Software Design Document
Smart Budget Tracker

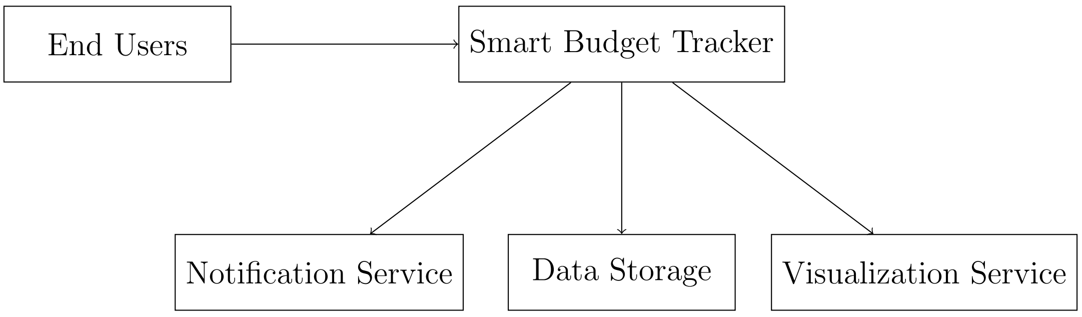
Emirhan Yavuz Berşan Kayra Korkmaz Sabri Mert Pişkin Çağan Durgun

1 System Overview

The **Smart Budget Tracker** is a personal finance management application that enables users to track expenses, manage budgets by category, and receive intelligent real-time alerts when approaching or exceeding set limits. The system emphasizes ease of use, clear data visualization, and smart notifications, making budgeting proactive and intuitive. It also serves as a practical implementation example of multiple software design patterns to ensure scalability and maintainability.

2 System Context

The following diagram shows the high-level context of the system and its interactions:



Actors:

- **End Users**: Interact with the system via a mobile or web interface to manage expenses and budgets.
- **Notification Service**: Provides real-time alerts when budget limits are approached or exceeded.
- **Data Storage**: Ensures persistent storage of expenses, budgets, and user configurations.
- **Visualization Service**: Generates graphs, reports, and dashboards for expense and budget analysis.

3 Key Features and Functionality

The Smart Budget Tracker includes the following primary functionalities:

Expense Tracking

- Add, edit, and delete expenses with details (category, amount, date).
- Support for recurring expenses (e.g., monthly subscriptions).

Budget Management

- Set customizable budget limits for each category.
- Monitor spending relative to set limits.

Smart Notifications

- Real-time alerts when approaching or exceeding a category budget.
- Notifications for recurring expenses or unusual spending patterns.

Dashboard and Reports

- Visual overview of spending by category.
- Remaining budget visualization.
- Analytics including graphs, trends, and summary reports.

Multi-Account Support

- Ability to manage expenses from different sources (e.g., cash, bank account, credit card).

Design Pattern Usage

- **Observer Pattern**: For real-time notifications on budget threshold breaches.
- **Strategy Pattern**: For flexible budget optimization suggestions.
- **Singleton Pattern**: For managing shared resources like the notification service.

4 Assumptions and Dependencies

Assumptions

- The application is used by individuals, not organizations.
- Users manually enter expenses; no automatic bank API integration is included.
- The application runs on mobile and desktop environments with responsive design.
- Basic AI suggestions for budgeting are rule-based, not involving complex machine learning models.

Dependencies

- **Persistent Storage**: Local database (e.g., SQLite, Realm) for storing expenses and budgets.
- **Notification Framework**: Platform-dependent notification services for real-time alerts.
- **Graphing Library**: For generating interactive visualizations (e.g., Chart.js, D3.js).
- Development frameworks and tools (e.g., React Native, Flutter, or platform-specific alternatives).

5 Architectural Design

5.1 System Architecture Diagram (High-Level)

The Smart Budget Tracker system is designed using a layered architecture based on the Spring Boot framework. The architecture consists of the following major layers:

- **Controller Layer:** Handles HTTP requests and maps them to appropriate service methods.
- **Service Layer:** Contains business logic and orchestrates operations across repositories and other components.
- **Repository Layer:** Handles database interactions using Spring Data JPA.
- **Persistence Layer:** Manages data storage in an embedded H2 database.

The system exposes RESTful APIs at http://localhost:8080, allowing interaction through standard HTTP methods (GET, POST, PUT, DELETE).

5.2 Architectural Patterns and Styles

The architecture follows:

- **Layered Architecture:** Clear separation of concerns between controllers, services, and repositories.
- **MVC Pattern:** Spring Boot's MVC model is used to map requests to business logic.
- **RESTful Design:** Exposes resources via REST APIs, supporting standard HTTP verbs.

5.3 Rationale for Architectural Decisions

Spring Boot was chosen due to its simplicity, scalability, and rich ecosystem. Layered architecture improves maintainability and testability by isolating business logic, persistence logic, and presentation. The RESTful approach ensures interoperability with future front-end clients (e.g., React or Angular).

6 Component Design

6.1 Subsystems and Modules

The system consists of the following key components:

- **ExpenseController** – Handles HTTP requests related to expense operations.
- **BudgetController** – Manages budget-related requests.
- **ReportController** – Generates reports based on expense and budget data.
- **ExpenseService**, **BudgetService**, **ReportService** – Contain business logic.
- **ExpenseRepository**, **BudgetRepository** – Interface with the H2 database.
- **Model Classes (Expense, Budget, Category)** – Represent domain entities.

6.2 Responsibilities of Each Component

- **Controllers** – Accept API calls, validate input, delegate to services.
- **Services** – Implement core business logic and apply design patterns.
- **Repositories** – Provide data access methods through Spring Data JPA.
- **Models** – Map to database tables, enforce entity relationships.

6.3 Interfaces Between Components

- Controllers call services via dependency injection.
- Services call repositories for data persistence.
- Repositories interact directly with the H2 database.

6.4 Component Diagrams

The component diagram includes:

- **ExpenseController** → **ExpenseService** → **ExpenseRepository**
- **BudgetController** → **BudgetService** → **BudgetRepository**
- **ReportController** → **ReportService**

7 Data Design

7.1 Data Model / ER Diagram

The data model consists of:

- **Expense (expense_id, amount, date, category)**
- **Budget (budget_id, limit, category)**
- **Category (category_id, name)**

Relationships:

- Each expense belongs to one category.
- Each budget is associated with one category.

7.2 Data Storage (Database or File Structure)

The application uses an embedded **H2 database**. The schema is auto-generated by Spring Data JPA based on entity annotations.

7.3 Data Flow Diagrams

- **User** submits an expense via REST API.
- **Controller** validates and forwards to the service.
- **Service** applies business rules and persists via repository.
- **Repository** saves to the H2 database.
- **ReportService** fetches and aggregates data for reports.

7.4 Data Validation Rules

- Expense amounts must be positive decimal values.
- Dates must be valid ISO dates (YYYY-MM-DD).
- Categories must exist before assigning to expenses or budgets.
- Budget limits must be non-negative.

8 Design Patterns

Applied Design Patterns

The Smart Budget Tracker applies a variety of design patterns tailored to its core use cases to ensure modularity, scalability, and maintainability:

- **Core Case 1: Add / Edit / Delete Expense**
  - **Command Pattern**: Encapsulates each user action (add/edit/delete) as an object, enabling features like undo/redo and improving action traceability.
  - **Factory Pattern**: Used to create standardized expense objects based on type (e.g., fixed vs variable expenses), improving modularity and scalability.
- **Core Use Case 2: Set Budget & Receive Notifications**
  - **Strategy Pattern**: Enables flexible budget evaluation logic by allowing interchangeable strategies (e.g., weekly, monthly, custom thresholds).
  - **Observer Pattern**: Automatically triggers notifications when spending nears or exceeds defined limits, by observing relevant budget state changes.
- **Core Use Case 3: View Analytics Dashboard**
  - **Builder Pattern**: Builds complex analytics reports from various data points in a stepwise manner, supporting extensibility and modular report generation.
  - **Facade Pattern**: Simplifies complex backend interactions (data fetch, calculations, formatting) behind a unified dashboard interface.
- **Core Use Case 4: Manage Spending Categories**
  - **Factory Pattern**: Centralizes creation of user-defined or predefined spending categories, ensuring consistent structure and easing future additions.
- **Dashboard and Expense Analysis/Reporting**
  - **Builder Pattern**: Can be used to enable the step-by-step construction of complex expense reports and analyses.
  - **Facade Pattern**: Can be used to simplify complex financial calculations and reporting operations. The dashboard allows the user to easily view financial summaries without needing to know the complex backend processes.
- **Application Configuration and Settings Management**
  - **Singleton Pattern**: Can be used to provide a single, central access point for managing application-wide configuration settings and preferences. This pattern ensures that settings are accessed and modified consistently throughout the application.
- **Integration with Different Data Sources or External APIs**

  - **Adapter Pattern**: When the application needs to retrieve data from different data sources or external services, this pattern can be used to adapt these external interfaces to the application's internal data format. This increases the system's flexibility and integration capability.

- **Adding Additional Features to Expense Entries**
  - **Decorator Pattern**: Can be used to add additional features (e.g., tags, payment methods, etc.) to basic expense entries. This pattern allows for dynamically adding new functionalities without modifying the core expense object.

- **Managing Application States (Viewing, Editing, Analyzing)**
  - **State Pattern**: Can be used to manage different user interface states of the application. This encapsulates each state's behavior, making the code more organized and sustainable.

Context and Justification

The chosen design patterns align with the application's goals of maintainability, extensibility, and user experience:

- **Command Pattern** facilitates undo/redo functionality and improves action management, critical for user operations on expenses.
- **Factory Pattern** centralizes object creation, promoting consistency and easing future expansions for expense types and categories.
- **Strategy Pattern** allows easy modification or addition of budget evaluation methods, supporting diverse user needs.
- **Observer Pattern** provides timely notifications crucial for proactive budget management.
- **Builder** and **Facade Patterns** simplify complex report generation and backend processes, enhancing user interface simplicity.
- **Singleton Pattern** ensures consistent application-wide settings management.
- **Adapter Pattern** enables flexible integration with varied data sources without disrupting internal system logic.
- **Decorator Pattern** supports dynamic feature addition to expenses, enhancing customization.
- **State Pattern** helps manage UI states cleanly, improving code organization and user interaction flow.

Together, these patterns build a robust and scalable system architecture that supports current requirements and future enhancements.

9 Implementation Notes

Project Overview

The project was developed as a collaborative team effort and is built on the Spring Boot framework.

Configuration and Infrastructure

- Local MySQL database connection is configured via `application.properties`.
- Connection pool management is optimized using *HikariCP* for high-performance resource handling.

Technologies Used

- Database operations are managed with *JPA*, enabling automatic synchronization of table structures.
- *Lombok* is leveraged to minimize boilerplate code and improve developer productivity.
- *Liquibase* handles database versioning and change tracking, ensuring reliable schema evolution.

Design and Implementation

- The system follows a *layered architecture*, promoting separation of concerns and maintainability.
- Established *design patterns* are applied to enhance code readability and ensure scalable development.
- API endpoints are designed adhering to *REST principles*, prioritizing simplicity and functionality.
- Project structure focuses on sustainability and team alignment, with responsibilities clearly assigned based on defined standards.

10 User Interface Design

The Smart Budget Tracker provides a clean and user-friendly interface. It features:

- A quick expense entry form with category selection, amount input, and description field.
- A budget overview section showing spending by category against defined limits.
- A recent transactions list displaying latest expenses with timestamps.
- A placeholder for spending analysis graphs.

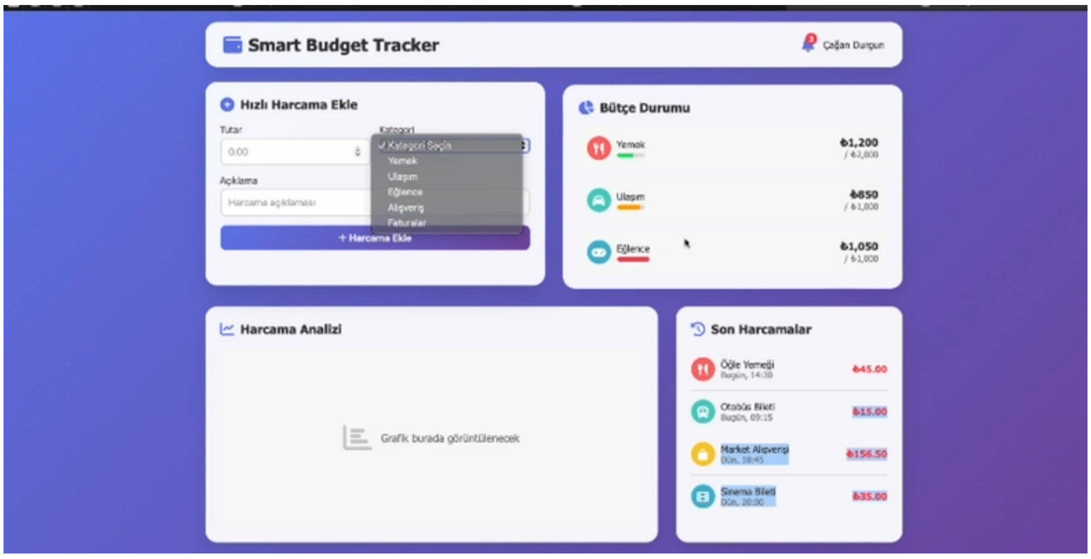


Figure 1: Smart Budget Tracker User Interface

11 External Interfaces

The Smart Budget Tracker exposes a set of RESTful APIs that provide access to core functionalities of the system. These APIs follow standard HTTP methods (GET, POST, PUT, DELETE) and return responses in JSON format for ease of integration with various clients.

APIs

- **Expense API:**
  Endpoints under `/expenses` allow clients to create, retrieve, update, and delete expense records.
- **Budget API:**
  Endpoints under `/budgets` provide operations for managing budget categories and limits.
- **Report API:**
  Endpoints under `/reports` enable generation of summary reports and analytics data.
- **Health Check / Misc:**
  Potential endpoints (e.g., `/actuator/health`) can be added for monitoring and diagnostics.

These APIs are accessible locally via http://localhost:8080 and are designed for future integration with front-end applications (e.g., React, Angular) or external consumers.

Third-party Systems

The system relies on the following third-party components:

- **MySQL:** External database server used for persistent storage of all application data.
- **Liquibase:** A third-party tool used for database versioning, schema migrations, and change tracking.
- **HikariCP:** A high-performance JDBC connection pool library integrated with Spring Boot for efficient database connection management.
- **Lombok:** A compile-time code generation library that reduces boilerplate code in the model and service layers.

There is currently no direct integration with external financial institutions, payment providers, or public APIs. The system is designed to be extensible for such integrations in the future via API clients or adapter patterns.

12 Performance Considerations

Performance Requirements

The Smart Budget Tracker application is designed to provide real-time responsiveness, especially for notification delivery and dashboard updates. The system must efficiently handle frequent expense entries and budget checks without noticeable delay to ensure a smooth user experience. Performance should remain consistent even with large datasets of expenses and budgets, supporting potentially thousands of entries without degradation.

Scalability & Optimization Strategies

To ensure scalability, the application employs efficient data structures for managing expenses and budgets. Lazy loading and pagination techniques are used for rendering large datasets in the UI. The notification system is optimized via event-driven mechanisms (Observer pattern) to avoid unnecessary polling. Backend services, if any, are designed to be stateless and horizontally scalable to accommodate growing user bases. Caching strategies are applied where applicable to minimize repeated calculations and database hits.

13 Error Handling and Logging

Exception Management

The system implements robust exception handling to gracefully manage unexpected conditions such as data input errors, storage failures, or notification delivery issues. Validation checks prevent invalid data entry at the UI and backend layers. Exceptions are caught and handled with user-friendly error messages, and fallback mechanisms are in place to maintain system stability.

Logging Mechanisms

Comprehensive logging is integrated throughout the application to record significant events, errors, and system states. Logs support debugging, performance monitoring, and auditing. Log levels (e.g., info, warning, error) are used to categorize messages appropriately. Sensitive user data is excluded from logs to ensure privacy.

14 Design for Testability

The system architecture promotes testability by modularizing components and applying design patterns that decouple dependencies. Unit tests cover core functionalities such as expense management, budget calculations, and notification triggers. Integration tests verify interactions between modules and external services. Automated testing pipelines ensure code quality and facilitate continuous integration and delivery.

15 Deployment and Installation Design

Environment Configuration

Packaging and Dependencies

16 Change Log

Version	Date	Description
0.1	2025-06-20	Initial project proposal and requirement gathering.
0.3	2025-06-20	Completed system overview, context, and key features documentation.
0.5	2025-06-26	Added architectural design and component design sections.
0.7	2025-06-26	Finalized data design, design patterns, and implementation notes.
0.9	2025-06-26	Incorporated performance considerations and error handling sections.
1.0	2025-06-30	Added user interface design mockups and external interfaces documentation.
1.1	2025-07-01	Updated change log and included deployment and testability sections.

17 Future Work / Open Issues

- *Authentication and Authorization:*
  Complete development and integration of authentication and authorization modules to secure the system comprehensively.
- *Input Validation and Error Handling:*
  Enhance input validation mechanisms and error handling to ensure data integrity and maintain application stability.

- *Automated Testing:*
  Implement unit and integration tests to uphold code quality and support continuous development and deployment cycles.
- *Database Performance Optimization:*
  Optimize database queries and indexing strategies to efficiently manage larger data volumes and improve response times.
- *Logging and Monitoring:*
  Expand logging and monitoring capabilities for effective issue tracking, system diagnostics, and maintenance.
- *API Documentation:*
  Finalize and maintain comprehensive API documentation to support developers and facilitate seamless integration.
- *Frontend Integration and UI Improvements:*
  Refine frontend integration and user interface to boost usability, responsiveness, and overall user experience.
- *Project Documentation:*
  Improve project documentation including setup instructions and contribution guidelines to streamline onboarding and collaboration.

Addressing these issues is critical to advancing the project toward production readiness and ensuring long-term maintainability.