Group 2 - Final Project Report [Complete]

SENG 401 - Winter 2024

Dr. Ronnie de Souza Santos

Aryan Sharma
Krishna Shah
Lujaina Eldelebshany
Mohamed Amara
Nour Ajami
Youssef Saad
Zuhaer Rahman

Table of Contents

Table of Contents **2**
Inception **3**
Architectural Diagram **4**
Project Links **4**
Sequence Diagrams **5**
Class Diagram **7**
Non-Functional Requirements **11**
Design and Architecture Patterns **12**
UI/UX **15**
RTM **16**
Test Plan **18**
Test Explanation **19**
Miscellaneous **23**
AI Citation 23
User stories 24
Use Cases 25
Changes in development process 26
Teamwork process 26

Inception

Date: January 25, 2024 – March 27, 2024

Purpose

This document outlines the development, features, and deployment of our financial planning application tailored specifically for Generation Z. Aimed at fostering financial literacy from an early age, this application is designed to empower users aged 15-25 to make better financial decisions. By integrating features such as expense tracking, budget management, insightful reporting, and educational resources, the application seeks to instill a foundation of financial wisdom that users will carry forward. The application also includes goal setting and secure authentication mechanisms to enhance engagement and ensure privacy.

Versioning

This document represents the first version of the application, which is all encompassing since its inception, and plans for future updates. We aim to keep consistent versioning, highlighting key differences between each major software update. Versioning ensures clarity in the development process, allowing stakeholders to track changes and improvements over time.

Team Details

Aryan Sharma – Educational page and system testing, Report
Krishna Shah – User Login/Registration Functionality, Budget Backend/Frontend, RTM
Lujaina Eldelebshany – Expenses Frontend/Backend, Report
Mohamed Amara – Excel Export Backend, System testing, Report
Nour Ajami – Backend Lead, Database work, Frontend integration, Authentication/Authorization
Youssef Saad – Frontend, UI design, Deployment, Report
Zuhaer Rahman – Budget Frontend

Status [Complete]

As of March 27, 2024, the project is ready for submission, with a focus on system documentation.

Executive Summary

Spending habits within our generation are beyond atrocious. By combining practical tools with educational content, we aim to create an environment where young individuals are encouraged to set and achieve their financial goals. This document serves as a comprehensive guide to the

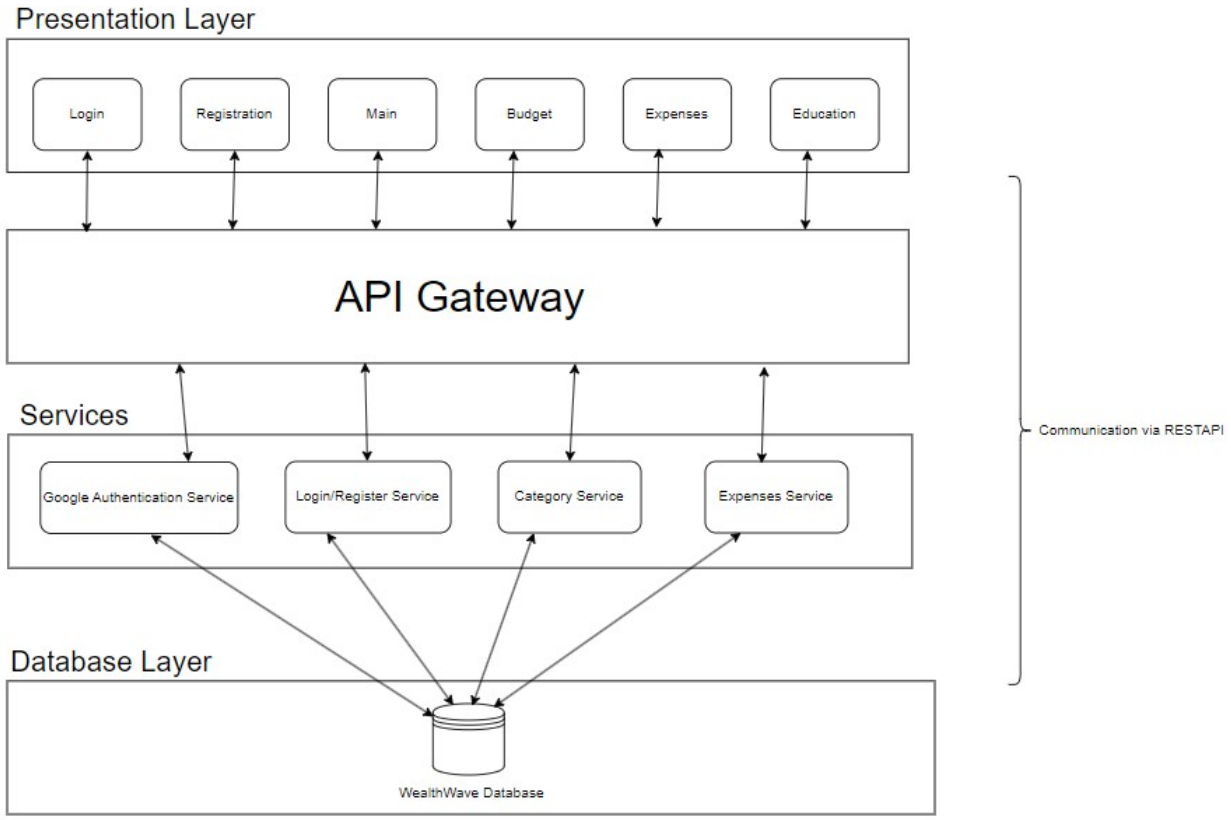
application's features, development process, and future roadmap, highlighting our commitment to enhancing financial literacy among young adults.

Future Developments

Based on user feedback, we aim to introduce new features and improvements such as integrating artificial intelligence to predict user behavior and suggest improvements, gather user information over several months to be able to provide insights such as tracking, habit trends, and recommendations based on user behavior. Our goal is to remain with the trend at the forefront of financial education technology, continuously expanding our system features to anticipate the needs of our users.

Architectural Diagram

Figure 1: Architectural Diagram



Project Links

Project Deployment: https://wealthwave-414021.web.app/

- Please make sure you are in a private/incognito tab for deployment to ensure full functionality

Sequence Diagrams

Figure 2: Registration and Login

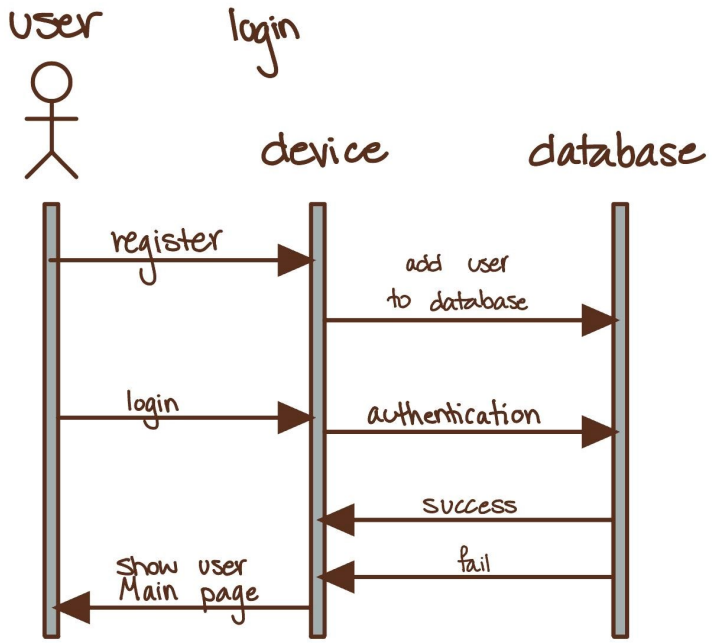


Figure 3: Monthly Budget Tracking (Expenses page)

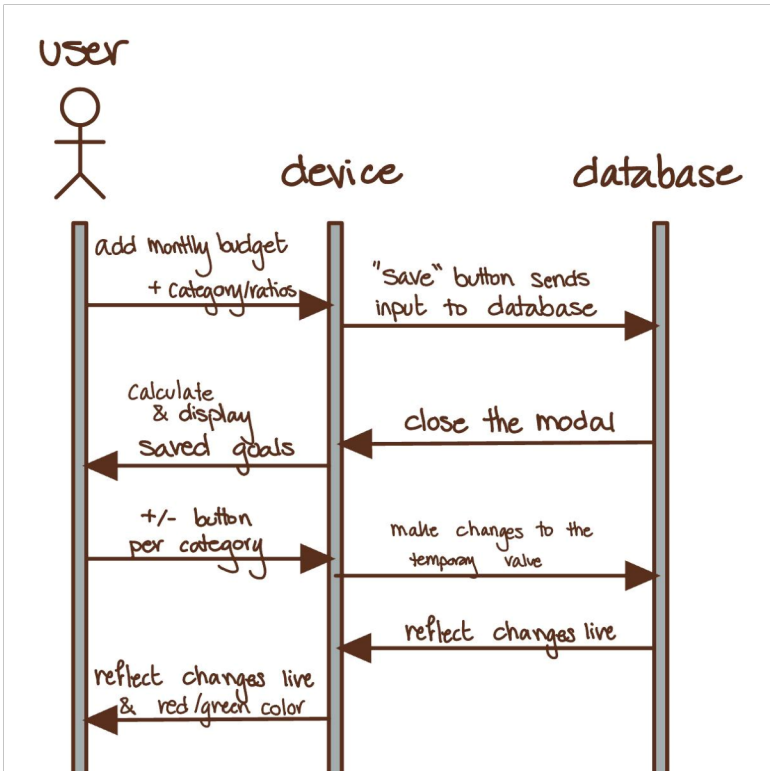
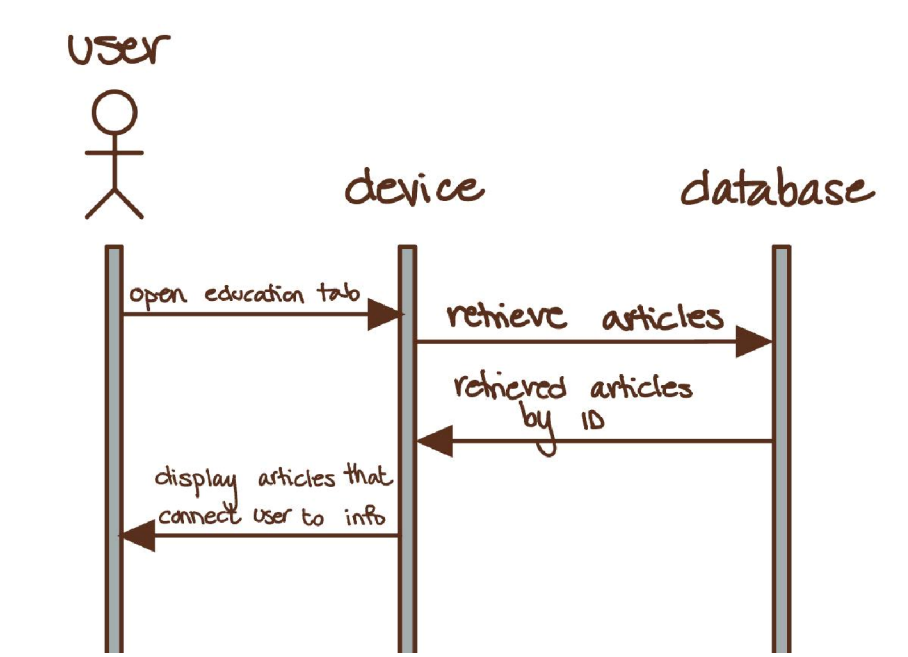
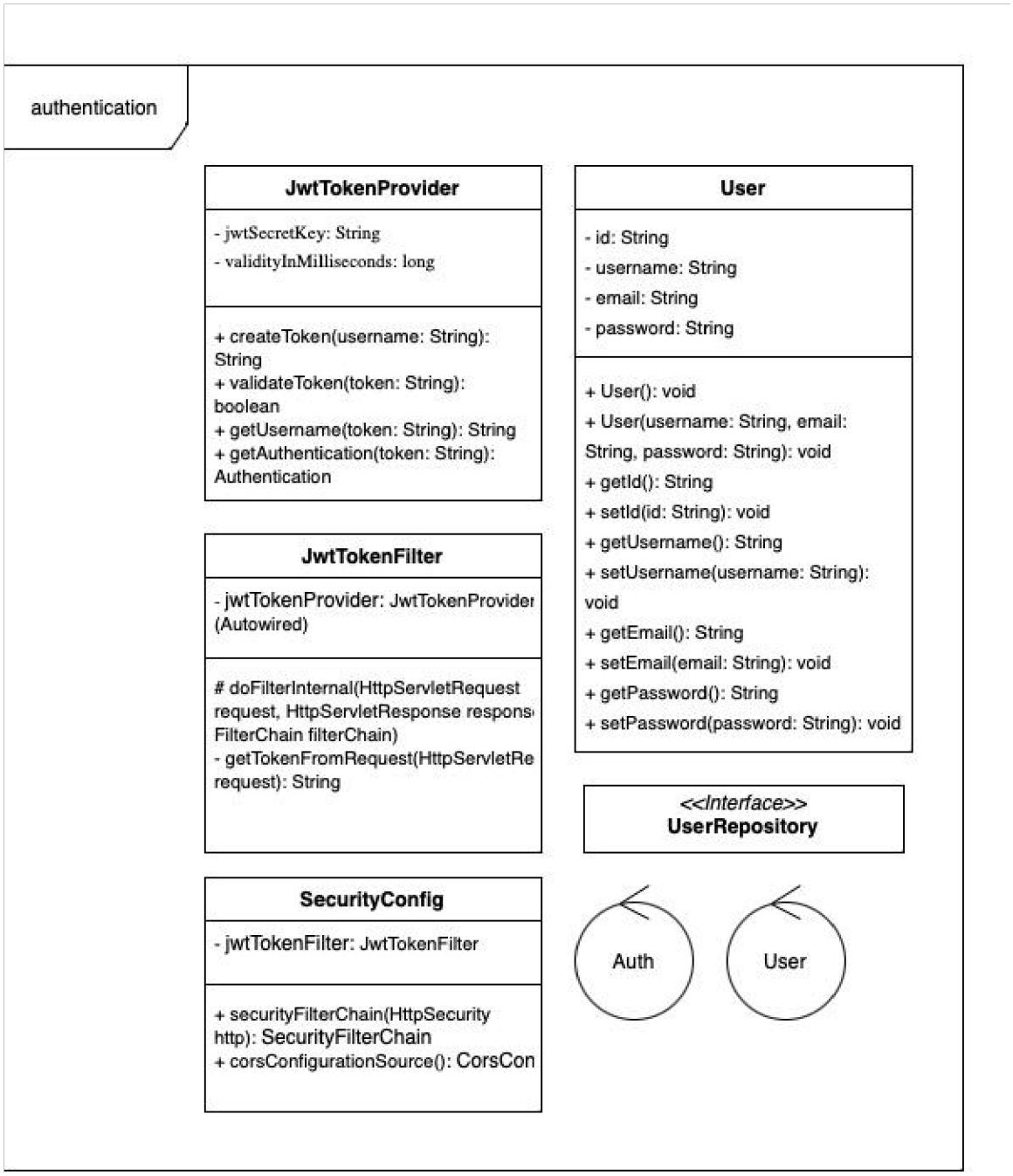
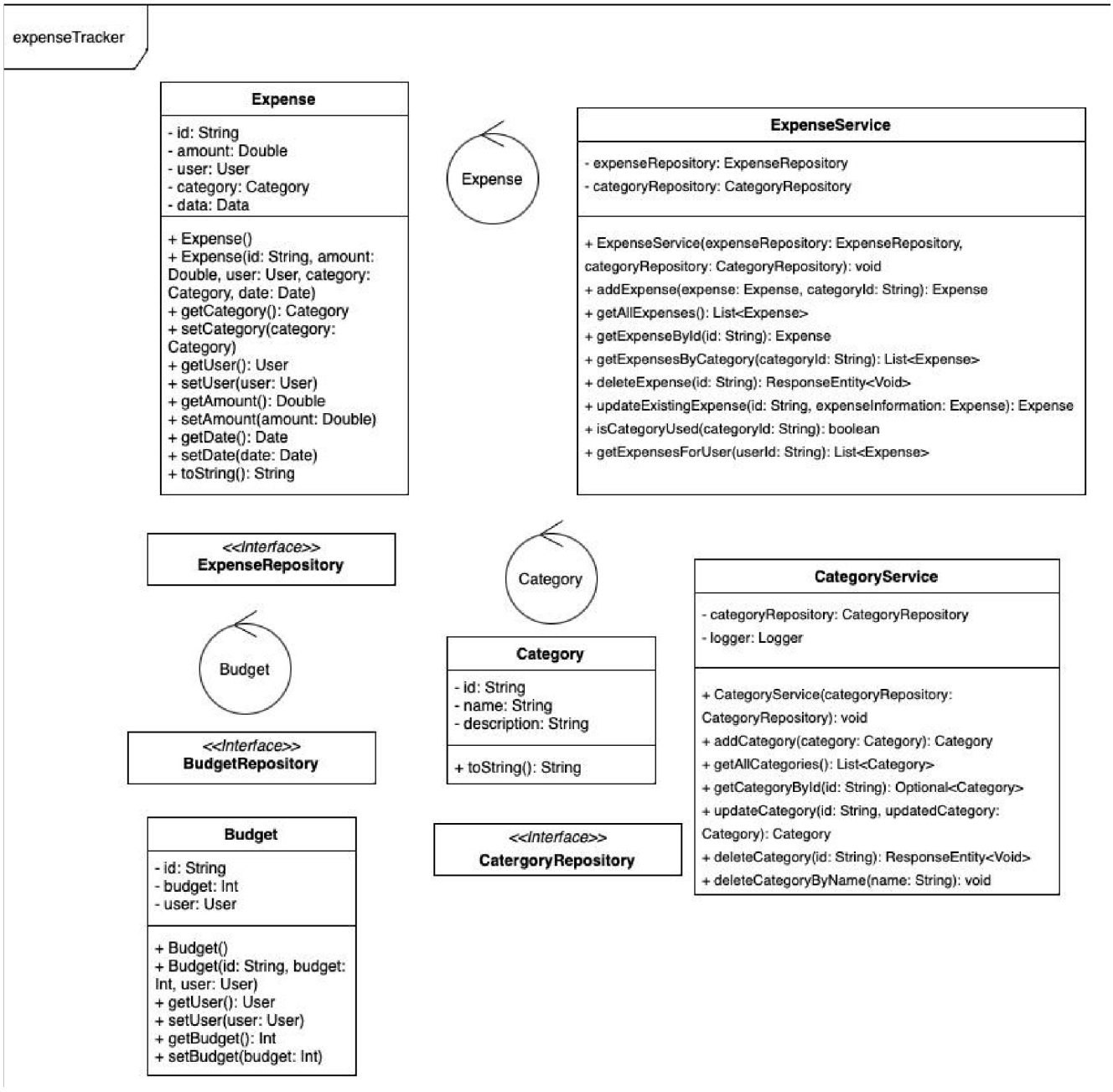


Figure 4: Accessing articles (Education Page)



Class Diagram




expenseTracker
Expense
- id: String
- amount: Double
- user: User
- category: Category
- data: Data
+ Expense()
+ Expense(id: String, amount: Double, user: User, category: Category, date: Date)
+ getCategory(): Category
+ setCategory(category: Category)
+ getUser(): User
+ setUser(user: User)
+ getAmount(): Double
+ setAmount(amount: Double)
+ getDate(): Date
+ setDate(date: Date)
+ toString(): String
Expense
ExpenseService
- expenseRepository: ExpenseRepository
- categoryRepository: CategoryRepository
+ ExpenseService(expenseRepository: ExpenseRepository, categoryRepository: CategoryRepository): void
+ addExpense(expense: Expense, categoryId: String): Expense
+ getAllExpenses(): List<Expense>
+ getExpenseById(id: String): Expense
+ getExpensesByCategory(categoryId: String): List<Expense>
+ deleteExpense(id: String): ResponseEntity<Void>
+ updateExistingExpense(id: String, expenseInformation: Expense): Expense
+ isCategoryUsed(categoryId: String): boolean
+ getExpensesForUser(userId: String): List<Expense>
<<Interface>>
ExpenseRepository
Budget
<<Interface>>
BudgetRepository
Budget
- id: String
- budget: Int
- user: User
+ Budget()
+ Budget(id: String, budget: Int, user: User)
+ getUser(): User
+ setUser(user: User)
+ getBudget(): Int
+ setBudget(budget: Int)
Category
Category
- id: String
- name: String
- description: String
+ toString(): String
<<Interface>>
CatergoryRepository
CategoryService
- categoryRepository: CategoryRepository
- logger: Logger
+ CategoryService(categoryRepository: CategoryRepository): void
+ addCategory(category: Category): Category
+ getAllCategories(): List<Category>
+ getCategoryById(id: String): Optional<Category>
+ updateCategory(id: String, updatedCategory: Category): Category
+ deleteCategory(id: String): ResponseEntity<Void>
+ deleteCategoryByName(name: String): void

Functional Requirements

Welcome View
The homepage is the initial interaction point for visitors, which is meant to capture their attention and encourage them to sign up. It introduces the program's purpose and core functionalities, ensuring users feel guided from the outset. All done in an easy and friendly format.

Navigation Bar
The navigation bar provides seamless and intuitive access to different features. Designed with usability in mind, it allows users to effortlessly transition between different sections, such as expense tracking, budget overview, and educational articles or to even sign-out.

User Login/Google
The login feature offers a secure and convenient sign-in experience, with support for Google authentication. Along with regular secure login from the system database, Google integration allows users to access their accounts using existing Google credentials consequently making the user’s life easier.

User Registration
To allow customers the ability to use our site, the registration process is streamlined and user-friendly, allowing new users to create an account with minimal effort and only the necessary information.

Expense Allocation
A total overarching budget allocation is assigned here. The user can choose to input their monthly budget as a reference point. This will allow the program to ensure the user does not over-allocate. This section comes with instructions.

Expense Section Addition
The expense allocation feature allows users to assign expenses to specific categories, aiding in a granular approach to their financial tracking and management. Users can assign customizable categories to accommodate various user needs, allowing the user autonomy over their program usage. Each category includes a user specified budget which the program will ensure stays within the total budget allocation. All changes to these categories are saved/editable and will be displayed in addition to the ‘goal’ monetary amount.

Budget Overview
The overview presents a comprehensive summation of the user's financial situation based on their inputs. Displaying key metrics, it allows the user to have a visual understanding of their spending habits and financial health at a glance. Visuals are usually easier to learn from.

CSV Export
Our CSV export functionality provides users with the option to download their financial data. CSV files are easy to work with as they are widely accepted. Furthermore, it gives us room for future implementation of data analytics.

Article Education Page
The education page is paramount to living up to our vision. Carrying on the idea of financial literacy for the next generation, many articles were collected on topics such as budgeting strategies, expense management, saving, insurance, and so on. This resource is updated to reflect the current societal economic state.

Footer Contact Links
Lastly, a collection of contact options is located in the program's footer. The footer offers direct access to various communication channels, including customer support and social media platforms. This ensures users can easily reach out for assistance, fostering a responsive user experience.

Sign-out Button
In order for the user to quit the program, a sign-out button is conveniently placed within the navigation bar. The users' information will be retained for when they sign back in.

Simulate Purchase
To simulate a user using their budgeted resources, we added a feature where they can add a recent purchase.

Non-Functional Requirements

Performance
The application is able to handle concurrent user access without significant degradation in performance. Moreover, the application can respond within 10 seconds for most user interactions.

Scalability
The system is scalable and can accommodate an increasing number of users and data volume.

Reliability
The application has a high level of availability with over 90% uptime. Moreover, it can handle errors effectively and provides clear messages to users in the case of a failure.

Security
Robust authentication and authorization mechanisms are implemented to prevent unauthorized access. Data is also encrypted.

Maintainability
Codebase is well documented and follows industry standards for better readability and maintainability.

Compatibility
Application is compatible with various web browsers (e.g. Edge, FireFox, Chrome). Future work will include compatibility with mobile devices (IOS, Android).

Auditability
System maintains logs of user activities. Logs can easily be read to detect issues or identify user activity.

Data Integrity
Financial data is stored securely and without any discrepancies in values. Backup and recovery procedures put in place in the case of loss of data.

Design and Architecture Patterns

The overall system architecture:

The system will adopt a microservice and n-tier architecture, which will ensure scalability and seamless integration of individual components. The key architectural elements will include:

- **Client Application:** A web application development using React, providing a friendly user interface that is easy to navigate
- **API Gateway:** This will be the entry point for the client requests that are incoming. The requests will be directed to the appropriate microservice and provide an additional layer of security.
- **Microservices:** We will implement independent services that are each responsible for specific functionality, such as the user management, expense tracking, budget management, and financial insights.
    - **Scalability for individual features:**
        - Services like expense tracking, budget management, and financial insights can scale independently. For example, if a user is heavily utilizing the budget management feature at the beginning of a new year, this service can scale without the need to scale the entire application.
    - **Easier Integration with possible External Services:**
        - The way the application is set up, it's possible to integrate with many financial institutions or API's for real-time expense tracking or to fetch financial data. Microservices architecture makes it easier to build and maintain these integrations as standalone services, which will enhance the applications capabilities with little to no impact on the existing functionalities.
    - **Feature Development and Deployment**
        - Implementing a microservice approach makes it easier for the team to develop, test, and deploy new features. For example, new methods for categorizing expenses or even machine learning models for any potential financial forecasting.
- **N-Tier:**
    - **Clear Separation between User interface and Business Logic:**
        - This application has a distinct presentation layer (frontend), business logic layer, and a data access layer. For example, if we wish to improve the interface for the expense tracking service for a greater user experience, it

will not interfere with the underlying logic that calculates and categorizes the expenses.

    - Flexibility in Data Management:
        - The data layer of our application is optimized for financial transactions, which will ensure fast, eligible access to expense records, budget allocation, and financial insights. Any upgrades that need to be done to the database can be performed without affecting the user interface part of the application,
    - Enhanced Security:
        - With the N-tier architecture in place, any sensitive data operations can be isolated in the data layer, which can be securely managed and accessed, which will minimize the risk of data breaches.
- **Database:** MongoDB (NoSQL) to store all the information necessary
- **Authentication Service:** Manages user authentication and authorization with Google authentication API ensuring secure access to the application. Additionally JW tokens are issues for necessary user identification, security, and proper request handling.

How software components interact:

- **Client to API Gateway:** The application will communicate with the backend through the API gateway, which simply routes requests to the appropriate microservice upon the request.
- **Microservice Communication:** Microservices can communicate together directly for operations that require any type of data or functionalities from other services
- **Microservices to Database:** Every single microservice will interact with its own dedicated database for CRUD operations, which will ensure data consistency and integrity throughout the entire application.
- **Authentication and Authorization:** Implemented Google's OAuth 2.0 protocol for authentication and authorization. The authentication service will essentially validate user credentials and issues tokens, which are then used by other services to authorize user actions. Additionally, regular login service will also authenticate the user and issue a token for user authentication and validation for subsequent requests to the server.

How data flows through the system:

- **User Input:** The data will enter the system through the user requests in the user interface, such as logging in, entering transactions, or setting monthly budgets.
- **Data Processing:** The microservice will process the data, and perform budget management, generating insights, and will store transaction records

- **Data Retrieval:** Users will query the system for reports, analytics, articles, budgets, or any transactional histories with the data gathered and formatted by the microservices before it gets presented in the client application.
- **Data Export:** Whenever the user requests the transactions and budgets data are compiled into a CSV format and will be made for download through the user interface.

How system is deployed:

The system deployment was done through a structured process encompassing several critical stages

- **(CI/CD) Code Integration:** we regularly merge code changes into a central repository through Github, accompanied by Junit tests.
- **Logging:** Comprehensive logging messages are in place throughout the program facilitating detailed analysis and debugging.
- **Feedback Integration:** A structured process through labs was used to gather information from the TA's to ensure our project met demands and was suitable. This ensures that the system evolves in alignment with user needs and expectations.
- **Documentation:** A comprehensive report document is provided, detailing system deployment, architecture, usage, features, and every other detail we could think of. This resource supports developers and system administrators in understanding and managing the system effectively.

How users interact with the system:

The user will login with Google Sign in, and set up their profile including financial goals and preferences. For expense tracking, the users enter transaction details manually and categorize each expense as they please. For budget management, the users create and adjust budgets for different categories, receiving notification if they are close to exceeding the limit. Users will receive personalized financial analytical insights and reports which will be generated based on their data, ultimately helping them to make informed decisions. Lastly, the users can export their transactions and budget data in a CSV format for external use or analysis.

UI/UX

The application's user interface (UI) is designed to ensure an intuitive navigation experience. You know when your google maps starts acting up so you switch to apple maps? Ya we do not want that. Catering specifically to enhance user experience, the site routing has been rigorously tested to ensure logical flow from the moment the website is loaded. Upon first access, users are greeted by a welcome page that presents an overview, tailored to visitors who have yet to make an account. This initial interaction sets the tone for the application's user-centric design.

For user authentication, the application offers two options: traditional account creation or the convenience of Google Sign-in. Both methods have been optimized for efficiency, requesting only essential information to maintain simplicity.

Upon successful login, users are directed to the home page, which guides the user to which page they should navigate to. This is effective as an introduction to new users, and a reminder for previously registered users.

The navigation bar, a constant across the application, clearly delineates each section available to the user, including dedicated budget, expenses, and education pages. This section is meticulously structured to facilitate efficient tracking and management of personal finances. Furthermore, a curated resource filled with articles, designed to empower users with knowledge, is readily available to the user, enabling them to make informed decisions about their financial well-being.

Targeting a younger demographic, the application leverages a relaxed design aesthetic, and text that resonates with the audience. A minimalistic approach was taken to reduce user visual bombardment.

To begin the development process, a preliminary design was created using Figma. This conceptual design laid the foundational design principles that were carried through to the final implementation, bearing minor changes. Access to the original Figma file is provided below, offering insight into the application's preliminary design.

By prioritizing user experience at every step, the application stands as a comprehensive and evolutionary tool for financial management and literacy enhancement. And on the development side, clean/modular code was a focus of the team.

https://www.figma.com/file/NYJ2tM34JCrKkPe6wWAD8B/Financial-Planning-Application?type=design&node-id=0-1&mode=design&t=FpGmRZBBXVJM1AEa-0

RTM

ID	Main Requirement	Sub Requirement	Design	Development	Tested ?	Test ID	Deployed?
1	Google Authentication	Allow users to create a account using google sign-in	Yes	Complete	Yes	TC1 TC10	Yes
2	Database Setup	Create MongoDB database and connect to the backend	Yes	Complete	Yes	TC7	Yes
3	Register Page	Initial Page	Yes	Complete	No	N/A	Yes
4		User Creation Functionality	Yes	Complete	Yes	TC6 TC9	Yes
5	Login Page	Initial Page	Yes	Complete	No	N/A	Yes
6		User Login Functionality	Yes	Complete	Yes	TC6 TC9	Yes
7	Main Page	Initial Page	Yes	Complete	No	N/A	Yes
8		Budget Planner	Yes	Complete	No	N/A	No
9	Budget Page	Initial Page	Yes	Complete	No	N/A	Yes
10		Displaying Expenses	Yes	Complete	Yes	TC11 - TC13	No
11	Expenses Page	Initial Page	Yes	Complete	No	N/A	Yes
12		Expense Functionality	Yes	Complete	Yes	TC11 - TC18	Yes
13	Education Page	Initial Page	Yes	Complete	No	N/A	Yes

14		Educational material	Yes	Complete	No	N/A	No
15	Settings Page	Initial Page	Yes	Complete	No	N/A	Yes
16		Custom user preferences	Yes	Incomplete	No	N/A	No

Test Plan

Unit Testing:
During the code development stage, each team member responsible for a specific section or functionality of the project conducts unit testing on their own code. This involves testing individual methods and functions to ensure they produce the expected output and are free from code smells. Unit tests are performed locally on each developer's computer, allowing for the verification of the smallest units of code in isolation. This testing phase occurs immediately after the project plan is approved.

Sectional Testing:
As our project is modular, with different team members assigned to develop specific sections of the frontend, sectional testing ensures that each page operates reliably and as expected in relation to other sections and team members' code. This testing phase aims to prevent overlapping functionalities or duplicate pages within the application. Sectional testing is conducted as developers become more familiar with their assigned functionality, allowing them to anticipate and accommodate the output from other team members' pages. It occurs progressively throughout the project development cycle.

Integration Testing:
In the final week of development, integration testing is conducted to ensure seamless interaction between different modules of the application, both frontend and backend, to fulfill all functional requirements. This testing phase requires access to the MongoDB Database and the merging of all frontend developers' branches into the main branch. Additionally, security considerations are addressed during this stage to fortify the application against potential vulnerabilities.

System Testing:
Following the successful development and integration of the application, a dedicated day is allocated for system testing. This comprehensive testing phase evaluates the end-to-end functionality of the application and verifies that non-functional requirements are met. Testing is performed on various computer setups and with different web browsers to ensure compatibility and reliability across different environments. Any final modifications are made during this stage to ensure the application performs as expected before deployment.

Test Explanation

Authentication Tests:

1. **AuthControllerTest (TC1):** Unit test for the googleAuthentication method in the AuthController class. It first creates a TokenDto object and a User object with some test data. It then sets up the mocks to return these objects when their methods are called with the appropriate arguments. @InjectMocks annotation is used to create an instance of AuthController and automatically inject the mocked dependencies. The init() method, annotated with @BeforeEach, initializes these mocks before each test. Then it asserts that the status code is 200, indicating that the request was successful. If this test passes, it means that the googleAuthentication method is correctly authenticating users with Google.
2. **JwtTokenFilterTest (TC2):** Unit test for the doFilterInternal method in the JwtTokenFilter class. It first sets up the mocks to return specific values when their methods are called with the appropriate arguments. The @InjectMocks annotation is used to create an instance of JwtTokenFilter and automatically inject the mocked dependencies(jwtTokenProvider). It calls the doFilterInternal method which is exposed publicly as doFilterPublic for testing purposes with the request, response, and filterChain objects. Asserts verifies that the validateToken and getAuthentication methods of jwtTokenProvider were called with the correct arguments. Finally, it asserts that the Authentication object in the Security ontextHolder is the same as the Authentication object returned by jwtToken Provider.getAuthentication. If this test passes, it means that the doFilterInternal method is correctly validating the JWT token, getting the Authentication object, and setting it in the SecurityContextHolder.
3. **JwtTokenProviderTest (TC3):** This is the test for the JwtTokenProvider class. It contains four test methods that test the createToken, validateToken, getUsername, and getAuthentication methods of JwtTokenProvider. testCreateToken checks if createToken returns a non-null token. testValidateToken ensures validateToken correctly validates a token. testGetUsername validates getUsername extracts the username from a token. testFetAuthentication checks if getAuthentication creates an Authentication object from a token. @BeforeEach annotation is used to initialize the mocks and set the jwtSecretKey and jwtExpirationInMs fields of jwtTokenProvider using ReflectionTestUtils.setFiled. If all these tests pass, it means that the JwtTokenProvider class is correctly creating, validating, and extracting information from JWT tokens.
4. **SecurityConfigTest (TC4):** This is a test class for the SecurityConfig class of the application. The @SpringBootTest annotation is used to indicate that this is a test for the entire Spring application context. The @AutoConfigureMockMvc annotation is used to

automatically configure MockMvc. The @Autowired annotation is used to inject an instance of MockMvc into the test class. The test first checks an unauthenticated endpoint (/public/test) by sending a GET request using mockMvc.perform(get("/public/test")) and ensuring the response status is 200(OK) with .andExpect(Status().isOK()). Then, it tests an authenticated endpoint (/api/protected/test) by sending a GET request using mockMvc.perform(get("/api/protected/test")) and ensuring the response is 401(Unauthorized) with .andExpect(status().isUnauthorized()). Passing this test confirms that the security configuration properly allows access to public endpoints.

5. **TokenDtoTest (TC5):** This is a test class for the TokenDto class. It contains two test methods that test the getToken and setToken methods of the TokenDto class. testGetToken checks that the getToken method returns the current token by asserting that the returned token is the same as the original token. testSetToken checks that the setToken method correctly sets the token by asserting that the returned token is the same as the original token. If both these tests pass, it means that the getToken and setToken methods of TokenDto are working correctly.
6. **UserControllerTest (TC6):** This is a test class for the UserController class. It contains three test methods that test the getAllUsers, Userslogin, and createUsers methods of the UserController class. testGetAllUsers checks if getAllUsers returns the correct user list. Sets up a User object, mocks userRepository.findAll() to return a list with this user, then asserts that the returned list from getAllUsers contains the expected user. testUserLogin validates UserLogin's authentication and token generation. Sets up a User object and request body, mocks userRepository.findByUsername to return the user. Calls UserLogin with the request body and asserts the response status code and body. testCreateUsers ensures createUsers correctly creates a user. Sets up a User object and request body, mocks userRepository.findByUsername to return null. Calls createUsers with the request body and asserts the response status code and body. Passing these tests confirms the correctness of getAllUsers, UserLogin, and createUsers methods in UserController class.
7. **UserRepositoryTest (TC7):** This is a test class for the UserRepository class, which is a MongoDB repository for User objects. It contains two test methods that test the findByEmail and findByUsername methods of UserRepository. testFindByEmail checks that the findByEmail method returns the correct user. It calls findByEmail with the email of the user and asserts that the returned user is not null and has the correct username. testFindByUsername checks that the findByUsername method returns the correct user. It calls the findByUsername with the username of the user and asserts that the returned user is not null ,and has the correct email. The @DataMongoTest annotation is used to indicate that this is a test for MongoDB repositories. If both these tests pass, it means that the findByEmail and findByUsername methods of UserRepository are working correctly.

8. **UserSessionDtoTest (TC8):** This is a test class for the UserSessionDto class. It contains four test methods that test the getEmail, getName, getId, and getSessionToken. testEmail checks that the getEmail method returns the correct email. It first sets up an expected email, then calls getEmail and finally asserts that the returned email is the same as the expected email. testName checks that the getName method returns the correct name. testId checks that the getId and setId methods work correctly. testSessionToken checks that the getSessionToken and setSessionToken methods work correctly. If all these tests pass, it means that the getEmail, getName, getId, getSessionToken, setId, and setSessionToken are working correctly.
9. **UserTest (TC9):** This is a test class for the User class. It contains four test methods that test the getId, getUsername, getEmail, and getPassword methods, as well as the setId, setUsername, setEmail, and setPassword methods of User. testId checks that the getId and setId methods work correctly. testUsername checks that the getUsername and setUsername methods work correctly. TestEmail checks that the getEmail and setEmail methods work correctly. testPassword checks that the getPassword and setPassword methods work correctly.
10. **WebConfigTest (TC10):** The @SpringBootTest annotation is used to indicate that this is a test for the entire Spring application context. The @AutoConfigureMockMvc annotation is used to automatically configure MockMvc. The @Autowired annotation is used to inject an instance of MockMvc into the test class. The testCorsConfig method tests that the application correctly sets the Access-Control-Allow-Origin header in response to a request from http://localhost:3000. If this test passes, it means that the CORS configuration is correctly allowing requests from http://localhost:3000.

ExpenseController Tests:

1. **getAllExpensesTest (TC11):** This test is used to verify that the endpoint "/api/expenses" returns all expenses successfully with the expected format and content. The test case passes if all assertions are successful. The getAllExpenses endpoint returns a 200 OK response with a JSON array containing two expenses, and the amount of the second expense is 10.50. This test case assumes that the behavior of the ExpenseService.getAllExpenses() method and the structure of the /api/expenses endpoint remain unchanged.
2. **getExpenseById_successTest (TC12):** This test is used to verify that the endpoint /api/expenses/{id} returns the correct expense object when provided with a valid expense ID. The test case passes if all assertions are successful. The getExpenseById endpoint returns a 200 OK response with a JSON object representing the expense with ID E1, and the id attribute of the response matches the expected value.

3. **getExpenseById_NotFoundTest (TC13):** This test is used to verify that the endpoint /api/expenses/{id} returns a 404 Not Found error when provided with an invalid or non-existent expense ID. The test case passes if the HTTP response status is 404 Not Found, indicating that the expense with the provided ID does not exist.
4. **addExpenseTest (TC14):** This test is used to verify that a new expense can be successfully created via the endpoint /api/expenses with the provided data. The test case passes if all assertions are successful. The createExpense endpoint returns a 200 OK response with a JSON object representing the created expense, and the amount attribute of the response matches the expected value.
5. **updateExpenseTest (TC15):** This test is used to verify that an existing expense record can be successfully updated via the endpoint /api/expenses/update/{id} with the provided data. The test case passes if all assertions are successful. The updateExpenseRecord endpoint returns a 200 OK response with a JSON object representing the updated expense record, and the amount attribute of the response matches the expected updated value.
6. **deleteExpense_successTest (TC16):** This test is used to verify that an existing expense record can be successfully deleted via the endpoint /api/expenses/remove/{id}. The test case passes if the HTTP response status is 200 OK, indicating that the expense record with the provided ID has been successfully deleted from the system.
7. **deleteExpenseId_NotFoundTest (TC17):** This test is used To verify that an existing expense record cannot be deleted if the provided expense ID does not exist. The test case passes if the HTTP response status is 404 Not Found, indicating that the expense record with the provided ID does not exist and cannot be deleted.
8. **getExpenseByCategoryTest (TC18):** This test is used to verify that the endpoint /api/expenses/category/{categoryId} returns the correct expense object when provided with a valid expense ID. The test case passes if all assertions are successful. The getExpenseByCategory endpoint returns a 200 OK response with a JSON object representing the expense with ID E1, and the id attribute of the response matches the expected value.

Miscellaneous

AI Citation

During the research stage, leveraging ChatGPT proved invaluable for informing our UI/UX exploration for the application. By tapping into its vast resources and insights, we gained a multifaceted understanding of how users in the financial planning sphere interact with different application styles. This comprehensive approach not only empowered us to craft a visually appealing application tailored to our target audience but also ensured that our design choices were rooted in proven success within the domain. Moreover, ChatGPT played a pivotal role in brainstorming an original application name, steering clear of existing large companies, and providing invaluable resources on features that would resonate with our users.

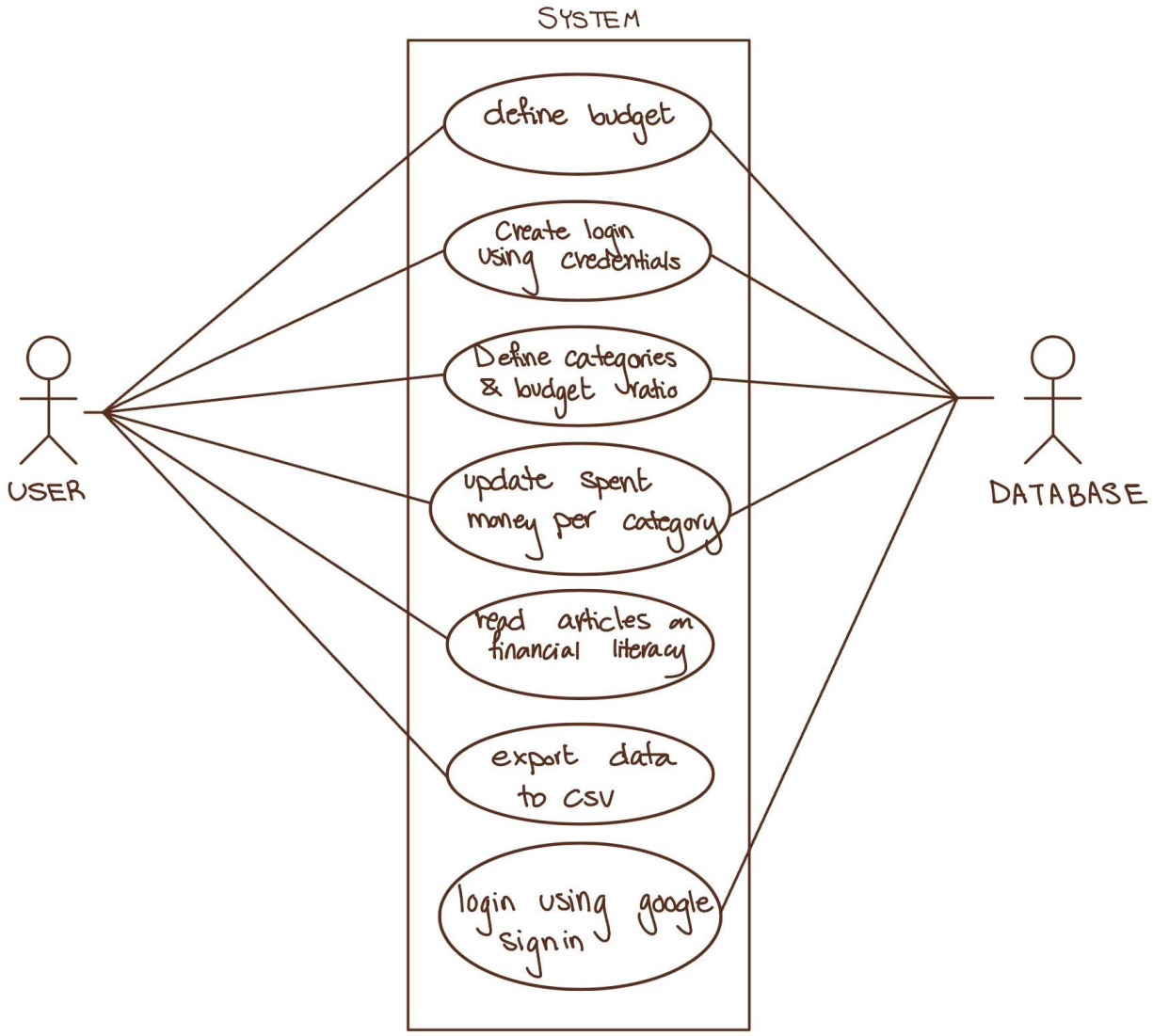
By consulting ChatGPT as a third-party resource, we consciously mitigated potential biases inherent in our team's composition of seven software engineering students ensuring that our application catered directly to user needs without undue influence from our group's shared perspectives.

In the frontend development phase, ChatGPT continued to be a valuable asset, offering insights to refine code and suggesting more efficient implementation strategies for various functions. Notably, it aided in identifying and rectifying code smells such as redundant code, empty functions, and unused variables. These optimizations not only enhance the application's performance, making it run faster, but also lay a foundation for seamless collaboration with potential future contributors, as the logic is streamlined and easily navigable.

User stories

1. As a new user, I want to register using my email address so that I can access personalized information off the app.
2. As a returning user, I want to log in with my existing credentials so that I can access my account quickly.
3. As a budget-conscious user, I want to set monthly spending limits so that I can manage my finances better.
4. As a detail-oriented user, I want to categorize my transactions so that I can see where I'm spending my money.
5. As a visual user, I want to see my expenses represented in charts and graphs so that I can easily analyze my financial habits.
6. As a planning user, I want to create financial goals for savings so that I can work towards personal milestones.
7. As a security-conscious user, I want to ensure a secure and private sign-in/sign-up process so that I can ensure my information is protected and persistent.
8. As a growing user, I want to access articles on financial education so that I can make smarter money decisions.
9. As a collaborative user, I want to share my budget plans with my pet so that we can manage our finances together.
10. As a forward-thinking user, I want to forecast my future spending based on past trends so that I can reallocate my budget effectively.
11. As a digital user, I want to download my financial data in CSV format so that I can do further analysis.
12. As a responsible user, I want to track my debt repayment progress so that I can be debt-free and escape the matrix.
13. As an organized user, I want to tag transactions with custom labels so that I can find them easily later.

Use Cases


SYSTEM
define budget
Create login using credentials
Define categories & budget ratio
update spent money per category
read articles on financial literacy
export data to csv
login using google signin
USER
DATABASE

Changes in development process

During the evolution of WealthWave, our team encountered numerous challenges and invaluable learning experiences. Initially guided by a clear vision and the Figma design, we embarked on this journey. However, as we delved deeper into development, we uncovered various aspects of our team dynamics. While we initially adhered to a frontend/backend split, it became evident that this division was not absolute. Certain team members found themselves navigating both realms, while others focused on different project aspects. These adjustments naturally arose, emphasizing the importance of flexibility over rigidity in team structures.

Additionally, our ambition to create an app that exceeded basic requirements aligned with the essence of WealthWave. However, practical constraints, particularly time limitations and the complexity of app development, compelled us to recalibrate our approach. Furthermore, feedback from our TA highlighted deficiencies in user interface (UI) friendliness, prompting a comprehensive overhaul. This redesign served as a valuable lesson, reinforcing the significance of simplicity in UI design to enhance user experience.

Despite these challenges, we remained steadfast in our commitment to the initially planned features. Our meticulous design phase enabled us to execute our plan efficiently, minimizing redevelopment delays stemming from inadequate preparation. Through these iterative adjustments, we witnessed collective growth in technical expertise and project management acumen. WealthWave exemplifies our adaptability and collaborative spirit, tracing our journey from conceptualization (Figma) to realization (WealthWave) amidst diverse challenges.

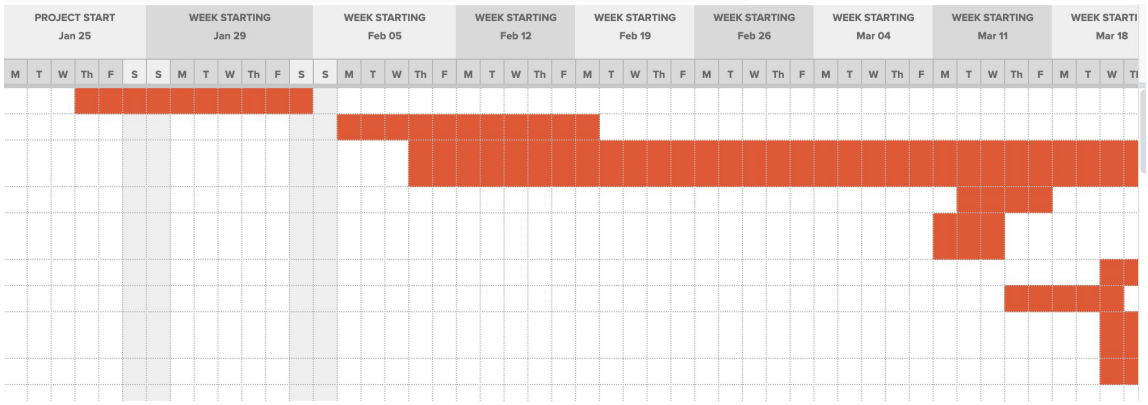
Teamwork process

We believe that our unanimous approach for this application, as well as early planning and consistent communication set us up for success. Our team used a Discord channel, which maintained an easygoing culture as the primary method of communication in regards to integration concerns, deadline reminders, and confirmation of ideas between developers. This allowed developers to check in with one another without worrying that no one was going to respond. This stream of communication also made it easy to host simple calls to talk more about the project development or check in. Given that we had discussed a rough timeline within our team, it was relatively simple to see if developers were on task, and if we needed to assist in any way. To organize our teamwork and task dependency, we used a Gantt Chart template sourced from the website below.

https://www.google.com/url?q=https://www.vertex42.com/ExcelTemplates/excel-project-management.html&sa=D&source=docs&ust=1711607745903212&usg=AOvVaw1Bzuqb49vjca_h2Krc4hpA

WEALTHWAVE	Start Date
Development Schedule for	1/25/2024
Course Project	Project Duration
prepared for SENG 401 Team 2	101 Days

TASK NAME	START DATE	END DATE	DURATION (WORK DAYS)
Project Approval	1/25	2/3	9
Backend Authentication	2/3	2/14	12
Expenses, Budget and Education Pages	2/8	3/22	45
Backend Table Setup	3/12	3/15	4
Backend/Frontend Testing	3/11	3/13	3
Remodel UI	3/20	3/23	4
Sectional Testing	3/14	3/20	7
Integration + Integration Testing	3/20	3/25	6
System Documentation	3/20	3/26	7
Final Feature Addition	3/24	3/26	3
Walk-thru + Submission	3/27	3/27	1



The complete Gantt Chart can be viewed at the link below:

https://docs.google.com/spreadsheets/d/1uBovwYBLshMtSvzezh5Hoqk6sc_1zikqFWAetI8D8_M/edit?usp=sharing

Dependencies

Format : (task) - (what the task depends on)

Backend/ Frontend Development - Depends on project approval.
Backend Authentication - No dependency.
Expenses/Education/Budget pages - Depends on project approval.
Backend Tables - Depends on project approval and team discussion.
Backend/Frontend Testing - Depends on the unit testing of individual functions and pages locally within branches.
Remodeling UI - Depends on the completion and confirmation of frontend pages against all functional requirements.
Sectional Testing - Depends on the partial completion of merging of branches into the main branch on Github.
Integration / Integration Testing - Depends on the completion of sectional testing, and setting up backend application and frontend application on various machines.
System Documentation - Depends on a completion of unified understanding of the system between team members, as well as the completion of sectional testing. Final Feature Addition - Depends on the starting of integration testing.
Walk-through and Submission - Depends on the completion of final feature addition, and depends on the completion of all previous tasks.