**Software Requirements Specification (SRS)**

**Smart Personal Budget Planner**

|  |  |
| --- | --- |
| **Project Name** | Smart Personal Budget Planner |
| **Document Version** | 1.0 |
| **Date** | February 5, 2025 |
| **Prepared By** | Group 1 |
| **Company** | Mekelle University |
| **Location** | Mekelle, Ethiopia |

****

**Table of Contents**

1. **Introduction**  
   1.1 Purpose  
   1.2 Document Conventions  
   1.3 Project Scope  
   1.4 References
2. **Overall Description**  
   2.1 Product Perspective  
   2.2 User Classes and Characteristics  
   2.3 Operating Environment  
   2.4 Design and Implementation Constraints  
   2.5 Assumptions and Dependencies
3. **System Features**  
   3.1 User Management
   * 3.1.1 Description
   * 3.1.2 Functional Requirements  
     3.2 Budget Management
   * 3.2.1 Description
   * 3.2.2 Functional Requirements  
     3.3 Expense Tracking
   * 3.3.1 Description
   * 3.3.2 Functional Requirements  
     3.4 Financial Reporting
   * 3.4.1 Description
   * 3.4.2 Functional Requirements  
     3.5 Security
   * 3.5.1 Description
   * 3.5.2 Functional Requirements  
     3.6 Savings Goals
   * 3.6.1 Description
   * 3.6.2 Functional Requirements  
     3.7 Notifications
   * 3.7.1 Description
   * 3.7.2 Functional Requirements
4. **Data Requirements**  
   4.1 Logical Data Model  
   4.2 Data Dictionary  
   4.3 Reports  
   4.4 Data Acquisition, Integrity, Retention, and Disposal
5. **External Interface Requirements**  
   5.1 User Interfaces  
   5.2 Software Interfaces  
   5.3 Hardware Interfaces  
   5.4 Communications Interfaces
6. **Quality Attributes**  
   6.1 Usability  
   6.2 Performance  
   6.3 Security  
   6.4 Safety  
   6.5 Reliability  
   6.6 Maintainability
7. **Internationalization and Localization Requirements**  
   7.1 Core Internationalization Requirements  
   7.2 Ethiopian Localization (Phase 1)  
   7.3 Global Localization (Future Phases)  
   7.4 Technical Implementation  
   7.5 Testing and Validation
8. **Other Requirements**  
   8.1 Legal and Regulatory Compliance  
   8.2 Installation and Configuration  
   8.3 Startup and Shutdown  
   8.4 Logging and Monitoring  
   8.5 Audit Trails  
   8.6 Accessibility Standards  
   8.7 Disaster Recovery  
   8.8 Transition Requirements

**Appendices**

* **Appendix A: Glossary**
* **Appendix B: Use Cases**
  + UC-01: User Registration
  + UC-02: Set Monthly Budget
  + UC-03: Create Savings Goal
* **Appendix C: Analysis Models**
  + Figure 1: System Architecture Diagram
  + Figure 2: Entity-Relationship Diagram (ERD)
  + Figure 3: Sequence Diagram
  + Figure 4:

**Introduction**

**1.1 Purpose**

This Software Requirements Specification (SRS) document defines the requirements for Release 1.0 of the **Smart Personal Budget Planner**, a web and mobile application designed to help users manage personal finances through automated expense tracking, budgeting, savings planning, and financial insights.

**Intended Audience:**

* **Developers**: Guide system design and implementation.
* **Project Managers**: Align deliverables with scope and timelines.
* **Testers**: Derive test cases and validate functionality.
* **Stakeholders/Investors**: Understand project feasibility and ROI.
* **End Users**: Review features and provide feedback during beta testing.

This document focuses on the core financial management subsystem of the application. Future phases, such as AI-driven advice and bank integration, are not included in this release.

**1.2 Document Conventions**

This SRS follows the IEEE 830-1998 standard. Key conventions include:

* **Functional Requirements**: Labeled as FR-XXX (e.g., FR-001: User Registration).
* **Non-Functional Requirements**: Labeled as NFR-XXX (e.g., NFR-002: Performance).
* **Use Cases**: Labeled as UC-XX (e.g., UC-01: User Registration).
* **Styling**:
  + **Bold**: Section headers.
  + *Italics*: Terms defined in Appendix A.

Manual requirement additions must follow the FR-XXX/NFR-XXX labeling sequence to avoid duplication.

**1.3 Project Scope**

The **Smart Personal Budget Planner** replaces error-prone manual methods (e.g., spreadsheets) with an automated platform to:

* Track income and expenses in real time.
* Set and monitor personalized budgets.
* Generate actionable financial reports and alerts.
* Promote financial literacy through goal-based savings planning.

**Alignment with Strategic Goals:**

* **User Value**: Empower individuals to achieve financial awareness and stability.
* **Business Objective**: Capture market share in the personal finance tool sector by addressing gaps in existing solutions.

**Key Features in Release 1.0:**

1. Manual expense logging and CSV/OFX file imports.
2. Customizable monthly budgets with overspending alerts.
3. Savings goal tracking with progress visualization.
4. Multi-language support (English, Amharic).

**Out of Scope:**

* Direct bank account integration (planned for Phase 2).
* AI-driven investment recommendations (future roadmap).

**1.4 References**

### ****1. IEEE Software Requirements Specification Standard****

* **Title**: IEEE 830-1998 – Recommended Practice for Software Requirements Specifications
* **Source**: IEEE (Institute of Electrical and Electronics Engineers)
* **URL**: <https://standards.ieee.org/>

### ****2. Database & Backend Documentation****

* **Title**: MySQL 8.0 Reference Manual (For database implementation)
* **Source**: Oracle
* **URL**: <https://dev.mysql.com/doc/refman/8.0/en/>

### ****3. Express.js Documentation****

* **Title**: Express.js Official Documentation (For backend development)
* **Source**: Express.js
* **URL**: <https://expressjs.com/>

## 2. Overall Description

### 2.1 Product Perspective

The **Smart Personal Budget Planner** is a new web-based application designed to simplify personal financial management. It operates as a standalone system with no dependencies on existing products or integrations with external services (e.g., banks, SMS gateways).

#### Context and Origin:

The product is developed to replace manual budgeting methods (e.g., spreadsheets) and address the lack of user-friendly, internet-based financial tools for expense tracking and budget analysis. It is built as a monolithic web application using:

* **Frontend**: React.js
* **Backend**: Express.js
* **Database**: MySQL

#### Key Interfaces:

* **Frontend-Backend**: RESTful APIs for data exchange (e.g., expense submission, report generation).
* **Backend-Database**: MySQL connections via Sequelize ORM for CRUD operations.

#### Visual Model :

* **User** interacts with the **React Frontend**.
* **Frontend** communicates with **Express.js Backend** via API calls.
* **Backend** reads/writes data to **MySQL Database**.
* No external systems are connected.

### 2.2 User Classes and Characteristics

| **User Class** | **Characteristics** | **Key Requirements** |
| --- | --- | --- |
| **Basic Users** | Individuals managing personal/household budgets. | - FR-001 (Registration)- FR-003 (Expense Tracking)- FR-005 (Budget Alerts) |
| **Administrators** | Manage user accounts and system health. | - FR-009 (Data Encryption)- NFR-004 (User Management) |

#### Favored User Class:

* **Basic Users**: Primary target audience (90% of expected users).
* **Administrators**: Internal staff.

#### Reusability:

User roles align with the Stakeholder Catalog in the Vision and Scope document (Section 1.4).

### 2.3 Operating Environment

#### Software/Hardware Environment:

* **Frontend**:
  + **Browsers**: Chrome (v90+), Firefox (v88+), Safari (v14+).
  + **Devices**: Desktops, tablets, smartphones (responsive via Bootstrap).
* **Backend**:
  + **Server**: Node.js v16+ Render.
  + **APIs**: RESTful endpoints for expense/budget management.
* **Database**:
  + **MySQL**: Supabase.
  + **Storage**: 10GB initially, scalable to 100GB.

#### Geographical Scope:

* **Users**: Accessible globally but optimized for Ethiopia (ETB currency support).

### **Diagrams**

**Figure 1:** System Architecture Diagram in [Appendix C](https://chatgpt.com/c/679fd183-ccfc-8000-b798-0d8af90f5b75" \l "appendix-c-analysis-models) includes Savings Goals.

### 2.4 Design and Implementation Constraints

#### Technology Stack:

* **Frontend**: React.js and Bootstrap (mandatory for UI consistency and responsiveness).
* **Backend**: Express.js (required for REST API development).
* **Database**: MySQL (chosen for relational data structuring).

#### Regulatory:

* Data encryption (AES-256) for user passwords and financial data.

#### Architecture:

* Monolithic design (no microservices) to simplify deployment.

#### Rationale:

* React.js enables reusable UI components; Bootstrap ensures mobile responsiveness.
* MySQL provides ACID compliance for reliable financial data management.

### 2.5 Assumptions and Dependencies

#### Assumptions:

* **Internet Access**: Users have stable connectivity (no offline functionality).
  + **Risk**: Users in low-connectivity areas (e.g., rural Ethiopia) may face access issues.
* **Browser Compatibility**: Modern browsers with JavaScript enabled.
  + **Risk**: Legacy browsers (e.g., IE11) will not render the React app correctly.
* **User Behavior**: Expenses/budgets are input manually and accurately.
  + **Risk**: Data entry errors may skew reports.

#### Dependencies:

* **External Libraries**:
  + react-chartjs-2 (for expense visualizations).
  + express-validator (for input sanitization).
* **Hosting Services**:
  + AWS EC2 (backend) and RDS (MySQL) uptime.
  + Netlify/AWS Amplify (frontend hosting).
* **Security**:
  + JWT token generation/validation for user sessions.

## ****3. System Features****

### ****3.1 User Management****

#### ****3.1.1 Description****

Priority: **High**  
This feature allows users to create accounts, log in, and manage their profiles. It ensures secure access and personalization of financial data.

#### ****3.1.2 Functional Requirements****

* **FR-001**: Users shall register with a valid email, username, and password.
  + Error Handling: Reject invalid email formats (e.g., user@invalid) and passwords without 1 uppercase letter and 1 symbol.
* **FR-002**: Users shall log in using their registered email/username and password.
  + Error Handling: Display "Invalid credentials" after 3 failed attempts; lock account for 15 minutes after 5 failed attempts.
* **FR-003**: Users shall reset forgotten passwords via email verification.
  + Error Handling: Validate email existence before sending reset links.

### ****3.2 Budget Management****

#### ****3.2.1 Description****

Priority: **High**  
Enables users to set monthly budgets, track spending against limits, and receive alerts.

#### ****3.2.2 Functional Requirements****

* **FR-004: Users shall input a monthly budget (e.g., 10,000 ETB) while minimizing borrowing from the previous month.**
  + Error Handling: Reject non-numeric or negative values; display "Budget must be ≥ 0."
* **FR-005**: The system shall calculate the remaining budget in real-time.
  + Error Handling: Handle division errors (e.g., if budget is 0).
* **FR-006**: Users shall receive alerts when spending exceeds 80% of the budget.
  + Error Handling: Trigger alerts only once per budget period.

### ****3.3 Expense Tracking****

#### ****3.3.1 Description****

Priority: **High**  
Allows users to manually log expenses with categories, dates, and descriptions.

#### ****3.3.2 Functional Requirements****

* **FR-007**: Users shall add expenses with amount, category, date, and description.
  + Error Handling: Reject future dates; show "Select a category" for invalid entries.
* **FR-008**: Users shall edit or delete expenses within the current budget period.
  + Error Handling: Restrict edits to the current user’s expenses.
* **FR-009**: Expenses shall display in a sortable table (by date, amount, category).
  + Error Handling: Show "No expenses recorded yet" for empty states.

### ****3.4 Financial Reporting****

#### ****3.4.1 Description****

Priority: **Medium**  
Generates visual and textual summaries of spending patterns and budget adherence.

#### ****3.4.2 Functional Requirements****

* **FR-010**: Users shall view pie charts showing expense distribution by category.
  + Error Handling: Default to "No data" if no expenses exist.
* **FR-011**: Users shall export reports as PDF/CSV files.
  + Error Handling: Disable export button if no data is available.
* **FR-012**: The system shall provide textual insights (e.g., "You spent 25% more on groceries").
  + Error Handling: Use placeholders like "Set a budget to see insights" if no budget exists.

### ****3.5 Security****

#### ****3.5.1 Description****

Priority: **High**  
Ensures data privacy and protection for user accounts and financial information.

#### ****3.5.2 Functional Requirements****

* **FR-013**: Passwords shall be stored as salted hashes (bcrypt algorithm).
  + Error Handling: Log hashing errors and block account creation until resolved.
* **FR-014**: API endpoints shall require valid JWT tokens for access.
  + Error Handling: Return 401 Unauthorized for invalid/expired tokens.
* **FR-015**: User sessions shall expire after 24 hours of inactivity.
  + Error Handling: Redirect to login page with "Session expired."

**3.6 Savings Goals**

**3.6.1 Description**

**Priority**: Medium  
Allows users to define, track, and manage savings objectives with target amounts and deadlines.

**3.6.2 Functional Requirements**

* **FR-016**: Users shall set savings goals with a target amount (e.g., 50,000 ETB) and deadline (e.g., "December 2024").
  + **Error Handling**: Reject deadlines earlier than the current date.
* **FR-018**: Users shall allocate funds from their monthly budget to savings goals.
  + **Error Handling**: Prevent allocations exceeding the remaining budget.

**3.7 Notifications**

**3.7.1 Description**

**Priority**: Medium  
Delivers real-time alerts and reminders for budget thresholds, savings progress, and system updates.

**3.7.2 Functional Requirements**

* **FR-019**: Users shall receive alerts when spending exceeds 80%, 90%, and 100% of their budget.
  + **Error Handling**: Alerts trigger once per budget cycle to avoid spam.
* **FR-020**: Notifications shall be accessible via a centralized panel (bell icon) with mark-as-read and delete options.
  + **Error Handling**: Limit notifications to 30 days of history.

## ****4. Data Requirements****

### ****4.1 Logical Data Model****

The system processes four core data entities: **Users**, **Expenses**, **Budgets**, and **Savings Goals**. Relationships are defined as follows:

* **User** → **Expense**
  + One user can log **many** expenses.
* **User** → **Budget**
  + One user sets **one** budget per month.
* **User** → **Savings Goal**
  + One user can create **multiple** savings goals.
* **Expense** → **Budget**
  + Expenses are linked to a monthly budget.

#### ****Entity Descriptions****

* **User**: Stores account credentials (email, password hash) and profile data.
* **Expense**: Captures transaction details (amount, category, date, description).
* **Budget**: Defines monthly spending limits.
* **Savings Goal**: Tracks target amounts, deadlines, and progress toward savings objectives.

**Visual Model**:

* **Figure 2**: ER-D in [Appendix C](https://chatgpt.com/c/679fd183-ccfc-8000-b798-0d8af90f5b75" \l "appendix-c-analysis-models) includes Savings Goals.

### ****4.2 Data Dictionary****

#### ****Users****

* **user\_id**: Integer (Primary Key, auto-incremented).
* **email**: String (255 chars, unique, required).
* **password\_hash**: String (255 chars, bcrypt-hashed, required).
* **created\_at**: Timestamp (auto-generated on account creation).

#### ****Expenses****

* **expense\_id**: Integer (Primary Key, auto-incremented).
* **user\_id**: Integer (Foreign Key → Users.user\_id).
* **amount**: Decimal (≥0 ETB, two decimal places).
* **category**: String (50 chars, e.g., "Food", "Transport").
* **date**: Date (≤ current date).

#### ****Budgets****

* **budget\_id**: Integer (Primary Key, auto-incremented).
* **user\_id**: Integer (Foreign Key → Users.user\_id).
* **monthly\_limit**: Decimal (≥0 ETB, two decimal places).
* **month\_year**: Date (Format: YYYY-MM-01).

#### ****Savings Goals****

* **goal\_id**: Integer (Primary Key, auto-incremented).
* **user\_id**: Integer (Foreign Key → Users.user\_id).
* **target\_amount**: Decimal (>0 ETB, two decimal places).
* **current\_amount**: Decimal (≥0 ETB, defaults to 0).
* **start\_date**: Date (≤ end\_date).
* **end\_date**: Date (≥ start\_date).
* **status**: Enum ("Active", "Achieved", "Abandoned").

### ****4.3 Reports****

#### ****Monthly Summary Report****

* **Content**:
  + Total spent, remaining budget, top expense categories.
  + Savings goal progress (if applicable).
* **Sorting**: Expenses sorted by date (oldest first).
* **Format**: PDF/CSV with pie charts and summary text.

#### ****Savings Progress Report****

* **Content**:
  + Target vs. current savings.
  + Progress percentage and time remaining.
* **Sorting**: Goals sorted by deadline (soonest first).
* **Format**:
  + Web: Progress bars and timelines (Chart.js).
  + Export: PDF/CSV with columns: Goal, Target, Saved, Deadline.

### ****4.4 Data Acquisition, Integrity, Retention, and Disposal****

#### ****Data Acquisition****

* **Manual Input**: Users enter expenses, budgets, and savings goals via React forms.
* **Savings Updates**: Users manually adjust current\_amount or allocate funds from budgets.

#### ****Data Integrity****

* **Validation Rules**:
  + current\_amount ≤ target\_amount (Savings Goals).
  + end\_date ≥ start\_date (Savings Goals).
  + date ≤ current date (Expenses).
* **Database Constraints**: Foreign keys, unique indexes, and NOT NULL checks.

#### ****Data Retention****

* **Active Savings Goals**: Retained until marked "Achieved" or "Abandoned."
* **Archived Goals**: Stored for 1 year post-completion/deletion.

#### ****Data Disposal****

* Users can delete savings goals, permanently erasing them from the database.
* Backups containing savings data are purged after 30 days.

### ****Key Enhancements****

* **Savings Integration**: Users can track both spending and saving in one platform.
* **Progress Visualization**: Dashboard widgets show savings progress (e.g., "₵5,000/₵20,000 saved for vacation").

**5. External Interface Requirements**

**5.1 User Interfaces**

* **UI Standards**:
  + Follow **Bootstrap v5** style guide for consistent fonts, colors, and component design.
  + Use predefined color schemes:
    - Primary: #2c3e50
    - Secondary: #3498db
  + Icons from **Font Awesome** for universal recognition (e.g., 💰 for expenses, 📅 for dates).
* **Layout**:
  + Responsive design for screen widths ≥320px (mobile) to 1920px (desktop).
  + Fixed navigation bar with links: *Dashboard*, *Expenses*, *Budgets*, *Reports*.
* **Accessibility**:
  + Alt text for images and ARIA labels for screen readers.
  + High-contrast mode for visually impaired users.
* **Validation**:
  + Real-time input validation (e.g., red borders for invalid amounts).
  + Tooltips for errors (e.g., "Date cannot be in the future").

**5.2 Software Interfaces**

* **Frontend-Backend API**:
  + **RESTful Endpoints**:
    - POST /api/expenses: Submit new expense (JSON: { amount: 500, category: "Food", date: "2023-10-15" }).
    - GET /api/budgets: Fetch monthly budget (Response: { monthly\_limit: 10000, remaining: 4500 }).
  + **Authentication**: JWT tokens in request headers (Authorization: Bearer <token>).
* **Third-Party Libraries**:
  + **Chart.js**: Render expense breakdown charts.
  + **React-Table**: Sortable/filterable expense lists.
* **Database**:
  + **MySQL**: Tables for Users, Expenses, Budgets, SavingsGoals, Notifications, Report, Catagories.
  + **Sequelize ORM**: Define model relationships and validations.

**5.3 Hardware Interfaces**

* **Device Compatibility**:
  + **Browsers**: Chrome, Firefox, Safari (latest versions).
  + **Mobile**: Android 8+ and iOS 12+ devices.
* **Input Methods**:
  + Touchscreen support for mobile devices.
  + Keyboard navigation (tab order aligned with UI flow).

**5.4 Communications Interfaces**

* **HTTPS**: All data transmitted over TLS 1.3.
* **CORS**: Restrict API access to trusted domains (e.g., https://smartbudget.et).
* **Data Formats**:
  + JSON for API payloads.
  + CSV for report exports.

**6. Quality Attributes**

**6.1 Usability**

* **Ease of Use**:
  + First-time users can add an expense in ≤3 clicks.
  + 90% of users rate navigation as "intuitive" in post-launch surveys.
* **Error Recovery**:
  + Auto-save draft expenses to localStorage to prevent data loss.
  + Undo/redo actions for expense edits.

**6.2 Performance**

* **Response Times**:
  + Dashboard loads in ≤2 seconds (95th percentile).
  + Charts render in ≤3 seconds under 3G network conditions.
* **Scalability**:
  + Support 1,000 concurrent users with ≤5s latency.

**6.3 Security**

* **Data Protection**:
  + AES-256 encryption for sensitive fields (passwords, JWTs).
  + Rate-limiting (100 requests/minute) to prevent brute-force attacks.
* **Compliance**:
  + GDPR principles applied (user consent for data collection).
  + Input sanitization to block SQL injection/XSS attacks.

**6.4 Safety**

* **Data Integrity**:
  + Daily backups with SHA-256 checksums to detect tampering.
  + ACID transactions for budget/expense updates.
* **Session Safety**:
  + Invalidate sessions after 24 hours of inactivity.

**6.5 Reliability**

* **Uptime**: 99.9% availability monitored via AWS CloudWatch.
* **Error Handling**:
  + Log errors to AWS CloudWatch for real-time alerts.
  + Fallback UI states (e.g., "Reports temporarily unavailable").

**6.6 Maintainability**

* **Code Quality**:
  + ESLint + Prettier for consistent React/Express.js code.
  + 80% unit test coverage for critical modules (e.g., budget calculations).
* **Documentation**:
  + Swagger API docs for backend endpoints.

**7. Internationalization and Localization Requirements**

The **Smart Financial Manager System** will support global users by adhering to regional conventions for currency, language, date/time, and cultural norms. Below are the requirements to ensure adaptability across different locales.

**7.1 Core Internationalization Requirements**

* **Currency Handling**:
  + Default currency: **Ethiopian Birr (ETB)** with the symbol "ብር" and format: *1,000.00 ETB*.
  + Support for other currencies (future scope) with auto-conversion based on user location (e.g., USD, EUR).
* **Date and Time**:
  + **Date Format**: *DD/MM/YYYY* (Ethiopian standard) with optional locale-specific formats (e.g., MM/DD/YYYY for US users).
  + **Calendar System**: Support Gregorian and Ethiopian calendars (future phase).
* **Number Formatting**:
  + Use commas as thousand separators and periods as decimal points (e.g., *10,000.50 ETB*).
  + Localize for regions using reverse notation (e.g., *10.000,50 ETB* in Europe).
* **Language Support**:
  + **Primary Languages**: English (default) and Amharic (አማርኛ) with Ge‘ez script.
  + **Future Languages**: Arabic (right-to-left support), French, Spanish.

**7.2 Ethiopian Localization (Phase 1)**

* **Amharic Support**:
  + **UI Localization**: Translate labels, buttons, and tooltips into Amharic.
  + **Font Rendering**: Use open-source fonts (e.g., *Noto Sans Ethiopic*) for Ge‘ez script.
  + **Input Methods**: Accept Amharic characters via keyboard or copy-paste.
* **Cultural Adaptations**:
  + **Expense Categories**: Include common Ethiopian categories (e.g., *ቤተክርስቲያን አበል* for church donations).
  + **Holidays**: Highlight Ethiopian public holidays in the calendar (e.g., *Meskel*, *Timkat*).
* **Address and Phone Formats**:
  + Ethiopian phone numbers: *+251 XX XXX XXXX*.
  + Address fields: Support regional structures (e.g., *Kebele*, *Woreda*).

**7.3 Global Localization (Future Phases)**

* **Multilingual UI**:
  + Store translations in JSON files for easy scaling (e.g., en.json, am.json, ar.json).
  + Use libraries like i18next (React) or react-intl for dynamic language switching.
* **Timezone Handling**:
  + Display dates/times in the user’s local timezone (e.g., *EAT* for Ethiopia, *PST* for California).
* **Regional Legal Compliance**:
  + **EU**: GDPR-compliant data collection consent banners.
  + **Ethiopia**: Adhere to *Data Protection Proclamation No. 1207/2020*.

**7.4 Technical Implementation**

* **Backend**:
  + Use **ICU (International Components for Unicode)** libraries for date/number formatting.
  + Store locale preferences in the Users table (e.g., locale: en-ET).
* **Frontend**:
  + Dynamic text replacement via translation keys (e.g., t('welcome\_message')).
  + Right-to-left (RTL) layout support for Arabic (future).
* **Database**:
  + Unicode (UTF-8) encoding for multilingual data storage.

**7.5 Testing and Validation**

* **Pseudolocalization**:
  + Test UI layouts with pseudo-translated text to detect overflow or alignment issues.
* **Localization Testing**:
  + Validate Amharic translations with native speakers for accuracy.
  + Ensure currency conversions align with real-time exchange rates (future).

**Test Case 1: Validate Amharic Date Formats (DD/MM/YYYY) in Expense Logs**

* **Steps**:
  1. Set the locale to am-ET.
  2. Add an expense with the date "15/10/2023".
  3. Verify that the date is stored as 2023-10-15 in MySQL.
* **Expected Outcome**: The date should be correctly stored in the database in the YYYY-MM-DD format, ensuring proper conversion from Amharic date format.

**Test Case 2: Ensure Ge‘ez Script Renders Correctly on Low-End Android Devices**

* **Steps**:
  1. Test UI labels (e.g., "አዲስ ወጪ") on Tecno Spark 8 (Android 11, 2GB RAM).
  2. Verify there is no font corruption or layout overflow.
* **Expected Outcome**: The Ge‘ez script should render without issues on lower-end devices, ensuring the UI remains responsive and accessible.

**8. Other Requirements**

**8.1 Legal and Regulatory Compliance**

* **Ethiopian Data Laws**:
  + Adhere to *Data Protection Proclamation No. 1207/2020* for user consent, data storage, and breach notifications.
  + Store user data on AWS Middle East (Bahrain) region to comply with Ethiopian data localization guidelines.
* **GDPR Compliance**:
  + Provide EU users with opt-in consent banners for data collection.
  + Allow data deletion requests via email (processed within 72 hours).
* **Licensing**:
  + Comply with open-source licenses (MIT for React.js, Apache 2.0 for Express.js).

**8.2 Installation and Configuration**

* **Deployment**:
  + **Frontend**: Host on Netlify/AWS Amplify with environment variables for API endpoints.
  + **Backend**: Deploy Express.js on Render EC2 with Node.js v18+ and PM2 process manager.
  + **Database**: MySQL 8.0+ on Supabase with automated backups enabled.
* **Configuration**:
  + Use .env files for sensitive data (e.g., JWT secret, AWS credentials).
  + Provide setup scripts for initial admin user creation.

**8.3 Startup and Shutdown**

* **Startup**:
  + Run database migration scripts automatically on server startup.
  + Health checks for API endpoints (e.g., /health returns 200 OK).
* **Shutdown**:
  + Gracefully terminate active sessions and complete pending transactions.
  + Log shutdown events with timestamps and reasons (e.g., maintenance, crash).

**8.4 Logging and Monitoring**

* **Log Types**:
  + **Security Logs**: User logins, password resets, and role changes.
  + **Activity Logs**: Expense additions, budget updates, report exports.
  + **Error Logs**: API failures, validation errors, database timeouts.
* **Retention**:
  + Retain logs for 90 days in AWS CloudWatch; archive older logs to S3.
* **Monitoring**:
  + Set CloudWatch alarms for API latency (>5s) and error rates (>5%).

**8.5 Audit Trails**

* **User Actions**:
  + Track edits/deletions to expenses and budgets (e.g., "User #12 updated Budget to 15,000 ETB").
  + Log IP addresses and timestamps for critical actions (e.g., report exports).
* **Audit Reports**:
  + Generate CSV reports of audit trails for admins (columns: User ID, Action, Timestamp, IP).

**8.6 Accessibility Standards**

* **WCAG 2.1 Compliance**:
  + Ensure keyboard navigation for all UI components (e.g., tab order, focus states).
  + Provide text alternatives for charts (e.g., "Pie chart: 40% spent on Food").
* **Screen Readers**:
  + Use ARIA labels for dynamic content (e.g., "Alert: Budget exceeded 80%").

**8.7 Disaster Recovery**

* **Backups**:
  + Daily encrypted backups of MySQL data stored in AWS S3.
  + Test backups quarterly for integrity and restoration speed.
* **Recovery Objectives**:
  + **RTO (Recovery Time Objective)**: ≤4 hours for critical system restoration.
  + **RPO (Recovery Point Objective)**: ≤1 hour of data loss tolerance.

**8.8 Transition Requirements**

* **Data Migration**:
  + Support CSV imports for users switching from legacy systems (e.g., spreadsheets).
  + Validate CSV formats during upload (columns: Amount, Category, Date).
* **User Training**:
  + Provide in-app tutorials for first-time users (e.g., "How to set a budget").

**Appendix A: Glossary**

| **Term** | **Definition** |
| --- | --- |
| **ACID** | Atomicity, Consistency, Isolation, Durability: Database transaction properties ensuring reliability. |
| **JWT** | JSON Web Token: A secure method for transmitting authentication claims via tokens. |
| **ERD** | Entity-Relationship Diagram: A visual model of data entities and their relationships. |

### ****Appendix B: Use Cases****

#### ****UC-01: User Registration****

|  |  |
| --- | --- |
| Id and Name: | UC-01- Register User |
| Created By: | Group 1 date created: January 29,2025 |
| Primary Actor :  Secondary Actor : | New User  None |
| Description: | This use case describes the process of a new user creating an account within the Smart Personal Budget Planner application. |
| Trigger | New User initiates the account creation process. |
| Preconditions | The user has access to the internet and a device capable of accessing the application.  The application is available and functional. |
| Post conditions | A new user account is created in the system database.  The user receives a confirmation message or email.  The user is able to log in to the application using their new credentials. |
| Normal flow | 1. The New User navigates to the "Create Account" screen.  2.The New User enters their desired email address.  3.The New User enters and confirms their password.  4.The New User clicks the "Create Account" button.  5.The system validates the entered information (e.g., checks for valid email format, password strength).  6.The system creates a new user account in the database.  7.The system sends a confirmation message/email to the user's provided email address.  8.The system displays a success message and redirects the user to the login page.  Alternative flows |
| Invalid input | 1. The New User enters invalid information (e.g., empty fields, invalid email format, weak password).  2.The system displays an error message indicating the invalid input.  3.The New User corrects the information and resubmits the form. |
| User name already exists | The New User enters an email address that is already in use.  The system displays an error message indicating that the email address is already registered.  The New User enters a different email address and resubmits the form. |
| System error | The system encounters an unexpected error during account creation.  The system displays an error message to the user.  The system logs the error for troubleshooting. |
| Exception | Database connection failure during registration. |
| Priority | High |
| Frequency of use | High (expected to occur frequently as new users join the system) |
| Business rules | Password must meet minimum complexity requirements (e.g., length, character types). |
| Assumtion | User have stable connection and necessary hardware and softwar |

#### ****UC-02: Set Monthly Budget****

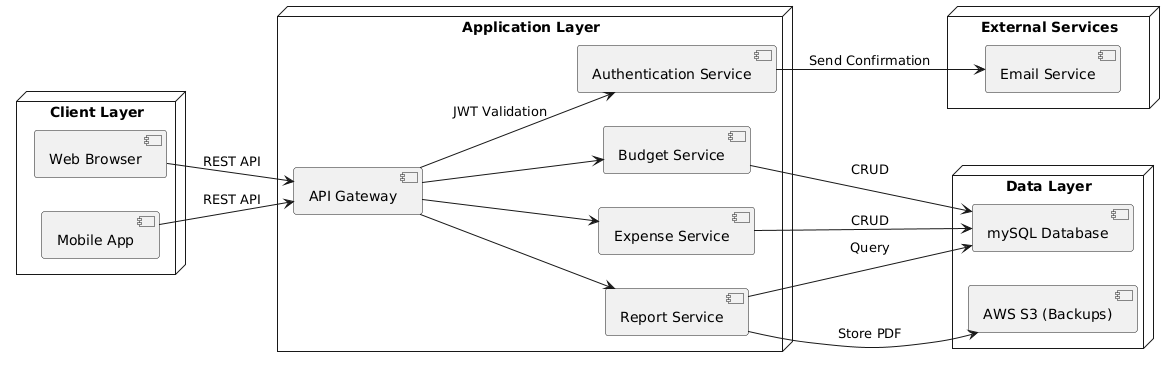
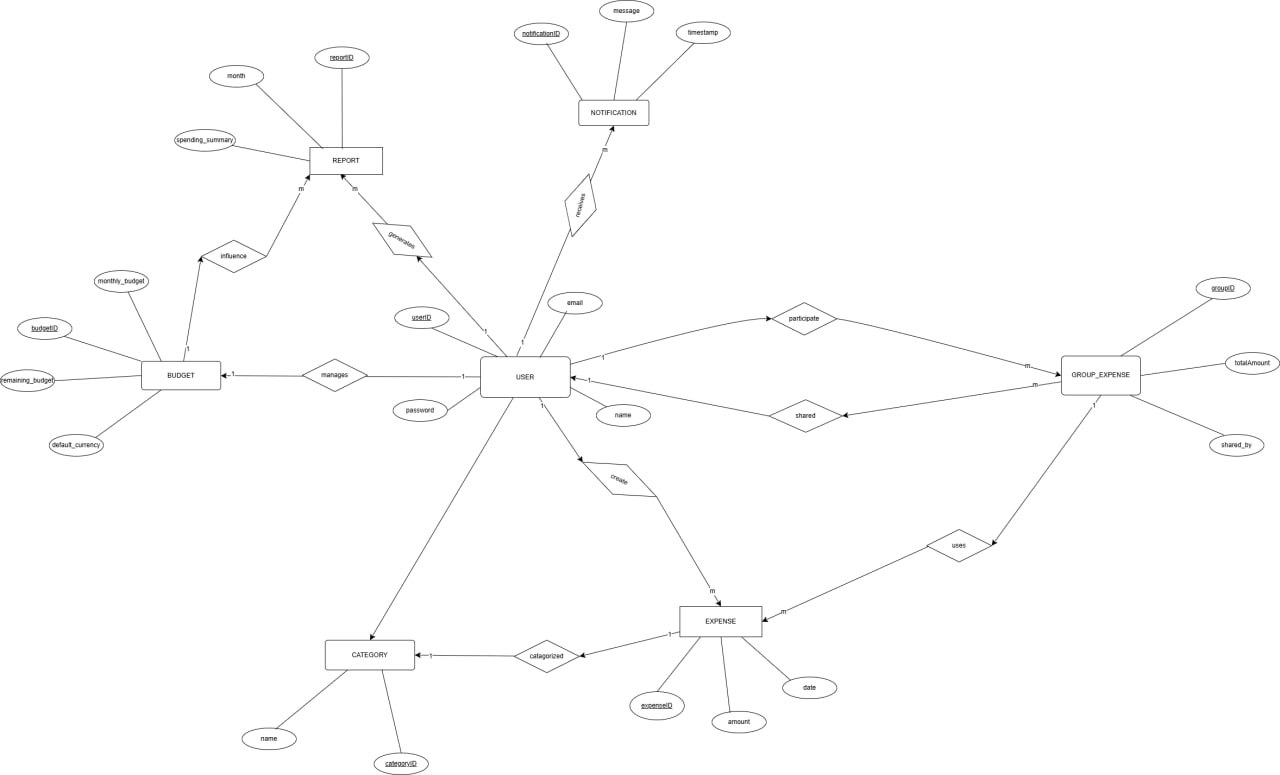
|  |  |
| --- | --- |
| ID and Name | UC-02 : Set Budget |
| Created by | Group 1 Date Created January 29,2025 |
| Primary actor | User Secondary actor: none |
| Description | This use case describes the process of a user setting a monthly budget within the Smart Personal Budget Planner application. |
| Trigger | User initiates the budget setting process. |
| Precondition | The User is logged in to the application. |
| Postconditions | A new monthly budget is created or updated for the user.  The system calculates and displays the available budget for expenses. |
| Normal flow | * + - The User navigates to the "Budget" section of the application.     - The User enters their monthly income.     - The User (optionally) sets a savings goal.     - The system calculates the available budget for expenses.     - The system displays the calculated budget and suggested daily spending limits.     - The User reviews and confirms the budget.     - The system saves the budget settings for the user. |
|  | **ALTERNATIVE FLOWS** |
| Missing input | The User fails to enter their monthly income.  The system displays an error message prompting the user to enter their income.  The User enters their income and proceeds to step 4 |
| Invalid input | The User enters invalid input (e.g., negative income).  The system displays an error message indicating invalid input.  The User corrects the input and proceeds to step 4. |
| Exceptions | Borrow from the previous months and API failure during budget calculation. |
| Priority | High |
| Frequency of use | High (expected to occur frequently, potentially at the beginning of each month) |
| Business rules | Budgets reset at the start of each month. |
| Other information | None |
| Assumptions | The user has a stable internet connection.  The user has the necessary software and hardware to access the application. |

#### ****UC-03:**** Add Expense

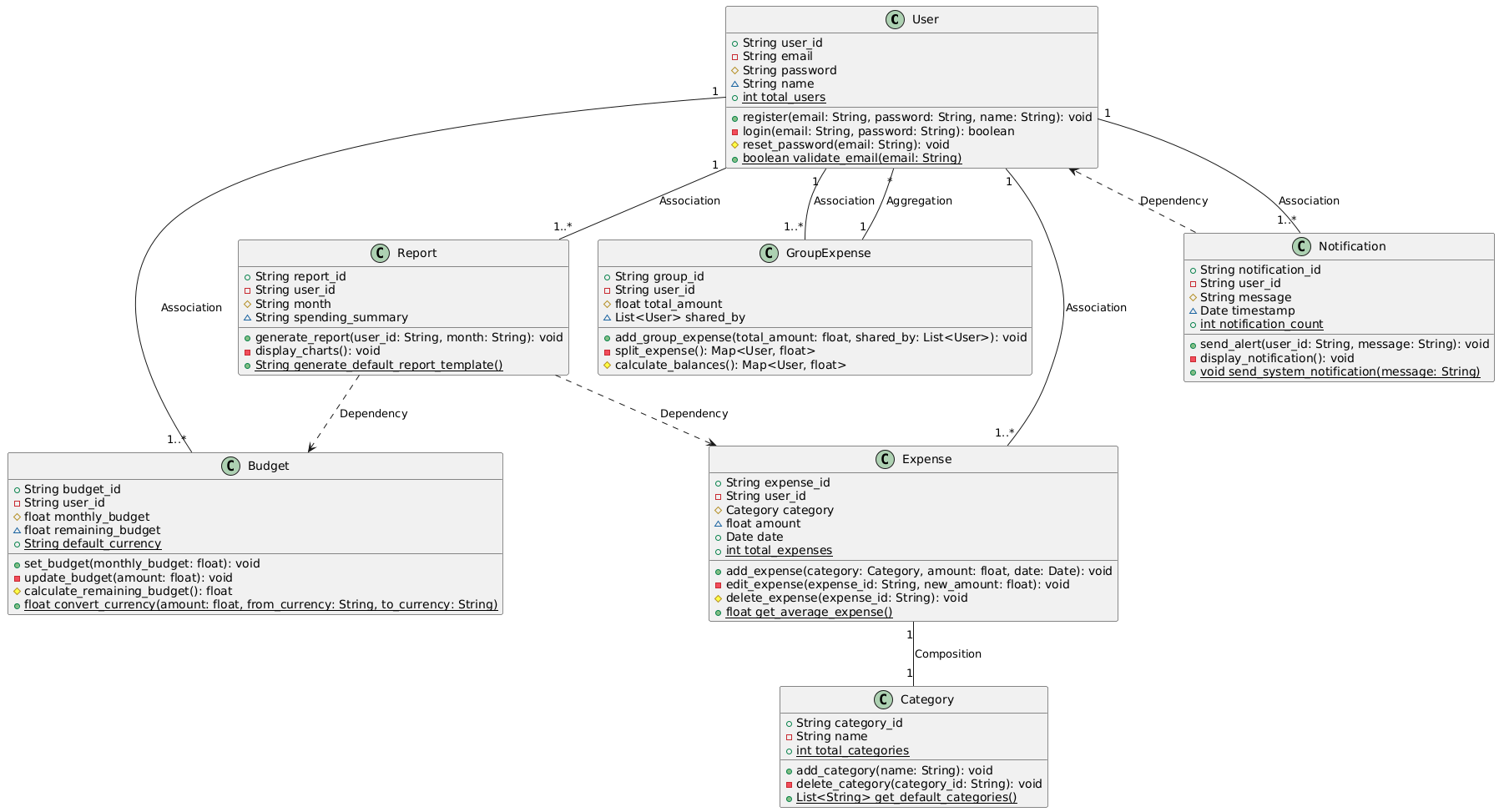
|  |  |
| --- | --- |
| **Use Case ID and Name:** | UC-03-Add Expense |
| **Created By** | Group-1 **Date Created:** January 29,2025 |
| **Primary Actor** | User **Secondary Actors:** None |
| **Description:** | This use case describes the process of a user recording an expense within the Smart Personal Budget Planner application. |
| **Trigger:** | User initiates the expense recording process. |
| **Preconditions:** | The User is logged in to the application.  A budget has been set for the current month. |
| **Postconditions:** | A new expense record is successfully added to the user's expense history.  The user's available budget is updated to reflect the new expense. |
| **Normal Flow:** | The User navigates to the "Add Expense" screen.  The User selects the expense date.  The User enters the expense amount.  The User selects or enters the expense category.  The User (optionally) adds a description for the expense.  The User clicks the "Save" button.  The system validates the entered data.  The system saves the expense record to the database.  The system updates the user's available budget.  The system displays a success message and updates the expense list. |
|  | 2 ALTERNATIVE FLOWS |
| **Invalid Input:** | The User enters invalid data (e.g., negative amount, invalid date).  The system displays an error message indicating invalid input.  The User corrects the input and resubmits the expense. |
| **No Internet Connection:** | The User attempts to add an expense while offline.  The system stores the expense locally.  Once an internet connection is restored, the application synchronizes the local data with the server. |
| **Exceptions** | None |
| **Priority:** | High |
| **Frequency of Use:** | Very High (expected to occur multiple times per day for most users) |
| **Business Rules:** | None |
| **Assumptions:** | The user has a stable internet connection.  The user has the necessary software and hardware to access the application. |

**Appendix C: Diagrams**

**Figure 1: System Architecture**

**Figure 2: ER Diagram**

**Figure 3: Class Diagram**



**Figure 4: Sequence Diagram**

