**Software Architecture and Development: Finance Planner**

**Overview**

This document provides a summary of the project architecture and CRUD operations implemented across various microservices for the shared expense management system.

**Table of Contents:**

1. **Introduction**
2. **Tech Stack Overview**
3. **Microservices Architecture**
4. **CRUD Operations Summary**
5. **Additional Features**
6. **Future Scope**
7. **Conclusion**
8. **Steps to set up**
9. **Tech Stack:**

* **Backend: FastAPI**
* **Frontend: React**
* **Database: PostgreSQL**
* **Automation: GitHub-Actions**
* **Cloud-Storage: Elastic-Cloud**
* **Deployment: Docker**
* **Data-Management: GCP**

1. **Introduction**

The Finance Planner is a microservices-architecture based application designed to simplify expense management, budgeting, and shared financial tracking, leveraging modern technologies for seamless user experience and robust performance.

1. **Microservices Breakdown**

The backend follows a microservices-based approach with different services handling specific functionalities.

**1. AuthService**

* Manages user authentication and authorization.
* Implements JWT-based authentication.
* Supports user registration, login, and token verification.
* Implements rate limiting for API requests.

**CRUD Operations:**

* Create User (register\_user)
* Read User Details (get\_user)
* Update User (update\_user)
* Delete User (delete\_user)

**2. UserService**

* Handles user profile management.
* Provides endpoints for retrieving and updating user data.

**CRUD Operations:**

* Get All Users (get\_all\_users)
* Get User by ID (get\_user)
* Update User (update\_user)
* Delete User (delete\_user)

**3. BudgetingService**

* Manages user budgets and categories.
* Provides insights into expenses based on predefined categories.
* Fetches transaction data from TransactionService.

**CRUD Operations:**

* Create Budget (create\_budget)
* Read Budgets (get\_all\_budgets, get\_budget\_by\_id, get\_all\_budgets\_by\_user\_id, get\_all\_budgets\_by\_user\_id\_and\_date)
* Update Budget (update\_budget)
* Delete Budget (delete\_budget)

**4. TransactionAnalysisService**

* Handles transaction-related functionalities.
* Supports analytics on user transactions, spending patterns, and category-based insights.

**CRUD Operations:**

* Fetch Transactions (get\_last\_10\_transactions, get\_last\_week\_transactions)
* Fetch Total Expenses by Category (get\_expenses\_by\_category)
* Predict Savings (predict\_savings)
* Categorize Transactions (categorize\_transactions)

**5. SharedExpensesService**

* Manages shared expense groups and participant transactions.
* Handles splitting transactions among participants.

**CRUD Operations:**

* Create Shared Group (create\_shared\_group)
* Read Shared Groups (get\_shared\_groups, get\_shared\_groups\_by\_participant\_id)
* Update Shared Group (update\_shared\_group)
* Delete Shared Group (delete\_shared\_group)
* Manage Participants (create\_shared\_group\_participant, delete\_shared\_group\_participant, get\_shared\_group\_participants\_by\_group\_id)
* Handle Shared Transactions (create\_shared\_transaction, update\_repayment\_transaction, get\_transaction\_by\_group\_user\_id, get\_transactions\_with\_names)

1. **Common Utilities**

* Provides shared functionality such as logging and authentication helpers

1. **Additional Features**

* **Security:** Implements JWT-based authentication and role-based access control.
* **Logging:** Centralized logging system for tracking API calls and debugging.
* **Rate Limiting:** Prevents API abuse with rate limiting on all operations.
* **Access control:** added Role based access control in all the microservices
* **Docker & Deployment:**
  + Implemented docker-compose for containerized deployments.
  + Use docker-compose build to build services and docker-compose up to start them.
* **CI/CD Pipeline:**
  + Implemented GitHub Actions for automated deployment.
  + Service deployment is initiated but currently facing errors.
* **Session Management:**
  + Cookies are used for session management.
  + The authentication token is saved in cookies.
* **Monitoring & Logging:** Integrated Elastic Cloud for centralized logging and visualization using Kibana.

|  |
| --- |
|  |

1. **Future Scope**
2. **End-to-End Testing Automation**
3. **OAuth Integration**
4. **Multi-Currency Third-Party Integration**
5. **Shared Expenses Full Logic**
6. **Dynamic Transaction Categories**
7. **Active/Inactive Users**
8. **Could-Have Features (Nice to Have, but Not Critical)**
9. **Gamification Elements**
   * **Backend**:
     + Implement logic for tracking and awarding financial milestones (e.g., saving goals, achieving budgeting goals).
   * **Frontend**:
     + Design UI for displaying achievements, badges, or rewards.
   * **Integration**:
     + Sync gamification data between frontend and backend (e.g., reward achievements, display progress).
   * **Key Focus**: User motivation, engagement, progress tracking.
10. **AI-Powered Financial Insights**
    * **Backend**:
      + Implement machine learning algorithms to provide personalized financial insights based on user data.
    * **Frontend**:
      + Display actionable financial insights (e.g., "spend less on food this month").
    * **Integration**:
      + Ensure the backend provides AI-generated insights in real-time to the frontend.
    * **Key Focus**: Personalization, AI-driven recommendations, user insights.
11. **Multi-Currency Support**
    * **Backend**:
      + Implement backend logic to handle multiple currencies and exchange rates.
      + Store expenses/income in different currencies and convert them as needed.
    * **Frontend**:
      + Design UI for selecting currencies and viewing balance in different currencies.
    * **Integration**:
      + Integrate currency conversion APIs to ensure real-time exchange rate updates.
    * **Key Focus**: Internationalization, multi-currency handling, currency conversion.
12. **Conclusion**

The backend system is structured to ensure modularity and scalability. Each microservice is responsible for a specific domain, making it easy to maintain, extend, and scale as needed. Future improvements can include adding caching mechanisms, completing the deployment to cloud, testing and enhancing security measures

1. **Steps to set up:**

**Step 1: Clone the repository**

**Step 2: Navigate to the project folder**

**Step 3: Create .env files within each microservice folder, and a .env in the backend folder. Example env files have been given in all the needed folders.**

**Step 4: Create an elastic cloud account (**[**https://cloud.elastic.co/**](https://cloud.elastic.co/)**). (Reference: elastic cloud lecture notes pdf in teams)**

**Step 5: Generate public and private keys from OpenSSL and save copies of each in app/keys folder in each microservice as private\_key.pem and public\_key.pem**

**Step 6: Keep your Docker desktop open**

**Step 7: run “docker-compose up --build”**

**The application is now accessible at** [**http://localhost:5173**](http://localhost:5173)

**To access the OpenAPI (Swagger) documentation:**

**AuthService:** [**http://localhost:8001**](http://localhost:8001)

**UserLoginService:** [**http://localhost:8002**](http://localhost:8002)

**BudgetingService:** [**http://localhost:8003**](http://localhost:8003)

**SharedExpensesService:** [**http://localhost:8004**](http://localhost:8004)

**ExpenseManagementService:** [**http://localhost:8005**](http://localhost:8005)

**TransactionAnalysis:** [**http://localhost:8006**](http://localhost:8006)