Project-> BudgetBlaze (Expense tracking – Budgeting App) -> Reference – FastBudget Web App

-----------------------------------------------------------------------------

1. <https://saaspo.com/> and dribble -> We will get our UI ideas from here (reference landing pages)
2. For UI assets/pages -> Material U or ShadCn with a mix of React and Tailwind CSS.

Prompt -> take the reference from "Fast budget" web app. assume we are trying to replicate the application end to end with minor changes. Now please act as a design analyst and suggest me requirement gathering points to initiate the LLD of this. First of all I want you to give me the technical requirements then functional requirements and at last LLD for each module

Requirement gathering

1. **What is expected from the system? -> user should be able to manage and analyze the expenses using visual data**
   1. User would be able to make all the payments to his bills -> payment gateway will be integrated later on.
   2. User will get notification for pending dues or crosses threshold budget. (friendly reminders for each category – budget for each category should be set by the user.)
2. Data persistence in system -> we will be using postgresSQL for both structured and unstructured data.
3. How the system will interact with the user -> GUI (designed with UI/UX)

**Technical Requirements**

**--------------------------------------------------------------------**

* **Technology Stack**:
  + **Frontend**:
    - Framework: React.js (for dynamic and responsive UI)
    - CSS/Styling: TailwindCSS, Material UI for styling
    - State Management: Context API -> (Redux optional)
    - Authentication: JWT (JSON Web Token) for secure login
  + **Backend**:
    - Framework: Spring Boot (Java) for building REST APIs
    - Database: PostgreSQL (relational database)
    - Authentication: OAuth 2.0(optional) or JWT for user authentication
    - Data Processing: Java for backend (Spring Boot)
    - Security: Spring Security (base) with JWT for user authorization
  + **API**:
    - RESTful API for communication between frontend and backend
    - Optional: WebSockets for real-time notifications
  + **Hosting/Cloud**:
    - Deployment on platforms like AWS or Heroku
    - Use CI/CD pipelines for deployment (GitHub)
  + **Logging & Monitoring**:
    - ELK Stack (Elasticsearch, Logstash, Kibana) or third-party services like Loggly for monitoring and logging
  + **Version Control**:
    - Git for source code management and GitHub for repositories

**Functional Requirements**

**-----------------------------------------------------------------**

These define the core functionality of the application:

* **User Management**:
  + **Sign Up**: User registration with email verification.
  + **Login**: Secure login via username/email and password or social login (e.g., Google, Facebook).
  + **Password Recovery**: Password reset functionality via email.
  + **Profile Management**: Ability to update personal details and preferences.
* **Budget Management**:
  + **Create Budget**: Users can set a monthly budget for different categories (e.g., Food, Transportation, Entertainment).
  + **Edit/Delete Budget**: Users can modify or remove a budget.
  + **Budget Overview**: Visual representation of the budget with graphs/charts showing budget vs. actual spending.
  + **Categories**: Different spending categories like Utilities, Groceries, Rent, etc.
  + **Currency Support**: Users can choose their preferred currency.
* **Expense Tracking**:
  + **Add Expense**: Users can log individual expenses with amount, category, and description.
  + **Recurring Expenses**: Ability to set recurring expenses (weekly/monthly).
  + **Edit/Delete Expenses**: Users can modify or remove logged expenses.
  + **Expense Overview**: A visual breakdown of expenses for each category and overall.
  + **Expense History**: Users can view their expense history and filter by date or category.
* **Reporting & Analytics**:
  + **Expense Summary**: Summary of spending for a given time frame (day, week, month).
  + **Expense vs. Budget**: Comparison of expenses against the set budget for each category.
  + **Trends**: Visual graphs (line, pie charts) to show spending trends over time.
  + **Financial Insights**: Notifications or insights based on user spending patterns (e.g., “You’ve exceeded your grocery budget this month”).
* **Notifications**:
  + **Alerts for Budget Overrun**: Push or email notifications when a user exceeds the allocated budget.
  + **Recurring Reminders**: Reminders for recurring expenses or budgeting tasks.
* **Data Synchronization**:
  + **Cloud Sync**: Allow users to sync their data across multiple devices.
  + **Offline Mode**: Ability to track expenses even when the user is offline, syncing later when online

### ****Low-Level Design (LLD) for Each Module****

### ****-> class diagram and schema diagram needs to be created based on the references below****

#### ****3.1. User Management Module****

* **Components**:
  + **Sign Up**:
    - Form with fields: Name, Email, Password, Confirm Password.
    - Backend API: POST /api/auth/signup
    - Input validation and user creation.
    - Email verification using a token sent to the user’s email.
  + **Login**:
    - Form with fields: Email, Password.
    - Backend API: POST /api/auth/login
    - Validate credentials and return JWT token.
  + **Profile Management**:
    - API to get user details: GET /api/user/profile
    - API to update user details: PUT /api/user/profile
* **Database Design**:
  + User table: id, email, password\_hash, name, created\_at, updated\_at

#### ****3.2. Budget Management Module****

* **Components**:
  + **Create Budget**:
    - API: POST /api/budget
    - Request body: category, budget\_amount, currency, month
  + **Edit/Delete Budget**:
    - API: PUT /api/budget/{id} (Edit), DELETE /api/budget/{id} (Delete)
  + **Budget Overview**:
    - API: GET /api/budget/overview
    - Return: Total budget, actual expenses, remaining budget, categories.
  + **Categories**:
    - Categories stored in a categories table (e.g., Food, Transport, Entertainment).
* **Database Design**:
  + budgets table: id, user\_id, category, budget\_amount, currency, month, created\_at
  + categories table: id, name, description

#### ****3.3. Expense Tracking Module****

* **Components**:
  + **Add Expense**:
    - API: POST /api/expense
    - Request body: amount, category, description, date, user\_id
  + **Recurring Expenses**:
    - API: POST /api/expense/recurring
    - Request body: amount, category, frequency, start\_date, end\_date
  + **Expense History**:
    - API: GET /api/expenses/history?start\_date={start\_date}&end\_date={end\_date}
* **Database Design**:
  + expenses table: id, user\_id, amount, category, description, date
  + recurring\_expenses table: id, user\_id, amount, category, frequency, start\_date, end\_date

#### ****3.4. Reporting & Analytics Module****

* **Components**:
  + **Expense Summary**:
    - API: GET /api/reports/summary
    - Return: Total expenses for the current month, categorized.
  + **Expense vs. Budget**:
    - API: GET /api/reports/compare
    - Return: Comparison of budget vs. actual expenses for categories.
  + **Trends**:
    - API: GET /api/reports/trends
    - Return: Data for generating visual graphs (spending over time).
* **Database Design**:
  + Reporting can be done using data aggregations (SUM, AVG) from the expenses and budgets tables.

#### ****3.5. Notifications Module****

* **Components**:
  + **Alerting**: Notify users if they exceed their budget in a category.
  + **Push/Email Notifications**:
    - API for sending notifications: POST /api/notifications/send
    - Integration with services like Firebase for push notifications.

#### ****3.6. Data Synchronization & Offline Mode****

* **Components**:
  + **Cloud Sync**: Sync user data with the cloud upon login, ensuring cross-device availability.
  + **Offline Mode**: Store user data locally (using IndexedDB or local storage) and sync it when the internet is available.

--------------------------------------------------------------------------------------------------------------------

Sure! Here's an overview of a **Microservice Architecture** for an **Expense Tracking Application** that allows users to log expenses, track them, and pay their dues. We'll split the system into multiple microservices, each responsible for a specific domain or functionality.

### 1. ****Microservices Overview****

* **User Service**: Manages user authentication, registration, and profile.
* **Expense Service**: Handles logging, categorizing, and viewing of expenses.
* **Payment Service**: Manages payment operations for dues (integration with payment gateways like PayPal, Stripe, etc.).
* **Notification Service**: Sends notifications to users about upcoming payments, expense trends, or budget limits.
* **Budget Service**: Tracks user budgets and alerts them if they’re going over their budget.
* **Reporting Service**: Generates detailed reports for users on their expenses, savings, and trends.
* **Authentication and Authorization Service**: Responsible for securing the application, verifying users, and ensuring access control.

### 2. ****Microservices Breakdown****

#### a. ****User Service****

* **Responsibilities**:
  + User registration and authentication (via JWT or OAuth).
  + Profile management (name, email, phone, etc.).
  + Password management and reset.
* **Technology**:
  + Spring Boot (Java) / Express.js (Node.js) / Django (Python).
  + Authentication using JWT (JSON Web Tokens) or OAuth 2.0.
  + Database: Relational DB like PostgreSQL or NoSQL (MongoDB).

#### b. ****Expense Service****

* **Responsibilities**:
  + Log and categorize expenses (food, entertainment, bills, etc.).
  + Track and view all logged expenses.
  + Set and track recurring expenses.
* **Technology**:
  + RESTful API for managing expenses.
  + Integration with external APIs for receipt scanning and OCR (Optional).
  + Database: PostgreSQL or MongoDB (for flexible expense categorization).

#### c. ****Payment Service****

* **Responsibilities**:
  + Integration with payment gateways (e.g., Stripe, PayPal) for paying dues.
  + Track payment status and logs.
  + Handle refunds, partial payments, or failed payments.
* **Technology**:
  + Integration with external payment providers via APIs (Stripe, PayPal, etc.).
  + Webhooks for payment status updates.
  + Database: Relational or NoSQL (to track payment history and status).

#### d. ****Notification Service****

* **Responsibilities**:
  + Send notifications (SMS, Email, Push Notifications) for due payments or alerts.
  + Provide reminders for budget limits or overspending.
* **Technology**:
  + Push Notification with services like Firebase or OneSignal.
  + Email notifications using services like SendGrid or Amazon SES.
  + SMS notifications using Twilio.

#### e. ****Budget Service****

* **Responsibilities**:
  + Set a monthly or weekly budget for different categories (food, entertainment, etc.).
  + Track real-time expense vs. budget.
  + Alert the user if they exceed their set budget.
* **Technology**:
  + RESTful API.
  + Database: Relational (PostgreSQL) or NoSQL (MongoDB).
  + Can use machine learning or simple heuristics for budgeting recommendations.

#### f. ****Reporting Service****

* **Responsibilities**:
  + Provide analytics on user spending patterns.
  + Generate and send periodic reports (weekly, monthly, yearly).
  + Visualize trends (graphs, charts) on user spending.
* **Technology**:
  + Reporting tools like Apache Kafka, ELK stack, or custom report generation.
  + Use of libraries like Chart.js or D3.js for frontend visualizations.
  + Data storage: NoSQL or Data Warehouses for analytics.

#### g. ****Authentication and Authorization Service****

* **Responsibilities**:
  + Authenticate users securely via tokens (JWT or OAuth).
  + Role-based access control (admin, user, etc.).
  + Handle session management and token expiration.
* **Technology**:
  + OAuth 2.0 / JWT / OpenID Connect.
  + Identity providers (Keycloak, Auth0, or Firebase Authentication).
  + Encryption: AES, RSA, etc.

Sure! Here's an overview of a **Microservice Architecture** for an **Expense Tracking Application** that allows users to log expenses, track them, and pay their dues. We'll split the system into multiple microservices, each responsible for a specific domain or functionality.

### 3. ****Communication Between Microservices****

* **Synchronous Communication**: Use REST APIs or gRPC for service-to-service communication.
  + Example: The **Expense Service** will call the **Payment Service** to request a payment for dues.
* **Asynchronous Communication**: Use a message broker like **Kafka** or **RabbitMQ** for decoupling services.
  + Example: The **Notification Service** listens to the message queue for updates about payments or expenses and sends notifications.
* **API Gateway**: An **API Gateway** (e.g., **Kong**, **Zuul**, or **AWS API Gateway**) routes all incoming requests to the appropriate services.
  + Handles load balancing, authentication, logging, etc.
* \*\*

To design a Microservice Architecture for an **Expense Tracking Application**, we can break down the core functionalities into individual, loosely-coupled services. Here's a high-level breakdown of the architecture and the various components you might need:

**High-Level Microservices for Expense Tracking Application**

1. **User Service**:
   * Responsible for managing user accounts, authentication, and authorization (Login, Registration, Profile Management).
   * API Endpoints:
     + POST /users/register: Register a new user.
     + POST /users/login: User login, JWT token generation.
     + GET /users/{userId}: Fetch user details.
2. **Expense Service**:
   * Manages user expenses including logging and categorizing expenses.
   * API Endpoints:
     + POST /expenses: Add a new expense (amount, category, description, etc.).
     + GET /expenses/{userId}: Fetch a list of expenses for a user.
     + GET /expenses/{userId}/{expenseId}: Get detailed information on a specific expense.
     + PUT /expenses/{expenseId}: Edit a specific expense.
     + DELETE /expenses/{expenseId}: Delete an expense.
3. **Category Service**:
   * Manages categories for expenses (e.g., food, transportation, rent, etc.).
   * API Endpoints:
     + GET /categories: Get all available categories.
     + POST /categories: Add a new category.
     + PUT /categories/{categoryId}: Edit an existing category.
     + DELETE /categories/{categoryId}: Delete a category.
4. **Payment Service**:
   * Manages user payments and dues. This service will handle processing payments for expenses or dues and track payments made by the user.
   * API Endpoints:
     + POST /payments: Create a new payment (amount, user ID, expense ID, etc.).
     + GET /payments/{userId}: Get all payments made by the user.
     + POST /payments/{paymentId}/process: Process a payment (Integration with third-party payment systems like Stripe, PayPal, etc.).
5. **Due Service**:
   * Manages dues between users (e.g., if a user owes money for shared expenses). This service will allow the tracking of dues and calculate amounts due to or from users.
   * API Endpoints:
     + POST /dues: Create a new due between users.
     + GET /dues/{userId}: List all dues for a user.
     + PUT /dues/{dueId}: Mark a due as paid.
     + DELETE /dues/{dueId}: Delete a due.
6. **Notification Service**:
   * Sends notifications (emails/SMS/push) to users about their dues, payments, and expenses.
   * API Endpoints:
     + POST /notifications: Create a notification for the user.
     + GET /notifications/{userId}: Get all notifications for a user.
7. **Analytics Service**:
   * Provides data and reports on expenses, payments, categories, and dues. This service could be used to generate charts or reports on the user’s spending habits.
   * API Endpoints:
     + GET /analytics/{userId}/overview: Provides a summary (expenses, payments, and dues).
     + GET /analytics/{userId}/category-stats: Provides statistics by category (e.g., how much the user has spent on food).
     + GET /analytics/{userId}/trends: Shows spending trends over time.
8. **Auth Service** (optional if you want to manage user sessions separately):
   * Handles user authentication via OAuth, JWT, etc.
   * Integrates with the User Service to authenticate and authorize users.

**Key Components in the Architecture**

1. **API Gateway**:
   * An API Gateway (like Kong, Nginx, or Zuul) sits at the entrance of the microservices to aggregate and route client requests to the appropriate microservices.
   * Can also handle tasks like rate-limiting, authentication, and logging.
2. **Service Discovery**:
   * A service discovery tool (like Consul or Eureka) enables microservices to dynamically find and communicate with each other.
3. **Database**:
   * Each service can have its own database (polyglot persistence) based on the requirements. For instance:
     + **User Service**: Relational DB like PostgreSQL for user info.
     + **Expense & Due Service**: NoSQL DB (MongoDB) or Relational DB.
     + **Payment Service**: External payment gateway for processing payments, but could have a database to store transaction records.
4. **Message Queue**:
   * Services can communicate asynchronously through a message queue like **Kafka** or **RabbitMQ** for operations like sending notifications, processing payments, or managing dues updates.
5. **External Integrations**:
   * **Payment Gateway Integration**: Third-party service like **Stripe** or **PayPal** for processing payments.
   * **Notification Providers**: Integration with third-party services like **SendGrid** or **Twilio** for sending emails or SMS notifications.

**Microservice Communication:**

* **Synchronous Communication** (HTTP/REST):
  + User Service communicates with the Expense Service, Payment Service, and Due Service for CRUD operations.
  + The API Gateway acts as a reverse proxy and routes the requests to the corresponding service.
* **Asynchronous Communication** (Message Queue):
  + The Payment Service can use a message queue to notify the Due Service when a payment is completed, allowing the user’s dues to be updated.
  + The Notification Service can listen to specific events (such as a due being paid) and send notifications to the user.

**Technologies You Might Use:**

1. **Backend Frameworks**:
   * Spring Boot (Java) or Express.js (Node.js) for building RESTful services.
   * Flask (Python) for lightweight services.
2. **Databases**:
   * PostgreSQL/MySQL for relational data (e.g., user information, transactions).
   * MongoDB for NoSQL (expenses, dues, etc.).
3. **Authentication**:
   * JWT for stateless authentication.
   * OAuth2 for user login integration (if needed).
4. **Message Queues**:
   * Kafka or Rabbit MQ for asynchronous communication.
5. **API Gateway**:
   * Kong, Nginx, or Zuul for API routing and management.

**Flow Example:**

1. **User Registration**:
   * A new user registers through the User Service. This service handles the registration logic and stores user details in the database.
2. **Expense Logging**:
   * The user logs an expense through the Expense Service, categorizing it (by querying the Category Service) and storing the details in the database.
3. **Due Creation**:
   * If there’s a due with another user (e.g., shared expense), the Due Service is invoked to create and track that due.
4. **Payment**:
   * The user makes a payment through the Payment Service, which interacts with a payment gateway (like Stripe) to process the payment.
5. **Notification**:
   * Upon completion of payment, the Notification Service sends a push notification or email to the user about the transaction's success.
6. **Analytics**:
   * The user can see an overview of their expenses and payments using the Analytics Service.

This micro services architecture can scale independently, with each service focusing on a single responsibility. It’s flexible and can evolve over time to handle growing user demand or additional features.

Backend

Service Architecture: Micro Service

- Service Target: User Service, Budget Service, Expense Service

Project Architecture: MVC

-Controller (End-point for the upcoming Requests)

-Service (Services the functionality initiated by controller)

-Dao (repos) [DB operations]

-Model (Entity)

-Configuration (Config. stuff)

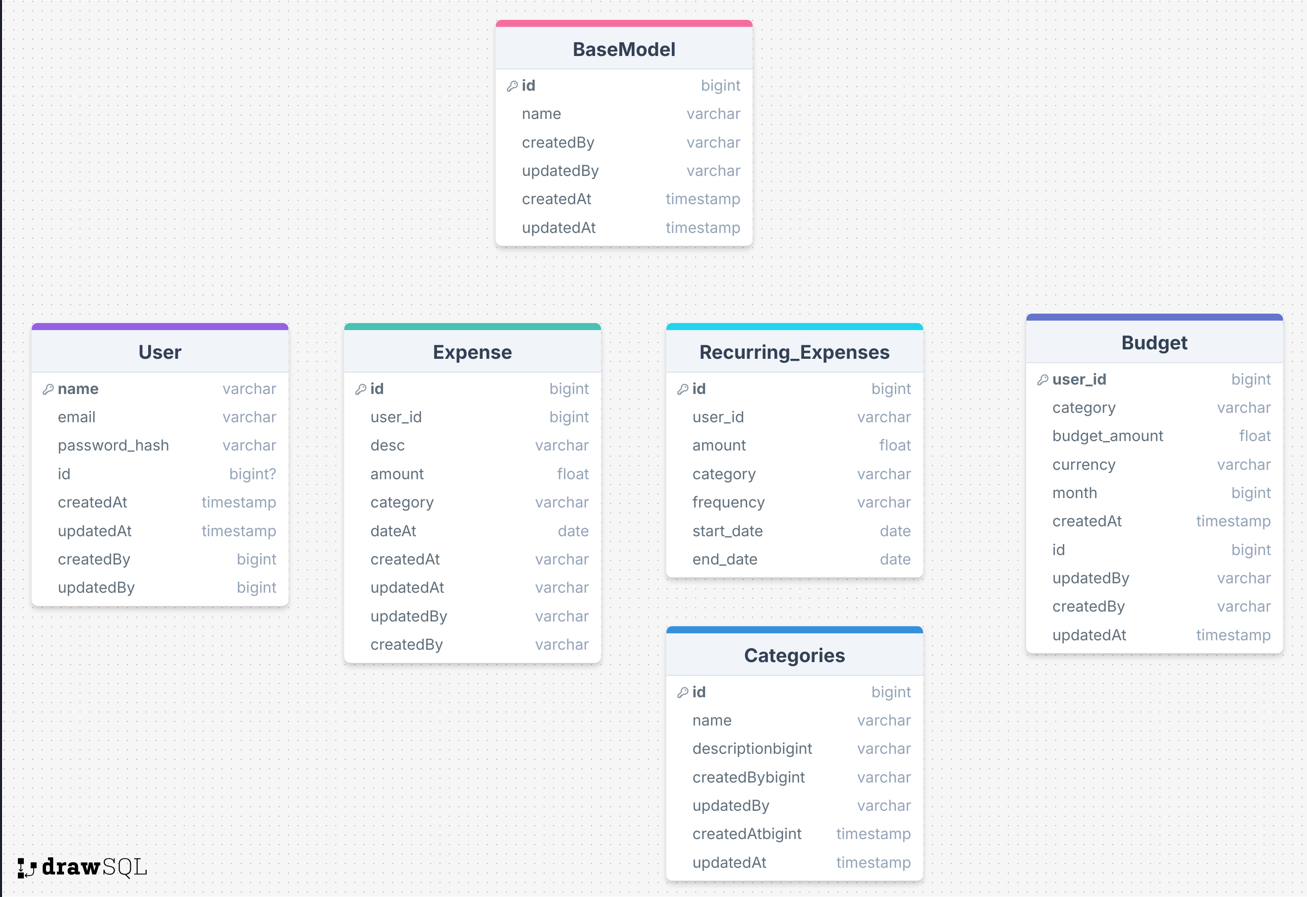
-Exceptions (Exceptions)

--- Application Work Flow

[User /Client] <------🡪 [Router] -------🡪 [Service / Proj] <-------🡪 [ [Controller] <------>

[Service] <------> [Dao]🡨----> [Model]]

Schema Design



---------------------------------------------------------------------------------------------------------------------

Line in Items -> Class Diagrams, Schema Diagrams and design patterns.

Note -> For class diagrams/schema diagrams (lets start with designing project structure) -> We will keep updating the project on github regularly.

------------------------------------------------------------------------------------------------------------------>

---User Service : class Diagrams, Schema Design done.

--IN Line-

-Process Map: Sign Up /Reg, Login