SmartDashboard Final Project Report

Students: Ben Zehavi, Yamit Patik, Inbar Miran

1. Application Overview

SmartDashboard is a comprehensive web application designed to help business owners and their teams turn raw data into actionable insights. In today's data-driven world, many businesses have access to valuable information but lack the tools to easily interpret it. SmartDashboard solves this problem by providing a secure, multi-user platform where users can upload business data, generate intelligent summaries, and create custom visualizations using the power of AI.

The application is built with a modern, containerized architecture using Docker, which ensures that it can be run reliably on any machine. The backend is powered by Flask and uses MongoDB for persistent data storage. The AI capabilities are provided by a separate microservice that leverages Google's Gemini API. This multi-container design ensures that the application is both scalable and resilient, which are key requirements for any modern web application.

A key feature of SmartDashboard is its collaborative environment. Business owners can invite other users as "editors," allowing for seamless collaboration on business data and visualizations. To prevent conflicts, the application uses WebSockets to ensure that only one user can edit a business's plots at a time, providing a smooth and professional user experience.

2. Key Features

- **Secure User Authentication**: Users can sign up and log in to a secure environment. All passwords are a-kind salted and hashed using `bcrypt`, ensuring maximum security for user credentials.
- **Business-Centric Organization**: After logging in, users can create and manage multiple businesses. Each business serves as a separate workspace with its own data, plots, and editors, allowing for a clean and organized user experience.
- **Collaborative Editing**: Business owners can grant editing privileges to other users, allowing them to upload files, generate plots, and edit the business's details. This feature turns SmartDashboard into a powerful tool for team collaboration.
- **AI-Powered Plot Generation**: The application's flagship feature allows users to generate custom plots and visualizations from their data by simply describing what they want to see in plain English (e.g., "Create a bar chart of total sales per city").

- **Real-Time Editing Lock**: To prevent conflicts and ensure data integrity, the application uses WebSockets to implement a real-time locking mechanism. This ensures that only one user can edit a business's plots at a time, providing a smooth and professional multi-user experience.

3. Testing Strategy

Our testing strategy is designed to be comprehensive, covering all aspects of the application's functionality and security, as required by the project guidelines.

Unit Tests

- **Purpose**: To test individual functions and components in isolation.
- **Implementation**: Our unit tests cover the application's data models, ensuring that data is correctly serialized and deserialized when communicating with the database. We also have unit tests for the database manager itself, which verify that our database queries are constructed correctly.

Integration Tests

- **Purpose**: To test how different parts of the application work together.
- **Implementation**: Our integration tests cover the interaction between the web application and the LLM service. We use mocks to simulate the behavior of the LLM service, allowing us to test how our application handles both successful and failed responses from the AI.

System (End-to-End) Tests

- **Purpose**: To simulate a user's entire journey through the application.
- **Implementation**: We have a suite of system tests that cover the full user workflow, from registering and logging in to creating a business, uploading a file, and generating a plot. These tests ensure that all the application's features work together seamlessly from the user's perspective.

Security Tests

- **Purpose**: To ensure that the application is secure and that user data is protected.
- **Implementation**: Our security tests verify that all protected pages require a user to be logged in. They also test the login and logout functionality to ensure that sessions are correctly created and destroyed.

Stress Tests

- **Purpose**: To evaluate how the application performs under a high volume of traffic.
- **Implementation**: We used the Locust framework to simulate a high number of concurrent users accessing the application. This allowed us to identify performance bottlenecks and ensure that the application remains stable and responsive under pressure.

4. Risk Assessment

Availability & Redundancy

- **Risk**: If a container crashes, the application could become unavailable.
- **Mitigation**: Our application is designed with a multi-container architecture using Docker Compose. If a container fails, Docker's restart policies can be configured to automatically restart it. In a production environment, this would be managed by a container orchestration platform like Kubernetes, which provides advanced self-healing capabilities.

Scalability

- **Risk**: A sudden increase in traffic could slow down the application.
- **Mitigation**: Our stress tests with Locust revealed that the AI-powered plot generation is the most resource-intensive feature. To scale the application, we could:
  - Increase the number of `web` container replicas to handle more concurrent users.
  - Implement a caching layer (e.g., with Redis) to store the results of frequently requested plots, reducing the load on the LLM service.

Spamming Requests

- **Risk**: A malicious user could send a large number of requests to overload the server.
- **Mitigation**: We have implemented rate limiting using the `Flask-Limiter` library. This restricts the number of requests a user can make in a given time frame, protecting the application from spam and abuse.

Security

- **Risk**: User data could be compromised.
- **Mitigation**: We have implemented several security measures:
  - **Password Hashing**: All user passwords are securely hashed using `bcrypt`, ensuring that even if the database is compromised, the passwords are not exposed.
  - **Containerization**: Only the `web` container is exposed to the internet. The database and LLM service are only accessible from within the Docker network, providing a strong layer of isolation.

- **Authentication**: All sensitive routes are protected and can only be accessed by logged-in users.

Main problems we encountered

1. Outdated and Inconsistent Code

As you've developed your application, you've transitioned from a single-user model to a more complex, multi-business architecture. However, some of the older code was not updated to reflect this change, which has been the root cause of many of the issues we've addressed.

- **Problem**: Many of your application's routes and database functions were still trying to use a `user_id` to find and save data, even though the application is now organized around businesses. This was causing "business not found" errors and preventing data from being saved correctly.
- **Solution**: We've updated your `views.py` and `db_manager.py` files to consistently use the `business_id` when fetching and saving data. This has resolved the errors and ensures that your application's logic is consistent with its new architecture.

2. Frontend and Backend Mismatches

Several of the issues you've encountered have been caused by a disconnect between your frontend JavaScript and your backend Python code.

- **Problem**: Your JavaScript files were sending requests to outdated URLs (e.g., `/edit_plots` instead of `/edit_plots/<business_name>`), which was causing "404 Not Found" errors.
- **Solution**: We've updated your JavaScript files to correctly build the URLs for your API requests, ensuring that the frontend and backend can communicate effectively.

3. Lack of Real-Time Communication

Your application's multi-user environment requires a way to keep all users in sync. Without a real-time communication channel, it's possible for one user's actions to unknowingly conflict with another's.

- **Problem**: Without a real-time locking mechanism, it was possible for two users to edit the same business at the same time, which could lead to data loss.
- **Solution**: We've implemented a real-time editing lock using WebSockets. This ensures that only one user can edit a business at a time, preventing conflicts and providing a better user experience.

By addressing these core issues, we've transformed your application from a single-user prototype into a robust, multi-user platform. The final step is to complete the documentation to ensure that your project is ready for submission.