Personal Finance Tracker

(Technical Documentation)

Project Overview

The Personal Finance Tracker is a full-stack web application designed to help users manage their finances effectively. It allows users to track income and expenses, set monthly budgets, and view analytics on their spending habits. The application features a secure RESTful API backend built with Spring Boot and a responsive, modern frontend built with React.

Core Features

- **User Authentication:** Secure user registration and login using JWT (JSON Web Tokens).
- **Transaction Management:** CRUD operations for income and expense transactions.
- **Budgeting:** Ability for users to set monthly spending limits for various categories.
- **Recurring Expenses:** Manage and track recurring bills and subscriptions.
- **Financial Analytics:** A dashboard with charts visualizing monthly income vs. expenses and spending breakdowns.
- **Expense Forecasting:** A simple forecast of upcoming expenses based on recurring items.

Technology Stack

Layer	Technology & Libraries
Frontend	React 18, Vite, TypeScript, Tailwind CSS, React Router, Redux Toolkit, Axios, Recharts
Backend	Java 17, Spring Boot 3, Spring Security 6, Spring Data JPA, PostgreSQL, JWT, BCrypt
Database	PostgreSQL
DevOps	Docker, Nginx, Render (Backend Hosting), Vercel (Frontend Hosting)
Tools	Git, GitHub, Postman, Maven

System Architecture

The application follows a decoupled, **client-server architecture**.

- **Frontend (Client):** A React Single-Page Application (SPA) that runs in the user's browser. It is responsible for the UI and user experience. It communicates with the backend via REST API calls.
- **Backend (Server):** A Spring Boot application that exposes a set of RESTful API endpoints. It handles all business logic, data persistence, and user authentication.
- **Database:** A PostgreSQL database stores all user data, including credentials, transactions, and budgets.

Personal Finance Tracker: Backend Documentation

This document provides a detailed overview of the backend architecture, modules, and deployment process for the Personal Finance Tracker application.

Backend Technology Stack

The backend is built using a robust and modern Java-based technology stack, designed for security, scalability, and maintainability.

Technology/Library	Purpose in this Project
Java 17	The core programming language. LTS version ensures long-term support and stability.
Spring Boot 3	The primary application framework. It radically simplifies the development of stand-alone, production-grade Spring-based applications.
Spring Security 6	Provides comprehensive security services, including authentication and authorization. It is used to protect API endpoints and manage user sessions.
Spring Data JPA	Simplifies data access by providing a familiar repository-based abstraction over the Java Persistence API (JPA).
PostgreSQL	A powerful, open-source object-relational database system used for all data persistence.
Hibernate	The JPA provider used by Spring Data JPA. It maps Java objects to database tables (ORM - Object-Relational Mapping).
JWT (jjwt library)	JSON Web Tokens are used for stateless authentication. The jjwt library is used to create and validate these tokens.
BCrypt	The password-hashing algorithm implemented via Spring Security's BCryptPasswordEncoder. It securely stores user passwords.
Lombok	A Java library that automatically plugs into your editor and build tools to reduce boilerplate code (e.g., getters, setters, constructors).
Maven	A powerful project management and comprehension tool. It manages the project's build, reporting, and documentation from a central piece of information.
OpenAPI (Swagger)	Used for generating interactive API documentation, allowing developers and frontend clients to easily understand and test the REST endpoints.
Docker	The containerization platform used to package the application and its dependencies into a standardized unit for deployment.

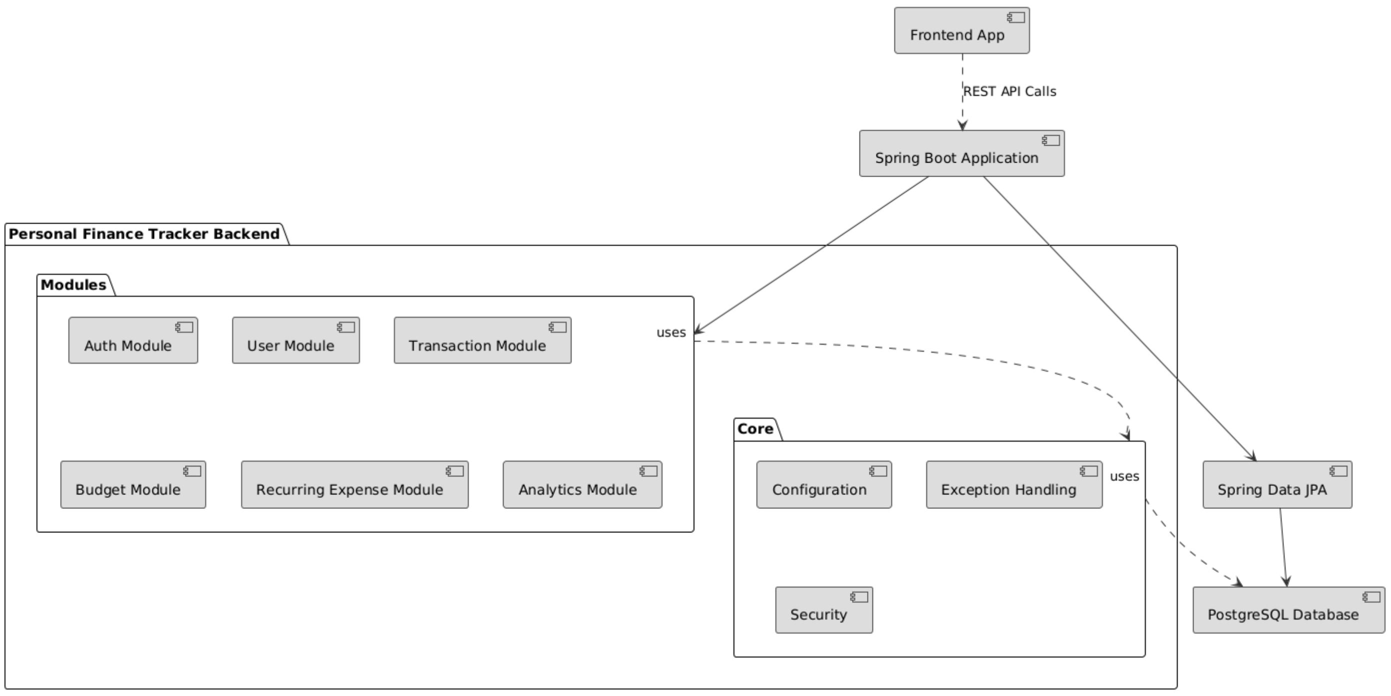
Architectural Overview

The backend follows a classic a mixture **of microservices (module-based approach) & three-tier layered architecture**, promoting a strong separation of concerns.

- **Controller Layer:** Exposes the REST API endpoints. It is responsible for handling incoming HTTP requests, validating them, and delegating business logic to the service layer.
- **Service Layer:** Contains the core business logic of the application. It orchestrates operations, interacts with the repository layer, and ensures data integrity.
- **Repository (Data Access) Layer:** Responsible for all communication with the database. Spring Data JPA provides repository interfaces that abstract away the boilerplate code for data access.

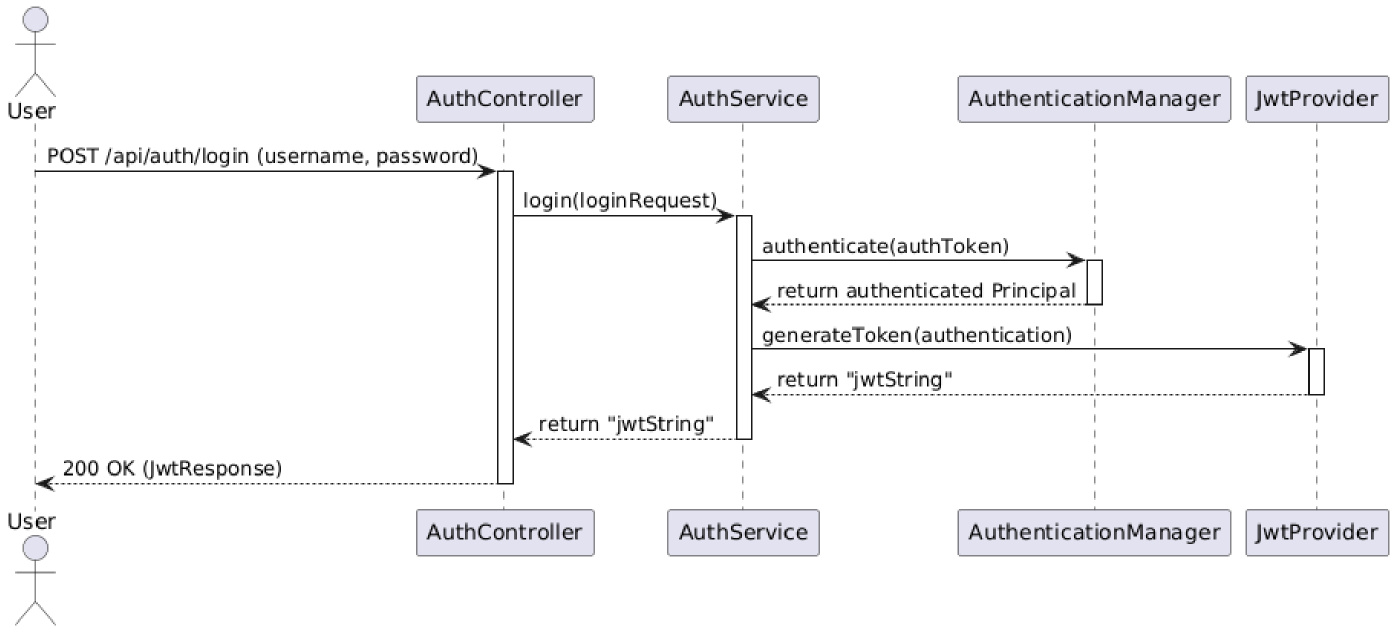
Component Diagram

This diagram shows the high-level components and their dependencies.



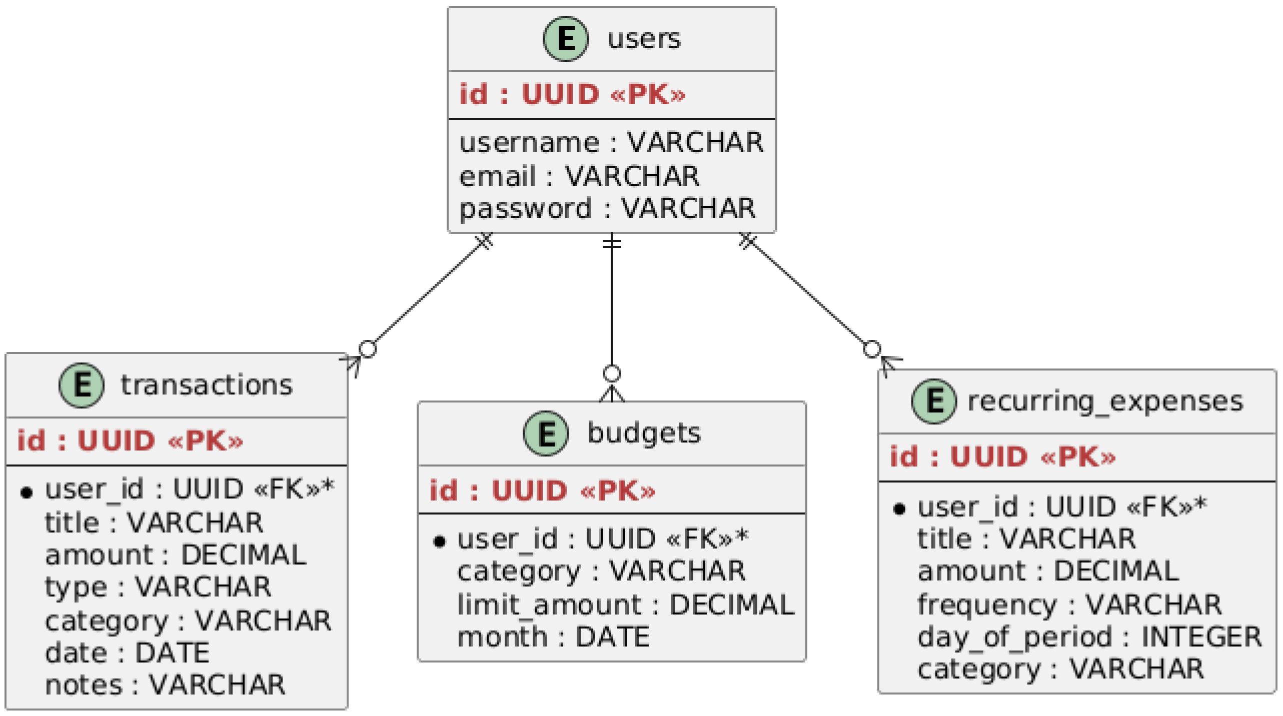
Sequence Diagram: User Login

This diagram illustrates the process of a user logging in.



Entity-Relationship Diagram (ERD)

This diagram shows the database schema and the relationships between tables.



Module Breakdown

The project is structured by features (modules) to promote separation of concerns.

```
src/main/java/com/tracker/finance/
├── core/              # Cross-cutting concerns
│   ├── config/        # Spring configurations (Security, CORS, OpenAPI)
│   ├── exception/     # Global exception handling
│   └── security/      # JWT logic, UserDetailsService
└── modules/           # Feature modules
    ├── auth/          # Authentication and User Profile
    ├── transaction/   # Transaction management
    ├── budget/        # Budget management
    ├── recurringexpense/  # Recurring Expense management
    └── analytics/     # Data aggregation for charts and forecasts
```

Core Module

This module contains cross-cutting concerns that are used by all other feature modules.

- **config**: Contains Spring configurations for Security (SecurityConfig), Cross-Origin Resource Sharing (WebConfig), and API documentation (OpenApiConfig).
- **exception**: Provides a GlobalExceptionHandler to catch exceptions and return standardized JSON error responses.
- **security**: Manages all aspects of security.
  - jwt: JwtProvider for creating/validating tokens and JwtAuthFilter to process tokens on incoming requests.
  - service: UserDetailsServiceImpl loads user-specific data for Spring Security.

Feature Modules

Auth Module

- **Purpose**: Handles user authentication (login).
- **Components**: AuthController, AuthService, LoginRequest DTO, JwtResponse DTO.
- **Flow**: A user submits credentials via the /api/auth/login endpoint. The AuthService validates them using Spring Security's AuthenticationManager and, if successful, returns a JWT.

User Module

- **Purpose**: Manages user registration, profile retrieval, updates, and deletion.
- **Components**: UserController, UserService, User entity, UserRepository, UserMapper, and various DTOs.
- **Relationships**: The User entity is the central entity in the database, linked to all other main entities (Transaction, Budget, etc.).

Transaction, Budget, and Recurring Expense Modules

- **Purpose**: These modules provide full CRUD (Create, Read, Update, Delete) functionality for their respective domains.
- **Structure**: They all follow a consistent structure with a Controller, Service, Repository, Entity, Mapper, and DTOs.
- **Security**: All endpoints in these modules are protected. The services contain authorization logic to ensure a user can only access or modify their own data.

Analytics Module

- **Purpose**: Provides aggregated data for financial insights.
- **Components**: AnalyticsController, AnalyticsService.
- **Functionality**:
  - **Monthly Summary**: Calculates total income, expenses, and budget status for a given month.
  - **Expense Forecast**: Projects upcoming expenses for the next month based on recurring expenses.
- **Relationships**: This is a read-only module that aggregates data from the Transaction, Budget, and RecurringExpense repositories.

Database Schema

The database consists of four main tables linked by user ID.

- **users**: Stores user credentials.
- **transactions**: Stores all income and expense records.
- **budgets**: Stores monthly budget limits per category.
- **recurring_expenses**: Stores recurring payment information.

API Endpoints

All endpoints are prefixed with /api. Authentication is required for all endpoints except /auth/register and /auth/login.

Endpoint	Method	Auth	Description
/auth/register	POST	✗	Registers a new user.
/auth/login	POST	✗	Authenticates a user and returns a JWT.
/auth/me	GET	✓	Gets the profile of the currently logged-in user.
/auth/profile	PUT	✓	Updates the current user's profile.
/auth/profile	DELETE	✓	Deletes the current user's account.
/transactions	GET	✓	Retrieves all transactions for the user.
/transactions	POST	✓	Adds a new transaction.
/transactions/{id}	PUT	✓	Updates an existing transaction.
/transactions/{id}	DELETE	✓	Deletes a transaction.
/budgets	GET	✓	Retrieves all budgets for the user.
/budgets	POST	✓	Adds a new budget for a specific month.
/recurring-expenses	GET	✓	Retrieves all recurring expenses for the user.
/recurring-expenses	POST	✓	Adds a new recurring expense.
/recurring-expenses/{id}	DELETE	✓	Deletes a recurring expense.
/analytics/monthly-summary?month=YYYY-MM	GET	✓	Gets a financial summary for the specified month.
/analytics/forecast	GET	✓	Gets a forecast of expenses for the next month.

Security

- **Authentication:** Handled by Spring Security using a custom JWT filter (JwtAuthFilter). A JWT is generated upon successful login and must be included as a Bearer token in the Authorization header for all protected requests.
- **Password Hashing:** Passwords are hashed using BCryptPasswordEncoder before being stored in the database.
- **CORS:** Cross-Origin Resource Sharing is enabled in WebConfig and is configurable via environment variables to only allow requests from the deployed frontend.
- **Exception Handling:** A GlobalExceptionHandler provides consistent, structured JSON error responses for different exception types.

Frontend Details

The frontend is a modern React SPA built with Vite and TypeScript.

Project Structure

The project follows a feature-sliced design.

```
src/
├── api/            # Central Axios instance configuration
├── components/     # Global, reusable UI components (e.g., Layout, Header)
├── features/       # Feature-based modules
│   ├── auth/       # Auth pages, components, and logic
│   ├── transactions/  # Transaction pages and components
│   └── ...
├── hooks/          # Custom hooks (e.g., useAppDispatch)
├── routes/         # Application routing configuration
├── store/          # Redux Toolkit store and slices
└── utils/          # Utility functions and validation schemas
```

State Management (Redux Toolkit)

Global application state is managed using Redux Toolkit.

- **Store:** The single source of truth, configured in `src/store/index.ts`.
- **Slices:** State logic is co-located by feature. Each slice (`authSlice`, `transactionSlice`, etc.) defines its initial state, reducers, and async thunks for interacting with the API.
- **Async Thunks:** Used for all asynchronous operations (API calls). They handle the pending, fulfilled, and rejected states of a request, allowing the UI to display loading indicators and error messages.

Routing

Client-side routing is handled by `react-router-dom`.

- **Public Routes:** Login and Register pages are accessible to everyone.
- **Private Routes:** All other pages (Dashboard, Transactions, etc.) are protected by the `PrivateRoute` component, which checks for a valid user session in the Redux store. If no user is authenticated, it redirects to the login page.

Future Enhancements

Feature	Description
AI Categorization	Use OpenAI to suggest categories
Notification Engine	Alert users on overspending or recurring bills
Export to Excel/PDF	Generate downloadable reports
Multi-Currency Support	Convert using real-time exchange rates
OAuth2 Google Login	Easier signup/login
Offline Mode (PWA)	Cache recent data with service workers

Deployment

Backend Deployment (Spring Boot & PostgreSQL on Render)

- Build the application with Maven
- Create the final, lightweight production image (using Docker)

```
FROM maven:3.8.5-openjdk-17 AS build
WORKDIR /app
COPY pom.xml .
RUN mvn dependency:go-offline
COPY src ./src
RUN mvn clean package -DskipTests
FROM openjdk:17-jdk-slim
WORKDIR /app
COPY --from=build /app/target/*.jar app.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "app.jar"]
```

- Prepare for Production Configuration
  Note the jdbc: prefix fix. Also make sure not to include the username and password in the URL when passing these separately.

```
spring:
  main:
    web-application-type: servlet
  datasource:
    url: ${JDBC_DATABASE_URL}
    username: ${DATABASE_USER}
    password: ${DATABASE_PASSWORD}
    driver-class-name: org.postgresql.Driver
  jpa:
    show-sql: true
    hibernate:
      ddl-auto: update
    properties:
      hibernate:
        dialect: org.hibernate.dialect.PostgreSQLDialect
app:
  jwtSecret: ${APP_JWT_SECRET}
  jwtExpirationMs: ${APP_JWT_EXPIRATION_MS:86400000}
  cors:
    allowed-origins: ${CORS_ALLOWED_ORIGINS}
server:
  port: ${PORT:8080}
logging:
  level:
    root: INFO
    org.springframework: INFO
    org.hibernate.SQL: INFO
```

- Generate a Secure JWT Secret
  The backend requires a secret key of at least 256 bits.

  On Windows (using PowerShell):
  1. Open PowerShell.
  2. Run the following commands one by one:
     ```
     $bytes = New-Object Byte[] 32
     $rng = [System.Security.Cryptography.RandomNumberGenerator]::Create()
     $rng.GetBytes($bytes)
     [System.Convert]::ToBase64String($bytes)
     ```

- Set Up Services on Render:

Frontend Deployment

1. Push finance-tracker-react project to its own GitHub repository.
2. On a Vercel, create a new project and link this repository.
3. In the project settings on Vercel, add an environment variable:
   - **Key:** VITE_API_URL
   - **Value:** The URL of your deployed Render.
4. Create a file named **vercel.json** in the root of frontend directory and include the following for proper routing:
   ```
   {
     "rewrites": [{ "source": "/(.*)", "destination": "/index.html" }]
   }
   ```
5. Deploy the frontend.