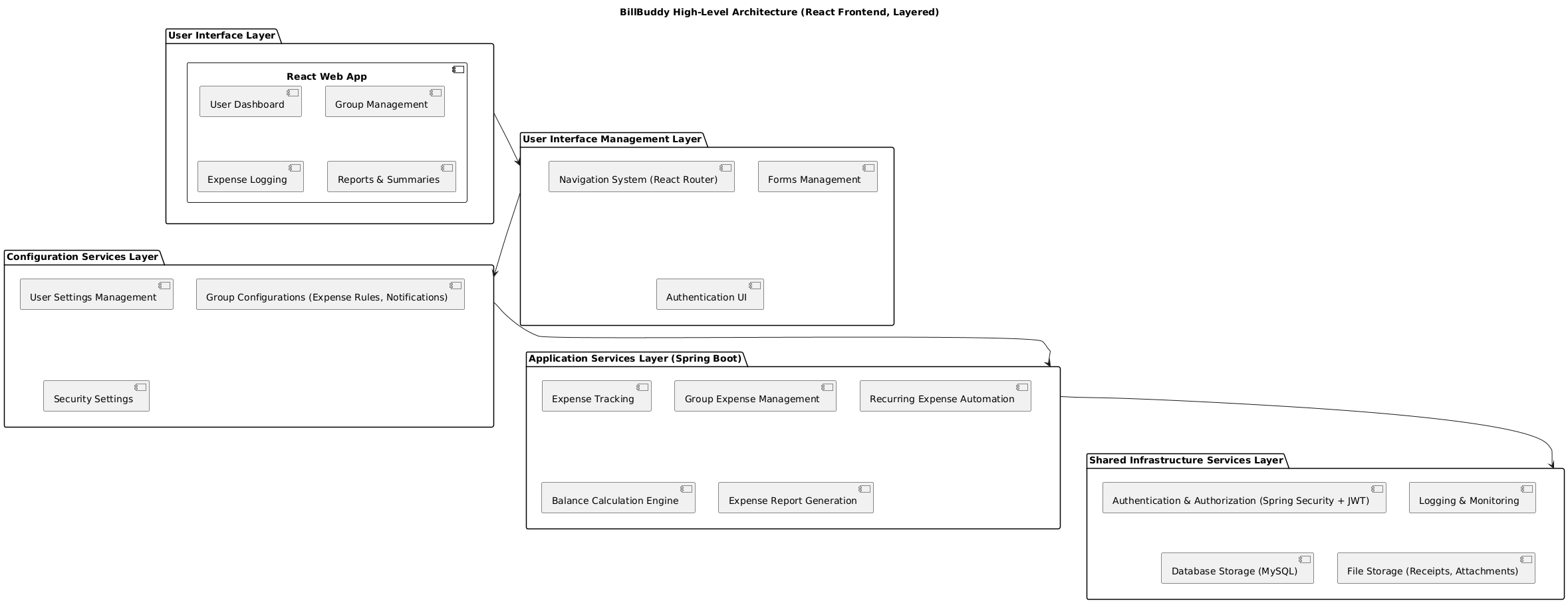
#### High-Level Architecture Diagram



#### Define a modular component structure for your project.

## ****User Interface Layer****

This layer provides the front-end interaction for users through **React** applications.

**Components:**

* **Mobile App**
  + User dashboard
  + Group management
  + Expense logging
  + Reports and summaries

## ****User Interface Management Layer****

This layer is responsible for handling UI operations and navigation, ensuring smooth user experience.

**Components:**

* **Forms management**: Expense entry, group creation
* **Navigation system**: React Navigation
* **Authentication UI**: Login, signup, password recovery

## ****Configuration Services Layer****

This layer enables **system customization and configuration** based on user preferences and group settings.

**Components:**

* **User settings management**
* **Group configurations** (Expense rules, notification settings)
* **Security settings**

## ****Application Services Layer****

This is the core business logic layer that handles expense tracking and financial calculations.

**Components:**

* **Expense tracking**: Log, edit, and delete expenses
* **Group expense management**: Split costs among members
* **Recurring expense automation**: Monthly rent, subscriptions
* **Balance calculation engine**: Tracks debts and settlements
* **Expense report generation**

## ****Shared Infrastructure Services Layer****

This is the **backend and storage layer**, ensuring security, authentication, and data storage.

**Components:**

* **Authentication & Authorization (Spring Security + JWT)**
* **Logging & monitoring**: Tracks errors and user activity
* **Database storage (MySQL)**: Users, groups, expenses, logs
* **File storage**: Uploading receipts for expenses

#### Identify dependencies between components and discuss potential coupling issues.

### ****User Interface Layer → User Interface Management Layer****

* **Dependency:** The **React** relies on the **navigation system** and **forms management** to handle user interactions.
* **Potential Coupling Issue:** If the **forms management or navigation system** is tightly coupled to the UI components, any changes to navigation (e.g., adding a new user flow) might require modifying multiple UI components.

### ****User Interface Management Layer → Configuration Services Layer****

* **Dependency:** The UI forms and settings pages depend on **user settings management** and **group configurations** to fetch data dynamically.
* **Potential Coupling Issue:** If UI management directly fetches configuration data from the backend **without a caching layer**, it can lead to performance bottlenecks due to frequent API calls.

### ****Configuration Services Layer → Application Services Layer****

* **Dependency:** Group settings (e.g., **expense split rules**) directly influence how expenses are processed.
* **Potential Coupling Issue:** If configuration settings are deeply embedded in the expense tracking logic, **changing group rules may require modifying the core expense tracking service**, increasing the risk of breaking other functionalities.

### ****Application Services Layer → Shared Infrastructure Services Layer****

* **Dependency:** The **expense tracking module** and **group management module** rely on **MySQL storage, authentication, and logging services** to ensure accurate data handling.
* **Potential Coupling Issue:** If the **expense tracking service** is tightly coupled to MySQL queries, **migrating to another database or making schema changes** will be difficult.

#### Outline a strategy for handling updates in different components over time

## ****API Versioning Strategy**** (Back-End - Spring Boot)

**Problem:** Changes in the API can **break existing front-end functionality** if not managed properly.

**Solution:** Implement **REST API versioning** to maintain backward compatibility.

**Implementation:**

* **Versioning in URL paths**: Maintain multiple API versions (e.g., /api/v1/expenses and /api/v2/expenses).
* **Deprecation strategy**: Keep older API versions for a **defined period** before phasing them out.

## ****Front-End Component Versioning****

**Problem:** UI changes may cause **disruptions** for existing users if deployed without proper control.

**Solution:**  
**Component-Based Updates:** Use a **modular React architecture** where components are updated **independently** (e.g., updating the dashboard without modifying group management).

## ****Database Migration****

**Problem:** Schema changes may cause **data inconsistency** or **downtime**.

**Solution:** Use **Flyway** for automated **database migrations**.

**Implementation:**

* Store migration scripts in **version-controlled SQL files**.
* Apply migrations **incrementally** without affecting running services.