

Travel Tracker

Olivia Folsom, Tasmia Iqbal, Panhapich Leang, Olivia Tarsillo

Deliverable Version 3.0

Chapter 1

Purpose

Travel Tracker is a web application that aims to streamline vacation planning into one central location. Our application allows users to manage the logistics of their trip(s), such as budgeting, flight information, stay information, itinerary, and track interests such as excursions, activities, and additional notes. Travel Tracker is a user-friendly planner that eases travel stress and encourages strong organization. Our budgeting services, equipped with real-time currency conversions, advise users when to save money on international trips and permit users to set allowances, alleviate uncertainty, and allow users to budget accordingly.

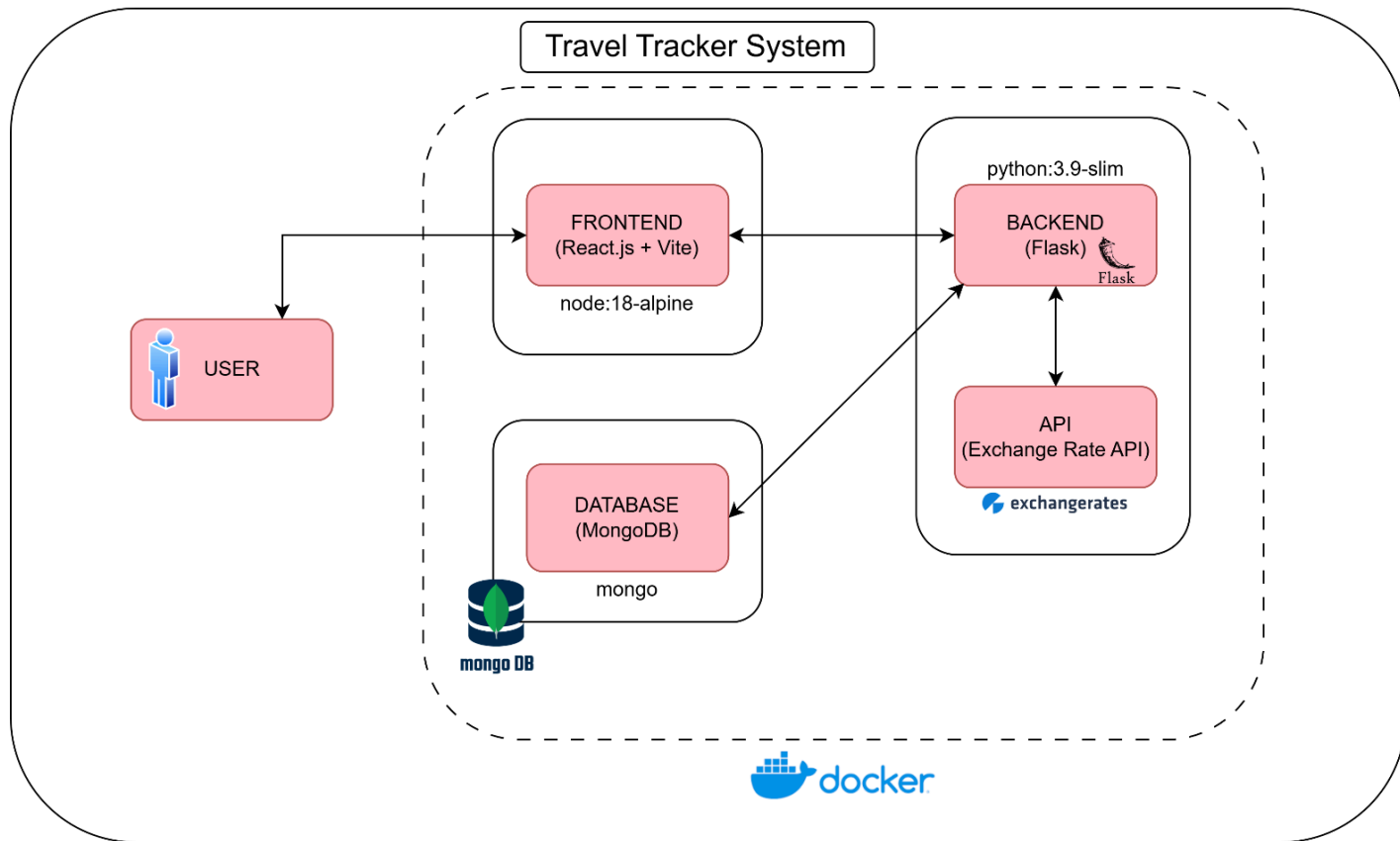
Planning vacations can be stressful and time-consuming, especially when managing multiple factors like expenses, itineraries, and currency conversions. Travelers often use separate tools to budget, convert currency, and organize their schedules, which can lead to inefficiencies and disorganization. Travel Tracker provides a user-friendly and secure platform that brings all these features together.

We plan to integrate expense tracking, budget management, itinerary planning, and real-time currency conversion into a single application :

1. Reduce travel planning stress through a centralized platform.
2. Enhance financial control by enabling users to track expenses, set allowances, and convert currencies seamlessly.
3. Improve organization with personalized itineraries and activity tracking.
4. Ensure security by safeguarding sensitive user data through robust authentication protocols.

This application encourages efficient planning, better financial management, and ultimately, a more enjoyable travel experience.

Architecture



Key Features:

- **User Authentication (Sign-up/Login)**
 - Users can create an account and log in securely using JWT authentication.
 - Ensures data privacy and secure access.
- **Vacation Budget Management**
 - Users can create a trip budget and track flight, hotel, food, and activity expenses.
 - The budgeting tool ensures expenses stay within limits
- **Expense & Banking Tracker**
 - Users can manually enter their income, expenses, and recurring payments. It helps in tracking finances efficiently.
- **Real-Time Currency Conversion**
 - Integrated with an *ExchangeRate API* to provide up-to-date currency conversions.
 - Users can convert expenses and budgets into different currencies.
- **Trip Itinerary & Activity Planner**
 - Users can add and organize their travel itineraries, including flights, hotel stays, and activities.
 - It helps users plan each day efficiently.
- **Notes & Additional Travel Info**
 - Users can add custom notes related to their trip.
 - A personalized section for important travel details.
- **Backend Data Storage & Management**
 - All user data, trip details, and transactions are securely stored in *MongoDB*.
 - Provides fast data retrieval and storage.
- **User Interface**
 - Elegant and user-friendly UI designed with *React.js*.
 - Simple navigation between budgeting, itinerary, and expenses.
- **Secure API Communication**
 - *Flask* handles all API requests, authentication, and interactions with MongoDB.
 - Ensures smooth data flow between the front end and the back end.

Chapter 2

Implementation

React: The client side of our application will be developed using **JavaScript** and **React**. This allows for functionality for user experience for easy logins, a clean user interface, and robust performance. Users can access and manage their trip details, such as budget, excursion plans, and notes. After logging in, users can input trip details stored in their user profiles. This data is synchronized with the Python-based backend through API calls. Changes in currency exchange rates will be up-to-date and displayed to the user through the ExchangeRate API. Users can monitor, edit, and save their trip plans and updates in real time.

Key Frontend Features:

- *Login/Signup Pages*: Secure authentication interfaces for user registration and login.
- *Tracker Page*: Users can log expenses, track budgets, and manage allowances. A drop-down menu allows users to convert finances into various currencies.
- *Travel Planner Page*: Users can create, manage, and edit vacation plans. The budgeting feature automatically compares expenses against the allocated budget and integrates with the currency conversion tool for international trips.
- *Currency Conversion Tool*: Real-time currency conversion using the ExchangeRate API for accurate budgeting and financial tracking.

MongoDB: MongoDB will handle user data. Each user profile will be stored, including username and password, and user inputs for budget, flight information, stay information, itinerary planning, personal notes, and interests.

Python: The backend will be developed using Python to manage server-side logic. Python will talk to the client about user data and communicate with the ExchangeRate API. This will use Flask. The back end will interact with MongoDB for data storage and retrieval.

API: To provide real-time currency conversion, we will integrate the **ExchangeRate API**. This API supplies up-to-date exchange rates for various global currencies, ensuring that users can accurately budget for international trips.

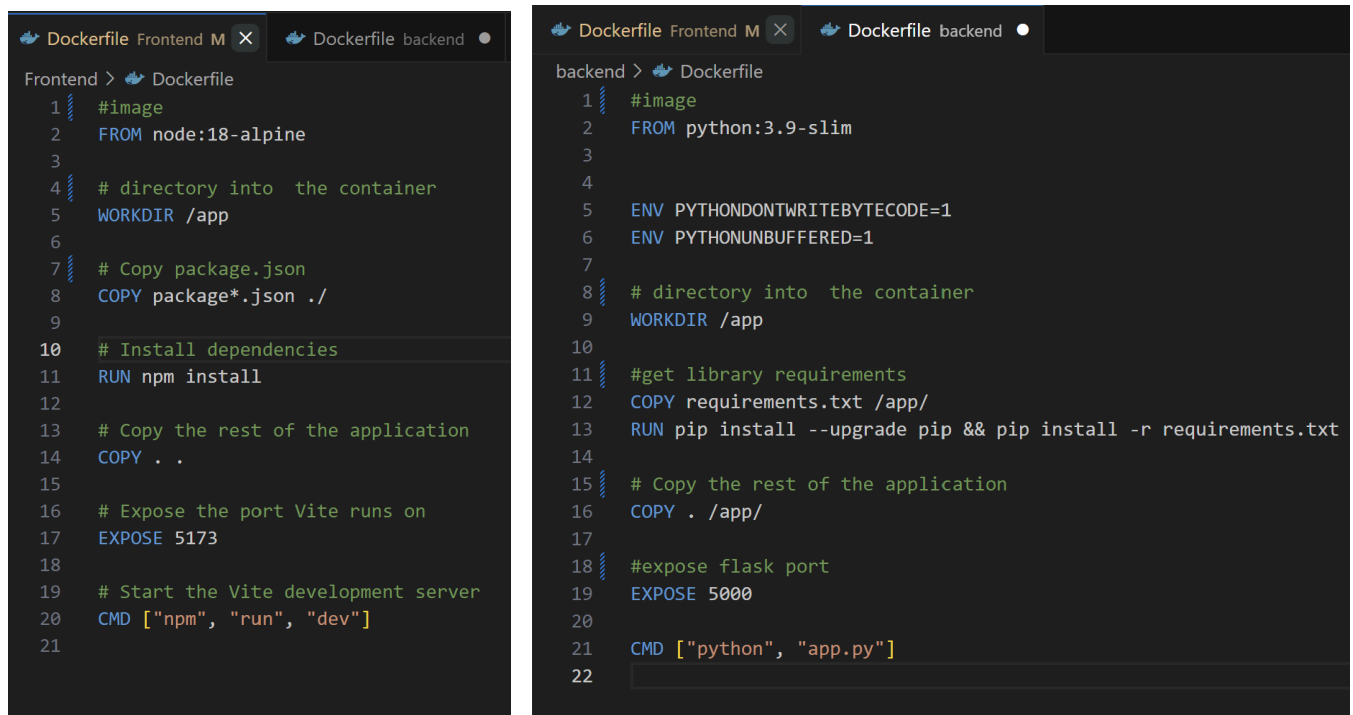
Key API Functions:

- *Currency Conversion*: Fetch current exchange rates based on user-selected currencies.
- *Automated Updates*: Regularly update exchange rates to reflect the latest market changes.
- *Integration with Expense Tracker*: Convert user-entered expenses into different currencies using API data.

Chapter 3

Dockerization

Travel Tracker is containerized into three containers: frontend, backend, and MongoDB. Containers are reliant on the ability to talk to one another to ensure that the user experience and user interactions are sufficient. Our React frontend must be able to send HTTP requests to our Flask backend for users to be able to create their accounts, log in to their accounts, and create and modify their trip information. All user input must be able to be retained in our MongoDB database. In our dockerization process, we utilize Dockerfiles to build our front-end and back-end containers. Our front end uses the official image *node:18-alpine* as our base for a lightweight OS. Our backend uses the base image *python:3.9-slim* for a lightweight deployment, which works well for our Flask backend.



The image shows two side-by-side code editors displaying Dockerfiles. The left editor is titled 'Dockerfile Frontend M' and the right is 'Dockerfile backend'. Both editors show a list of Dockerfile instructions with line numbers.

```
Frontend > Dockerfile
1 #image
2 FROM node:18-alpine
3
4 # directory into the container
5 WORKDIR /app
6
7 # Copy package.json
8 COPY package*.json ./
9
10 # Install dependencies
11 RUN npm install
12
13 # Copy the rest of the application
14 COPY . .
15
16 # Expose the port Vite runs on
17 EXPOSE 5173
18
19 # Start the Vite development server
20 CMD ["npm", "run", "dev"]
21
22
```

```
backend > Dockerfile
1 #image
2 FROM python:3.9-slim
3
4
5 ENV PYTHONDONTWRITEBYTECODE=1
6 ENV PYTHONUNBUFFERED=1
7
8 # directory into the container
9 WORKDIR /app
10
11 #get library requirements
12 COPY requirements.txt /app/
13 RUN pip install --upgrade pip && pip install -r requirements.txt
14
15 # Copy the rest of the application
16 COPY . /app/
17
18 #expose flask port
19 EXPOSE 5000
20
21 CMD ["python", "app.py"]
22
```

These Dockerfiles are built by our `docker-compose.yml`, which calls for both the `./frontend` and `./backend` builds. `Docker-compose.yml` is responsible for building our MongoDB database container. This container uses the official image for a MongoDB database, *mongo*.

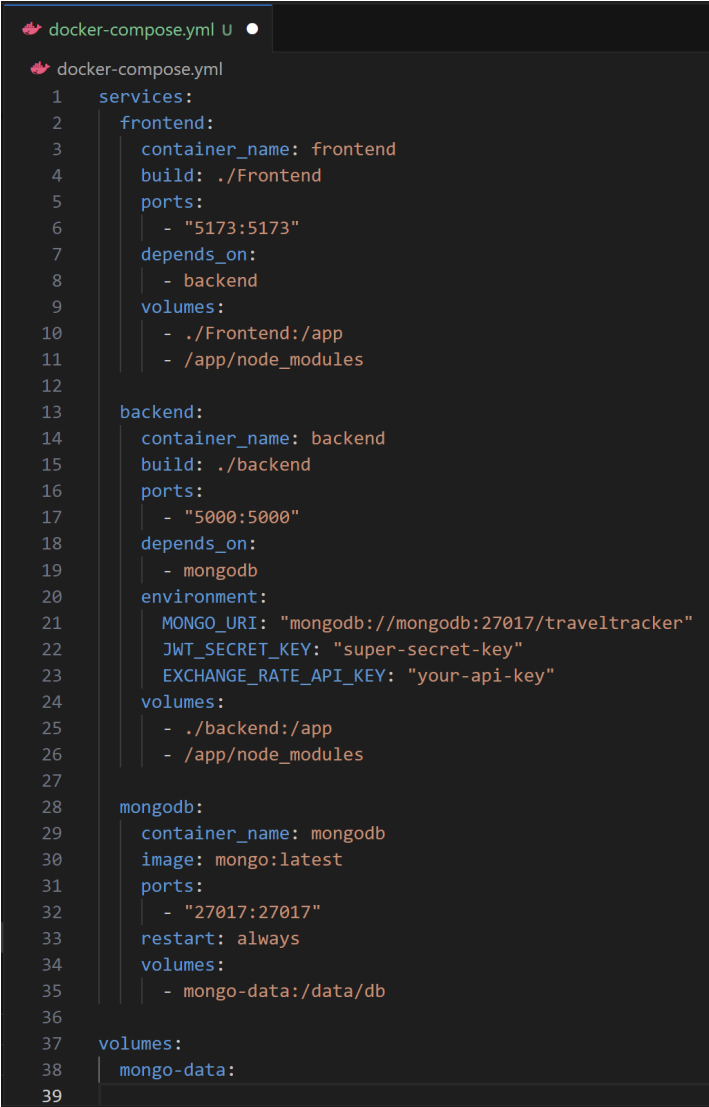
Travel Tracker's data is primarily collected through user interactions. Users create accounts, log in, and create/edit trip information. Our schema is specified in the backend configuration using

an `init.js` file. In the future, we intend to supply the database with a sample data python script that, upon the container's first initialization, our database will be ready for the backend interactions. Data collection occurs for all saved user input, like account information and trip information. Data is sent from the front end through a POST request to the backend and then stored in the collections in our MongoDB database.

Our database is structured into two collections: *users* and *trips*. As MongoDB stores individual records as documents, our collections are organized by user ID. For each new trip a user adds, a new document is added to the trips collection. The *user's* collection specifies username, email, and password. The *trips* collection formats each trip so that the user can specify their budget, different expenses like flight and hotel, itinerary, and notes. It is important that users have the functionality to not be restricted to a specified number of itinerary entries or character length for notes. Data is stored persistently, and the use of Docker volumes ensures that data is not lost during container restarts.

Our database must persist; thus, our volume configuration stores data in `mongo-data`. We also use volumes for our front-end and back-end containers. Specifying the volumes allows persistent data to ensure that user data is kept. For example, the `/app/node_modules` volume helps keep dependencies intact. By specifying `depends_on`, we know that the backend must wait for the frontend and MongoDB to be ready before starting. This helps avoid race conditions when one service is waiting for another to be up.

To test container communication, we simulated user login requests to ensure that the frontend can make requests, the backend can process the requests, and MongoDB responds. In addition to this, we established a simple test button to quickly establish that the front-end and back-end could talk to each other. Preliminary results show us that our user data is being stored. Our team has successfully built our images and has found success in running the services. The simplicity of *Docker Compose build* and *Docker Compose up* allows for the efficient deployment of Travel Tracker.



```
1  services:
2    frontend:
3      container_name: frontend
4      build: ./Frontend
5      ports:
6        - "5173:5173"
7      depends_on:
8        - backend
9      volumes:
10       - ./Frontend:/app
11       - /app/node_modules
12
13    backend:
14      container_name: backend
15      build: ./backend
16      ports:
17        - "5000:5000"
18      depends_on:
19        - mongodb
20      environment:
21        MONGO_URI: "mongodb://mongodb:27017/traveltracker"
22        JWT_SECRET_KEY: "super-secret-key"
23        EXCHANGE_RATE_API_KEY: "your-api-key"
24      volumes:
25       - ./backend:/app
26       - /app/node_modules
27
28    mongodb:
29      container_name: mongodb
30      image: mongo:latest
31      ports:
32        - "27017:27017"
33      restart: always
34      volumes:
35       - mongo-data:/data/db
36
37  volumes:
38    mongo-data:
```

Services: Defines the individual services (frontend, backend, MongoDB).

- **frontend:** Runs the frontend application, exposes port 5173, and depends on the backend.
- **backend:** Runs the backend API, exposes port 5000, depends on MongoDB, and uses environment variables to configure MongoDB and API keys.
- **MongoDB:** Runs the MongoDB database, stores data persistently using a mounted volume, and exposes port 27017.

Our Docker setup provides an efficient way to containerize and manage a multi-service application, including the front-end, back-end, and database. Docker Compose simplifies running all these services together with clear configuration, and the use of Dockerfiles ensures consistency across development and production environments. We are currently using Docker Compose for local development and multi-container orchestration. While Kubernetes is our intended deployment tool, we are still finalizing component integration and data validation. Full deployment and Kubernetes YAMLS will be prepared in the next stage.

```
app.py x
backend > app.py > ...
98 def exchange_rate():

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

tasmia@Tasmas-MacBook-Air Travel-Tracker-main % docker-compose up --build
[+] Building 17.5s (24/24) FINISHED
=> [backend internal] load build definition from Dockerfile
=> => transferring dockerfile: 592B
=> [backend internal] load metadata for docker.io/library/python:3.9-slim
=> [backend auth] library/python:pull token for registry-1.docker.io
=> [backend internal] load .dockerignore
=> => transferring context: 2B
=> [backend 1/5] FROM docker.io/library/python:3.9-slim@sha256:e52ca5f579cc58fed41efcbb55a0ed5dccc6c7a156cba76acfb4ab42fc19dd00
=> => resolve docker.io/library/python:3.9-slim@sha256:e52ca5f579cc58fed41efcbb55a0ed5dccc6c7a156cba76acfb4ab42fc19dd00
=> [backend internal] load build context
=> => transferring context: 178B
=> CACHED [backend 2/5] WORKDIR /app
=> CACHED [backend 3/5] COPY requirements.txt .
=> CACHED [backend 4/5] RUN pip install --upgrade pip && pip install --no-cache-dir -r requirements.txt
=> CACHED [backend 5/5] COPY . .
=> [backend] exporting to image
=> => exporting layers
=> => exporting manifest sha256:33a10df4f72bdc27a40dd5ef32d9c731140fd009a8a7e0759ebc5ecd7b99b4b
=> => exporting config sha256:101a037e5929df9afe0802ab9b508a97250a07ce6e9e22114e443a7705bf7f5
=> => exporting attestation manifest sha256:7f3da40aed96733f18ad2e59aa6dd9cb8b3003c27d948d695358ecae331bf3f
=> => exporting manifest list sha256:d98450d46948a3302cf09e1e4f8e001bed7369064e1f09a841b1e18da74898a5
=> => naming to docker.io/library/travel-tracker-main-backend:latest
=> => unpacking to docker.io/library/travel-tracker-main-backend:latest
=> [backend] resolving provenance for metadata file
=> [frontend internal] load build definition from Dockerfile
=> => transferring dockerfile: 447B
=> [frontend internal] load metadata for docker.io/library/node:18-alpine
=> [frontend auth] library/node:pull token for registry-1.docker.io
=> [frontend internal] load .dockerignore
=> => transferring context: 2B
=> [frontend 1/5] FROM docker.io/library/node:18-alpine@sha256:8d6421d663b4c28fd3ebc498332f249011d118945588d0a35cb9bc4b8ca09d9e
=> => resolve docker.io/library/node:18-alpine@sha256:8d6421d663b4c28fd3ebc498332f249011d118945588d0a35cb9bc4b8ca09d9e
=> => sha256:02bb84e9f3412827f177bc6c020812249b32a8425d2c1858e9d71bd4c015f031 443B / 443B
=> => sha256:8bfa36aa66ce614f6da68a16fb71f875da8d623310f0cb08aee1ecfa092f587f6 1.26MB / 1.26MB
=> => sha256:d84c815451acba96b6e6db47992922bec57121dfe10cc5b128c5c2dbaf10a 39.66MB / 39.66MB
=> => extracting sha256:d84c815451acba96b6e6db47992922bec57121dfe10cc5b128c5c2dbaf10a
=> => extracting sha256:0bfa36aa66ce614f6da68a16fb71f875da8d623310f0cb08aee1ecfa092f587f6
=> => extracting sha256:02bb84e9f3412827f177bc6c020812249b32a8425d2c1858e9d71bd4c015f031
=> [frontend internal] load build context
=> => transferring context: 266.06kB
=> [frontend 2/5] WORKDIR /app
=> [frontend 3/5] COPY package*.json ./
=> [frontend 4/5] RUN npm install
=> [frontend 5/5] COPY . .
=> [frontend] exporting to image
=> => exporting layers
=> => exporting manifest sha256:fad99c62d43ba64d1b9075e18dd917968b6cf1522e2b8c2caae361c0ad94ceac
=> => exporting config sha256:2033418d7157db1ee54b5fb380248225f1299e40ab7f8da098e36d2631d628
=> => exporting attestation manifest sha256:34136b08e30bc71f1f5cb9e7bae208114ec3f363f179cddf230c860b76a1d9fa0
=> => exporting manifest list sha256:51ebc83592c02d5d45af6ba76e1a42f1f0acdab0e8aac4036a062b54bb005
=> => naming to docker.io/library/travel-tracker-main-frontend:latest
=> => unpacking to docker.io/library/travel-tracker-main-frontend:latest
=> [frontend] resolving provenance for metadata file
[+] Running 3/3
✓ Container mongodb Running
✓ Container frontend Recreated
✓ Container backend Recreated
Attaching to backend, frontend, mongodb
```


Chapter 4: Final Results

By the end of the project timeline, the Travel Tracker application successfully achieved the majority of its intended goals. The final system consists of three containerized services: frontend (React), backend (Flask/Python), and a MongoDB database, managed through Docker Compose. The frontend allows users to sign up, log in, create and manage travel plans, track expenses, and convert currencies in real-time. The backend ensures secure authentication, handles all API interactions, and manages the application's business logic. MongoDB provides persistent storage for user and trip data, maintaining security and organization of critical information.

Key Functionalities Implemented

- **Containerized Deployment:** Frontend, backend, and database running in isolated, scalable containers through Docker Compose.
- **User Authentication:** Full signup and login capabilities with JWT authentication.
- **Vacation Budget Management:** Ability to set travel budgets, record expenses (flight, hotel, food, activities), and monitor budget status.
- **Expense Tracker:** Manual entry of expenses and income with support for recurring transactions.
- **Currency Conversion:** Real-time currency updates using the ExchangeRate API, integrated into both expense tracking and budgeting modules.
- **Persistent Database Storage:** MongoDB volumes are configured to maintain data across container restarts.

Challenges Faced

- **Updated Frontend With Schema Changes:** After improving our frontend with a more complex design, we needed to change how we intended to store user and trip data to connect the backend to the frontend and ensure data was being saved. Our code had difficulty adapting to the change, and we needed to work around the problem.
- **Time Constraint:** This semester was particularly challenging so we often had a hard time finding meetup times within this time constraint.

Missed Milestones

- **No AI Implementation**
- **Late Final Completion in the Overall Timeline**
- **Kubernetes Implementation**

Testing and Validation

Testing was performed across all major functionalities to ensure proper communication between the frontend, backend, and database components. Key testing activities included:

- Simulated user account creation, login, and session management.
- CRUD operations for trips and expenses.
- Currency conversion accuracy checks using live API data.
- Container resilience testing by restarting services and validating persistent data.

While full-scale automated testing was not completed, manual testing was thorough across multiple use cases.

Future Ideas

In the future, the following ideas will propel Travel Tracker's complexity, intrigue, and industry-competitiveness:

- **Kubernetes Deployment:** Completing the implementation of Kubernetes would make the application even more scalable and production-ready.
- **Improved UI/UX:** A polished design enables the user to be excited by the application, especially to make it mobile-friendly.
- **Automated Testing:** Building a full set of unit and integration tests to ensure smooth deployment, testing, and quality testing.
- **Trip Sharing/Other Tabs:** Letting users share their trips with friends or plan group vacations.
- **AI Recommendations:** Integrating a large language model to suggest hotels, activities, or destinations based on a user's budget and preferences would improve user experience and alleviate trip planning stress.
- **Advanced Uploads:** Allowing users to upload images or tickets for flights would enable advanced trip planning. Creating a hub for all the user's information with more advanced fields for flights, stays, and itinerary would benefit the user.

Conclusion

Creating, designing, and building Travel Tracker has been a rewarding learning experience for our team. Over the course of the project, we built a full-stack web application from the ground up, in which the team learned the hands-on skills of cloud deployment and web development and also the importance of teamwork and communication when working toward a goal.

One of the biggest takeaways from this project is how powerful containerization can be. Using Docker and Docker Compose simplified how we manage our different services: frontend, backend, and database in a consistent way that is replicable and scalable. We also learned about how APIs work in real-time applications, how to handle user authentication securely, and how to

store and manage user data properly with MongoDB. There were some challenges along the way, like reformatting our project's endpoints and managing the different dependencies required for the containers when they spin up. Working through these issues helped us improve our problem-solving, and we adapted in the event a challenge arose.

Overall, we are proud of the current state of Travel Tracker and our work to create it. Travel Tracker is a solid foundation that has potential to be expanded upon. Most importantly with this project, we learned through a real-world application of building an application to be deployed on the cloud. We learned through not just writing code, but designing systems, solving unexpected problems, and working together as a team.