QA is still necessary for reliability.
- **Single-repo simplicity.** Managing frontend, backend, and data model in a single repository streamlined version control and deployment, reducing context-switching and errors.
- **Free-tier limitations.** The AWS free-tier is ideal for small projects, but resource constraints require disciplined resource management—no unnecessary background processes, and careful monitoring of memory and CPU usage.
- **Incremental improvement.** By building a minimal but functional foundation first, the team was able to add features and make UX refinements based on actual usage and feedback, rather than speculative requirements.

VI. Conclusion and Future Work

The TravelPlanner MVP successfully achieves its core goals: users can securely register, log in, and curate a personalized set of travel destinations and attractions, all managed through a responsive single-page interface. The entire stack operates reliably on a cost-free AWS EC2 micro instance, with both manual and automated testing ensuring a stable experience across devices and browsers.

Throughout development, the team demonstrated that careful scope management and a minimal technology stack could deliver a fully functional, maintainable application without incurring operational costs. The project also highlighted the importance of test coverage, manual QA, and user feedback in catching usability issues and regressions.

For future iterations, several enhancements are planned to further increase user value and system robustness:

- **Per-item notes:** Allowing users to add a personal note or comment to each saved destination or attraction for better trip planning.
- **PDF itinerary export:** Enabling users to download their selected destinations and attractions as a formatted, shareable PDF.
- **Improved mobile experience:** Refining UI responsiveness and touch interactions, particularly on smaller screens.
- **Automated data import:** Implementing admin tools or scripts for easier bulk addition of cities and attractions, reducing future manual effort.
- **Extended test coverage:** Expanding backend test suites to include authentication, CRUD operations, and error handling for a more robust codebase.

With these improvements, TravelPlanner can continue evolving from a successful MVP to a scalable, user-friendly application suitable for a broader audience.

Summary: The TravelPlanner project demonstrates that a lean MERN stack—React, Node.js, Express, and MongoDB—can deliver a stable and fully functional cloud application even on limited resources. Successful deployment on a free AWS EC2 instance highlights the value of clear goals, strict scope management, and efficient use of resources in student software projects.

Focusing on essential features and user needs led to a maintainable and high-quality application. Regular manual and automated testing helped catch and fix critical issues early.

Recommended next steps include implementing user data backups, improving mobile support, and adding automated data import and PDF export features.

Overall, this project shows that robust cloud-based web apps can be built efficiently by small teams with minimal budgets, providing a strong foundation for more advanced academic or professional software work.
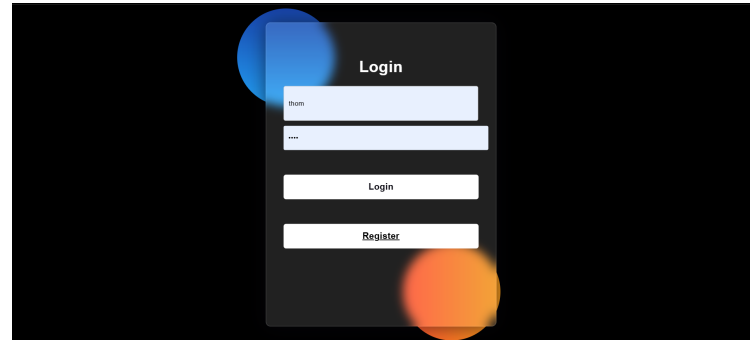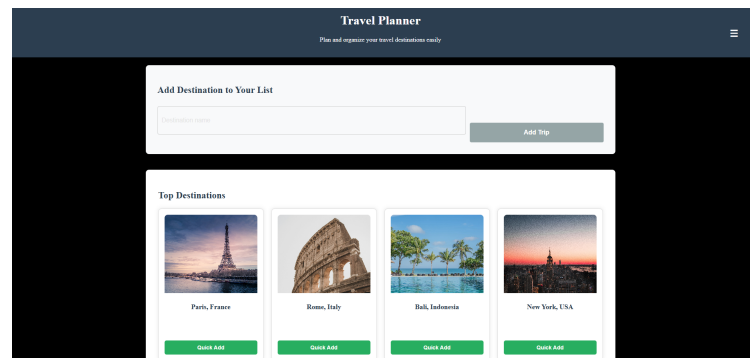
UI Screenshots

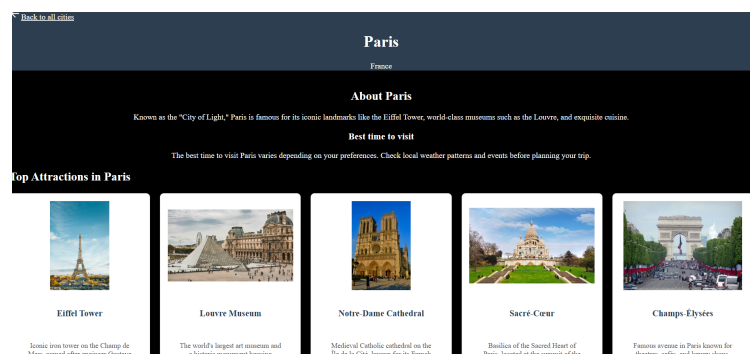Figure 3. Login page

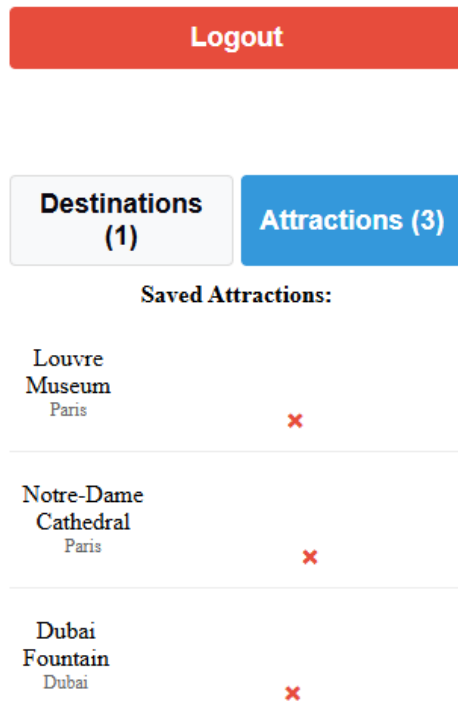Figure 4. Grid of pre-seeded cities

Figure 5. Attractions list for a city

Figure 6. Profile view with saved items

[1] Docker. *Compose: Defining and Running Multi-Container Docker Applications*. [Online]. URL: https://docs.docker.com/compose/.

[2] JWT. *[JSON Web Token]*. [Online]. URL: https://jwt.io/introduction.

[3] GeeksforGeeks. *[Beginner's Guide to User Authentication in MERN Stack]*. [Online]. URL: https://www.geeksforgeeks.org/mern/beginners-guide-to-user-authentication-in-mern-stack/.

[4] Unsplash. *[Unsplash]*. [Online]. URL: https://unsplash.com.

[5] Amazon. *AWS EC2*. [Online]. URL: https://aws.amazon.com/ec2/.

[6] Facebook. *React*. [Online]. URL: https://react.dev/.

[7] React. *[React Router]*. [Online]. URL: https://reactrouter.com/.

[8] Fetch API. *[Fetch API]*. [Online]. URL: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API.

[9] Ryan Dahl. *Node.js*. [Online]. URL: https://www.nodejs.org/.

[10] Douglas Christopher Wilson. *Express*. [Online]. URL: https://expressjs.com/.

[11] Bcrypt. *[Bcrypt]*. [Online]. URL: https://www.npmjs.com/package/bcrypt.

[12] Dwight Merriman, Eliot Horowitz, and Kevin Ryan. *MongoDB*. [Online]. URL: https://www.mongodb.com/.

[13] Facebook. *Jest is a delightful JavaScript Testing Framework with a focus on simplicity*. [Online]. URL: https://jestjs.io/.