SETA - Design and Implementation

SETA Smart Expense Tracker Application

Document Version:
Version 3.0
Revision: Initial Release

Printing Date:
May 5, 2025

Group Information:
Group ID: D2

Members:
Tang Bok Hei (1155193297)
Siu Wing Yiu (1155192075)
Leung Ho Chun (1155193014)
So Kin Pui (1155193506)

Department:
Department of Computer Science and Engineering

Contents

1 Introduction **2**
- 1.1 Purpose . . . 2
- 1.2 Scope . . . 2
- 1.3 Document Overview . . . 2

2 Architectural Design **2**

3 Component Design **5**
- 3.1 Backend Component Design (seta-api) . . . 5
  - 3.1.1 API Server (app/main.py) . . . 5
  - 3.1.2 Data Models (app/models.py) . . . 6
  - 3.1.3 Configuration Management (app/config_manager.py) . . . 7
  - 3.1.4 Licence Key Generation (app/generate_keys.py) . . . 8
- 3.2 Frontend Component Design (seta-ui) . . . 9
  - 3.2.1 Core Application Setup (src/App.jsx) . . . 9
  - 3.2.2 Routing and Layout (src/components/Dashboard/LayoutContainer.jsx, src/components/Dashbo src/components/common/Sidebar.jsx) . . . 10
  - 3.2.3 State Management and Context . . . 11
  - 3.2.4 Key Frontend Modules (Implementation Summary) . . . 13
- 3.3 Desktop Shell Design (Electron) . . . 14
  - 3.3.1 Main Process (electron.js - Detailed Interaction) . . . 14
  - 3.3.2 Packaging (electron-builder) . . . 16

4 Database Design **17**

5 Deployment and Build Process **18**

6 Security Considerations **19**

7 Limitations and Future Considerations **20**

1 Introduction

1.1 Purpose

This document serves as the primary developer guide for the SETA (Smart Expense Tracker Application). It provides a detailed account of the system's architectural design, the implementation of its core backend and frontend components, data persistence strategies, and key operational aspects like configuration and security. Its goal is to facilitate understanding, maintenance, and future development of the SETA system. For information does not covered in this document, please refer to

https://github.com/sokinpui/3100_Project/tree/main/doc

1.2 Scope

The document covers the technical design and implementation specifics of the `seta-api` (FastAPI backend) and `seta-ui` (React/Electron frontend) projects. This includes API structure, database models, frontend component architecture, state management, inter-process communication within the Electron shell, and the build/deployment pipeline. It details the core technologies employed, including Python 3.9+, FastAPI, SQLAlchemy, React, Electron, Material UI (MUI), and Node.js. It assumes familiarity with these technologies.

1.3 Document Overview

This document is organized as follows:

- **Architectural Design:** High-level overview of the client-server architecture and technology stack.
- **Component Design:** Detailed breakdown of backend and frontend components, including major modules and their responsibilities. UML diagrams are included for key components and interactions.
- **Database Design:** Description of the database schema (with ERD), ORM usage, and supported database types.
- **Deployment and Build Process:** Outline of how the application is packaged using PyInstaller and Electron Builder, managed via GitHub Actions.
- **Security Considerations:** Overview of security measures implemented.
- **Limitations and Future Considerations:** Known limitations and potential areas for future improvement.

2 Architectural Design

SETA follows a classic client-server architecture, physically decoupled into a backend API and a frontend client application, but packaged together for desktop use via Electron. This approach leverages web technologies for the UI while maintaining a robust Python backend for business logic and data persistence.

- **Backend** (`seta-api`)**:**
  - **Framework:** Python 3.9+ with FastAPI. This was chosen for its high performance (built on Starlette and Pydantic), asynchronous support (essential for I/O-bound tasks like database operations), automatic data validation and serialization/deserialization powered by Pydantic type

hints, and integrated interactive API documentation (Swagger UI and ReDoc) which significantly aids development and testing.

- **ORM:** SQLAlchemy provides database abstraction, allowing interaction with database models as Python objects. It facilitates complex queries and supports switching between different relational database systems (SQLite and PostgreSQL in this project) with minimal code changes. Alembic is included for managing database schema migrations, particularly relevant for PostgreSQL deployments.
- **Functionality:** Encapsulates all core business logic:
  * CRUD (Create, Read, Update, Delete) operations for all data entities (Expenses, Income, Accounts, Recurring Items, Budgets, Goals).
  * User authentication (signup, login, password hashing/verification, email verification, password reset flows) and authorization (ensuring users only access their own data).
  * Licence key validation against a predefined list.
  * Data import/export processing logic (CSV parsing, JSON backup/restore).
  * Complex data aggregation and calculation for reporting endpoints.
  * Configuration management (database connection details).

• **Frontend (`seta-ui`):**

- **Framework:** React (v18+), utilizing function components and hooks (`useState`, `useEffect`, `useContext`, `useMemo`, `useCallback`, `useRef`). React provides a declarative, component-based approach, enabling the creation of complex, interactive UIs efficiently.
- **UI Library:** Material UI (MUI) v5 offers a comprehensive suite of pre-built, customizable React components following Material Design principles. This accelerates UI development and ensures visual consistency. Key components used include `DataGrid` (for tables with sorting, pagination, selection), `DatePicker` (`@mui/x-date-pickers` with Day.js adapter), `Card`, `Dialog`, `Button`, `TextField`, `Select`, `Snackbar`, `Alert`, `Tabs`, `LinearProgress`, etc.
- **State Management:** Primarily uses React's built-in hooks (`useState` for component-local state, `useContext` for global state like theme, language, API instance). `localStorage` is used for persisting user sessions, settings, and dashboard layout/filters.
- **Routing:** `react-router-dom` (using `HashRouter` for Electron compatibility) manages navigation between different application modules. Lazy loading (`React.lazy` and `Suspense`) is used for protected modules to improve initial load time.
- **Data Visualization:** `Recharts` is employed within dashboard widgets to render various chart types (Line, Bar, Pie, Area).
- **API Communication:** Axios is used for making HTTP requests to the backend API, managed globally via `ApiContext`.
- **Internationalization:** `i18next` and `react-i18next` handle language translation (English/Chinese).
- **Functionality:** Renders the user interface, manages UI state, handles user interactions (form inputs, button clicks, drag-and-drop), communicates with the backend API, performs client-side validation, and visualizes data.

• **Desktop Shell (Electron):**

– **Runtime:** Electron allows packaging the web-based frontend (React SPA) as a standalone, cross-platform (Windows, macOS, Linux) desktop application, providing native capabilities beyond a standard web browser.
– **Process Management:** The Electron main process (`electron.js`) acts as the application orchestrator. It is responsible for:
  * Creating and managing the native application window (`BrowserWindow`).
  * Loading the React frontend application build (`index.html`).
  * Crucially, spawning the packaged Python backend executable as a child process using Node.js's `child_process.spawn`.
  * Passing necessary environment variables (like `SETA_USER_DATA_PATH`) to the backend process.
  * Managing the lifecycle of both the frontend window and the backend process, ensuring the backend is started on app launch and terminated cleanly on app quit.

This architectural choice promotes modularity (backend and frontend can be developed and tested somewhat independently), leverages the strengths of each technology (Python/FastAPI for backend logic, React/MUI for UI), and delivers a native desktop experience via Electron.

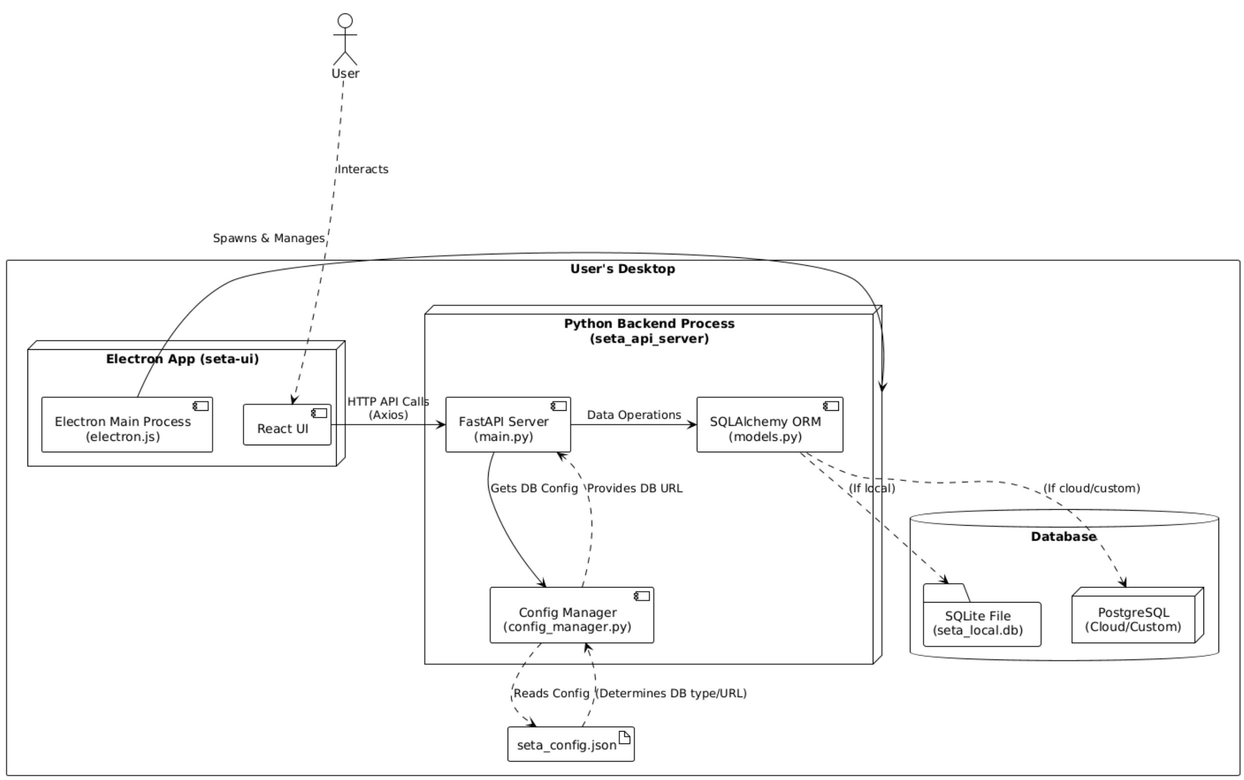

Figure 1: High-Level System Architecture

3 Component Design

3.1 Backend Component Design (`seta-api`)

The backend is structured around the FastAPI framework, organizing logic into the main application file, data models, configuration management, and helper utilities.

3.1.1 API Server (`app/main.py`)

This file serves as the central hub for the backend application.

- **Framework Initialization:** Instantiates the `FastAPI()` application. Sets title and description for documentation purposes.
- **Middleware:** Configures `CORSMiddleware` to allow requests from the frontend (running on `localhost:3000` during development or from `file://` within Electron). Specific origins, methods (`*`), and headers (`*`) are allowed.
- **Database Initialization:**
  - Retrieves the database connection URL using `get_database_url()` from the `config_manager`.
  - Creates the SQLAlchemy `engine` using `create_engine`. Specific arguments (`connect_args={"check_same_thread": False}`) are added for SQLite compatibility.
  - Defines `SessionLocal` using `sessionmaker` bound to the engine.
  - Calls `initialize_local_database()` at startup to ensure the database schema (tables defined in `models.py`) is created automatically if using the local SQLite option and the database file doesn't exist.
- **Dependency Injection:** Defines the `get_db()` function as a FastAPI dependency. This function yields a SQLAlchemy `Session` from `SessionLocal`, ensuring that each API request gets its own database session, which is automatically closed (and changes committed or rolled back) after the request is finished, even if errors occur.
- **API Endpoints:** Defines all RESTful API routes using FastAPI decorators (`@app.get`, `@app.post`, `@app.put`, `@app.delete`). Endpoint functions typically:
  - Define path parameters (e.g., `user_id: int`) and query parameters (e.g., `format: str = "json"`).
  - Specify request body schemas using Pydantic models (e.g., `user_data: UserLogin`). FastAPI handles request body parsing, validation, and potential error responses automatically based on these models.
  - Specify response models using the `response_model` parameter (e.g., `response_model=List[ExpenseResponse]`). FastAPI uses this to filter and validate the response data, and for documentation.
  - Use `Depends(get_db)` to inject the database session (`db: Session`).
  - Implement the core business logic, interacting with the database via SQLAlchemy ORM methods (e.g., `db.query(models.User).filter(...).first()`, `db.add(db_expense)`, `db.commit()`, `db.delete(...)`). Use functions like `func.sum`, `.scalar()`, `.all()`, `.in_()`, `.delete(synchronize_session=False)` for efficiency.

- Perform authorization checks (e.g., verifying `user_id` matches data being accessed).
- Raise `HTTPException` with appropriate status codes (`status.HTTP_404_NOT_FOUND`, `status.HTTP_401_UNAUTHORIZED`, `status.HTTP_400_BAD_REQUEST`, `status.HTTP_403_FORBIDDEN`, `status.HTTP_409_CONFLICT`) and detail messages for expected error conditions.
- Return data conforming to the specified `response_model`.

- **Pydantic Models:** Defined within `main.py` (or could be moved to a separate `schemas.py`). These models define the expected structure and types for API request bodies and responses. They use Python type hints (`int`, `str`, `float`, `date`, `datetime`, `Optional`, `List`, `EmailStr`) and inherit from `BaseModel`. Features like `ConfigDict(from_attributes=True)` enable creating Pydantic models directly from ORM objects. Validators (`@field_validator`) are used for custom validation logic (e.g., password confirmation).

- **Helper Functions:** Includes utilities like:
  - `hash_password(password: str) -> str`: Uses `hashlib.sha256` to securely hash passwords.
  - `verify_password(plain_password: str, hashed_password: str) -> bool`: Compares a plaintext password with a stored hash.
  - `get_user_by_username(db: Session, username: str)`: Retrieves a user by username.
  - Email Sending: `send_verification_email`, `send_password_reset_email` use the `fastapi-mail` library configured with SMTP settings (currently hardcoded Gmail credentials, ideally use environment variables) to send HTML emails with verification/reset links. Link generation uses base URLs (`API_BASE_URL`, `FRONTEND_BASE_URL`) potentially sourced from environment variables.
  - Licence Validation: `validate_licence_key` checks a provided key against the `ACCEPTED_LICENCE_KEYS` set. The `require_active_licence` dependency uses this check to protect specific endpoints, raising a 403 Forbidden error if the user's stored key is invalid.

3.1.2 Data Models (`app/models.py`)

This file defines the database schema using the SQLAlchemy Object-Relational Mapper (ORM).

- **Base Class:** A common `Base = declarative_base()` is created, which all ORM models inherit from.

- **Table Definitions:** Each Python class represents a database table (e.g., `class User(Base): __tablename__ = "users"`).

- **Columns:** Class attributes are defined using `Column(...)`.
  - **Types:** SQLAlchemy types map to database types (`Integer`, `String`, `Numeric` for precise decimals, `Date`, `DateTime(timezone=True)` for timezone awareness, `Boolean`, `SQLAlchemyEnum`).
  - **Constraints:** Primary keys (`primary_key=True`), non-null constraints (`nullable=False`), uniqueness (`unique=True`), database indexes (`index=True` for faster lookups on frequently queried columns like username, email, tokens), default values (`default=0.0`, `default=False`), and server-side defaults/updates (`server_default=func.now()`, `onupdate=func.now()`) are specified.

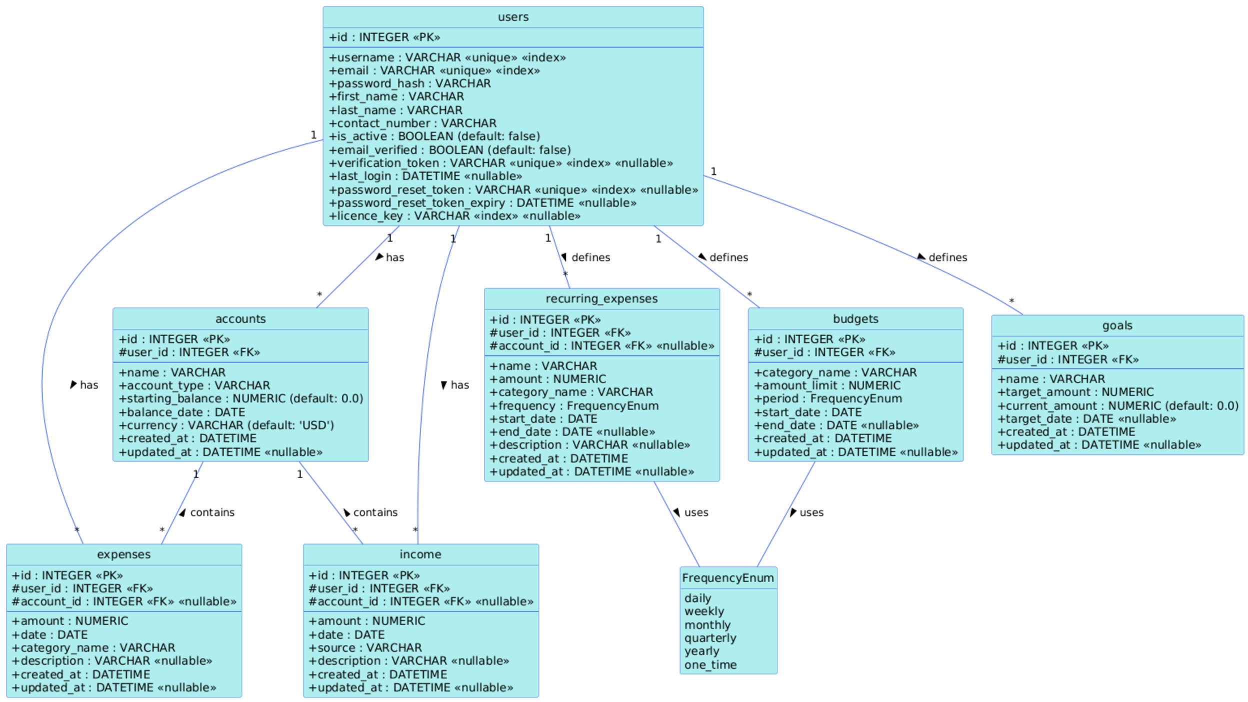

Figure 2: SETA Database Schema (ERD)

  - **Foreign Keys:** `ForeignKey("tablename.columnname", ondelete="CASCADE")` establishes relationships between tables. `ondelete="CASCADE"` ensures that if a referenced record (e.g., a User) is deleted, all referencing records (e.g., that User's Expenses) are automatically deleted by the database, maintaining referential integrity.

- **Relationships:** SQLAlchemy's `relationship()` function defines how ORM objects relate to each other, enabling convenient navigation (e.g., `user.expenses`).
  - **Bidirectionality:** `back_populates="attribute_name"` links relationships on both sides (e.g., `User.expenses` <-> `Expense.user`).
  - **Cascade (ORM level):** `cascade="all, delete"` instructs SQLAlchemy to cascade operations (like deletion) from a parent object to related child objects within the session. This complements the database-level `ondelete="CASCADE"`.
  - **Ordering:** `order_by=Model.column` can specify a default sorting order when accessing a collection relationship.

- **Enums:** Python's standard `enum.Enum` (e.g., `FrequencyEnum`) is used for defining fixed sets of values (like budget periods or recurring frequencies), mapped to database storage using `SQLAlchemyEnum`.

3.1.3 Configuration Management (`app/config_manager.py`)

This module is responsible for loading, saving, and providing access to application settings, primarily the database connection details.

- **User Data Path Determination:** The location for storing configuration (`seta_config.json`) and the local database (`seta_local.db`) is crucial.

1. **Priority 1:** `SETA_USER_DATA_PATH` **Environment Variable:** If this variable is set, its value is used directly. This is the intended mechanism for the packaged Electron application, where the main Electron process determines the standard user data location (e.g., using `app.getPath('userData')`) and passes it to the spawned backend process.
2. **Priority 2: Fallback Path:** If the environment variable is *not* set (e.g., during development or if run standalone):
   - If running from source (`sys.frozen` is false), it defaults to a directory named `seta_user_data` inside the `seta-api` project directory.
   - If running as a packaged executable (`sys.frozen` is true), it attempts to create `seta_user_data` *next to* the executable. This is unreliable due to potential write permission issues (e.g., in `C:Files`).
3. The code attempts to create the determined directory using `Path.mkdir(parents=True, exist_ok=True)` if it doesn't exist.

- **Configuration File (`seta_config.json`):**
  - **Loading (`load_config`):** Attempts to read and parse the JSON file from the user data path. If the file doesn't exist or is invalid, it logs a warning, calls `save_config` to write the `DEFAULT_CONFIG`, and returns the default.
  - **Saving (`save_config`):** Writes the provided configuration dictionary to the JSON file with indentation for readability.
  - **Structure:** Contains a nested structure, primarily `{"database": {"type": "...", "url": "..."}}`.
- **Database URL Logic (`get_database_url`):** This function reads the loaded configuration and determines the appropriate SQLAlchemy connection string (originally design for database switch between cloud and local for web host, but it is too annoying, so we drop it, but we do have the backend logic that handle switching):
  - If `database.type` is `"local"` (or defaults to it), it constructs the SQLite path (`f"sqlite:///LOCAL_DB_PATH.as_posix()"`).
  - If `database.type` is `"cloud"`, it returns the `DEFAULT_CLOUD_DATABASE_URL` (hardcoded fallback, but can be overridden by the `DATABASE_URL` environment variable if set *before* the backend starts).
  - If `database.type` is `"custom"`, it returns the value of `database.url` from the config file. If this URL is missing, it logs a warning and falls back to the `local` SQLite option.
- **Schema Creation Trigger (`initialize_local_database` in `main.py`):** This function, called during startup, checks `is_local_db_configured()` and if the `LOCAL_DB_PATH` file exists. If configured for local DB and the file is absent, it calls `models.Base.metadata.create_all(bind=engine)` to generate the SQLite database schema.

3.1.4 Licence Key Generation (`app/generate_keys.py`)

This is a standalone utility script used during development to create the set of valid licence keys.

- **Purpose:** To generate a predefined number (`num_keys_to_generate`) of unique, random licence keys adhering to the specified format (`AAAA-BBBB-CCCC-DDDD`).

- **Implementation:** Uses Python's `secrets` module (`secrets.choice`) for cryptographically secure random character selection from a pool of uppercase letters (`string.ascii_uppercase`) and digits (`string.digits`). Keys are generated part-by-part and joined with hyphens. A `set` (`generated_keys`) is used to automatically ensure uniqueness while generating keys in a loop until the desired count is reached.
- **Output:**
  1. Prints a Python set literal containing the sorted, generated keys to the standard output. This formatted string is intended to be copied and pasted directly into `app/main.py` to define the `ACCEPTED_LICENCE_KEYS` set.
  2. Writes the sorted list of generated keys, one per line, to a file named `licence_keys.txt` in the same directory, preceded by comments indicating the format and count.

3.2 Frontend Component Design (`seta-ui`)

The frontend is a React Single Page Application (SPA) designed to run within an Electron container, providing the user interface and interacting with the backend API.

3.2.1 Core Application Setup (`src/App.jsx`)

The root component responsible for initializing the application environment.

- **Context Providers:** Wraps the entire component tree with necessary global context providers:
  - `I18nextProvider`: Initializes and provides the `i18next` instance for internationalization, loading translation resources (from `src/locales/`).
  - `ThemeProvider` (from `src/contexts/ThemeContext.jsx`): Manages the application's theme (light, dark, or system preference). It reads the initial theme from `localStorage` (key: `'themeMode'`) or defaults to 'system', provides the current theme mode and a function (`updateTheme`) to change it, and applies the corresponding MUI theme (`createTheme`) using `CssBaseline`.
  - `LanguageProvider` (from `src/contexts/LanguageContext.jsx`): Manages the application language ('english' or 'zh'). Reads initial language from `localStorage` (key: `'language'`), provides the current language and an update function (`updateLanguage`) which also changes the `i18next` language.
  - `ApiProvider` (from `src/services/ApiProvider.jsx`): Creates and provides a pre-configured Axios instance for making API calls. Sets the `baseURL` (e.g., `http://localhost:8000`) and potentially default headers or interceptors if needed. Components access this via `useContext(ApiContext)`.
  - `ModuleProvider` (from `src/contexts/ModuleContext.jsx`): Loads module definitions from `src/modulesConfig.js` and provides them via context (`useModules`).
- **Routing:** Uses `HashRouter` component from `react-router-dom` as the top-level router. Hash routing (`//path`) is generally preferred for Electron apps loading local files (`file://`) to avoid server configuration issues associated with browser history routing.
- **Authentication Guard (`AuthGuard.jsx`):** Rendered immediately inside the router. It checks `localStorage` for the presence of `userId`. Based on the current route (`useLocation`) and authentication status, it either renders its children (`LayoutContainer`) or redirects (`<Navigate to="/login" />` or `<Navigate to="/" />`) to enforce authentication rules. It communicates the initial auth status back to `App.jsx` via a callback prop (`onAuthChange`).

- **Layout (`LayoutContainer.jsx`):** A wrapper component that renders the main application layout, typically including the `ModuleRouter` which defines the content area alongside the persistent `Sidebar`.
- **Initial State:** Reads initial theme and language settings from `localStorage` (or sets defaults). Checks initial login status by inspecting `localStorage` for `userId`.

3.2.2 Routing and Layout (`src/components/Dashboard/LayoutContainer.jsx`, `src/components/Dashboard/Mod` `src/components/common/Sidebar.jsx`)

These components define the application's structure and navigation flow.

- **`ModuleRouter.jsx`:**
  - **Route Definition:** Uses the `useModules` hook to get module configurations from `ModuleContext`. It maps over these configurations to generate <Route> elements within a <Routes> component.
  - **Lazy Loading:** Module components are imported using `React.lazy(() => import(module.componentPath))`. Each lazy-loaded route is wrapped in <React.Suspense fallback={<LoadingSpinner />}> to display a loading indicator while the component code is fetched.
  - **Protected Routes (`ProtectedRoute` HOC):** A Higher-Order Component checks the authentication status (passed down or read from context/localStorage). If authenticated, it renders the passed `element` (the lazy-loaded module component) wrapped within the `Sidebar` component. If not authenticated, it renders <Navigate to="/login" />. Routes requiring protection are wrapped like: <Route path="/expenses" element={<ProtectedRoute element={<ExpenseManager />} />} />.
  - **Public Routes:** Routes for Login (`/login`), Signup (`/signup`), and potentially password reset (`/reset-password/:token`) are defined outside the `ProtectedRoute` structure. Logic is included to redirect *authenticated* users away from these public pages (e.g., navigating to the dashboard `/`) if they try to access them while logged in.
  - **Default Route:** A catch-all route or a default route (e.g., path="/" element={<Navigate to="/dashboard" />}) directs users upon successful login or initial load.
- **`Sidebar.jsx`:** The persistent navigation component for authenticated users.
  - **State:** Manages its own open/closed state (`isSidebarOpen`), logout confirmation dialog state (`logoutDialogOpen`), and fetched licence status (`licenceStatus`, `isLoadingLicence`) using `useState`.
  - **Licence Status Fetching:** Uses `useEffect` and `useCallback` to fetch the user's licence status from the `/users/{userId}/licence` backend endpoint (using the Axios instance from `ApiContext`) when the component mounts or `userId` changes. Stores the result (`'active'`, `'inactive'`, `'not_set'`) in state.
  - **Navigation Items:** Filters modules from `useModules()` context that have `showInSidebar: true`. Maps over these (`sidebarMenuItems`) to render MUI `ListItemButton` components, using `NavLink` from `react-router-dom` for active state styling. Icons are sourced from `module.icon`.
  - **Licence-Based UI:** Before rendering each menu item, it checks `item.requiresLicence`.
    - If `true` and `isLoadingLicence` is true, it might show a spinner or disable the item temporarily.

        * If `true` and `licenceStatus` is not `'active'`, it renders the `ListItemButton` as disabled, adds a `LockIcon` (or similar), and wraps it in a `Tooltip` explaining that a licence is required.
        * If `false` or if `licenceStatus` is `'active'`, it renders the item normally.
    - **Theme/Language Toggles:** Buttons trigger MUI `Menu` components. Selecting an option calls the update functions (`updateTheme`, `updateLanguage`) from the respective contexts.
    - **User Info Display:** Reads `username` and `email` from `localStorage` to display in the sidebar header.
    - **Logout:** The logout button opens a confirmation `Dialog`. On confirmation (`handleLogoutConfirm`), it clears relevant items (`userId`, `username`, `email`, `loginTime`, etc.) from `localStorage` and uses `useNavigate` to redirect the user to the `/login` route.

3.2.3 State Management and Context

State is managed using a combination of React's built-in mechanisms and browser storage.

- **Context API (`useContext`):** Used for sharing global state and functions across the component tree without prop drilling. Key contexts include:
    - `ThemeContext`: Provides current theme mode ('light', 'dark', 'system') and `updateTheme` function.
    - `LanguageContext`: Provides current language ('english', 'zh') and `updateLanguage` function.
    - `ApiContext`: Provides the configured Axios instance for making backend requests.
    - `ModuleContext`: Provides the array of module definitions loaded from `modulesConfig.js`.
- `localStorage`**:** Used for persisting state across browser sessions and application restarts.
    - **Authentication:** Stores `userId`, `username`, `email`, `loginTime` upon successful login. Checked by `AuthGuard` and context initializers. Cleared on logout.
    - **Settings:** Persists user preferences for theme (`'themeMode'`) and language (`'language'`).
    - **Dashboard:** Saves the layout configuration (`'dynamicDashboardLayout_v2'`) generated by `react-grid-layout` and the user's filter settings (`'dynamicDashboardFilters_v2'`, `'dashboardTimePeri`
`'dashboardCustomStartDate'`, `'dashboardCustomEndDate'`) to restore the dashboard state on subsequent visits.
- **Component State (`useState`, `useEffect`, `useRef`, `useMemo`, `useCallback`):** Used extensively within individual components and modules for managing:
    - Form input values and validation errors.
    - Fetched data lists (e.g., expenses, income).
    - Loading indicators (e.g., `isLoading`, `isSubmitting`).
    - Error messages from API calls or validation.
    - Visibility of dialogs, modals, menus, or conditional UI elements.
    - Selections in tables (`selectedIds`) or dropdowns.
    - References to DOM elements (`useRef`) for tasks like triggering file inputs.
    - Memoized calculations or filtered data (`useMemo`) to optimize performance.
    - Memoized callback functions (`useCallback`) passed down as props to prevent unnecessary re-renders of child components.

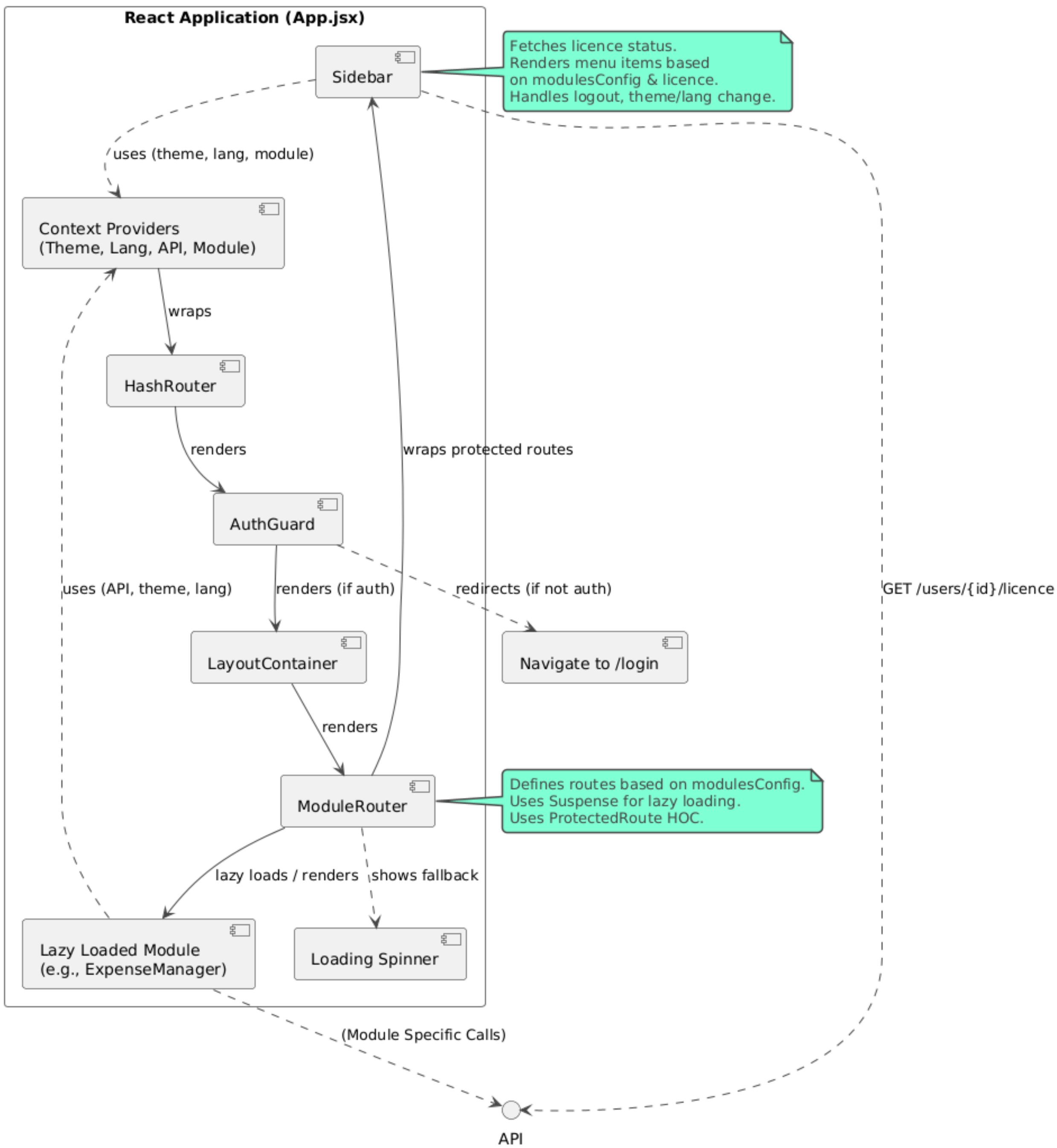

Figure 3: Frontend Core Component Interaction

3.2.4 Key Frontend Modules (Implementation Summary)

Below is a summary of the implementation approach for major frontend feature modules. Refer to the specific Markdown documentation for exhaustive details on each.

- **Authentication (`src/login/`, `src/modules/ResetPassword.jsx`):**
  - **Components:** `Login.jsx`, `Signup.jsx`, `ResetPassword.jsx`.
  - **Functionality:** Handles user login, registration (triggering backend email verification), and password reset (using token from URL). Uses controlled forms, client-side validation (regex for email/phone, password match/complexity), API calls via Axios, `localStorage` for session persistence, and navigation via `useNavigate`. Includes theme/language toggles.
  - **API:** `POST /login`, `POST /signup`, `POST /reset-password/{token}`.
- **Data Management Modules (`src/modules/*Manager/`):** (Expense, Income, Account, Recurring, Planning)
  - **Structure:** Typically follow a pattern: `Manager.jsx` (main orchestrator), `Form.jsx`, `List.jsx`. Planning Manager combines Budgets and Goals using Tabs (`PlanningManager.jsx`, `BudgetView.jsx`, `GoalView.jsx`, etc.).
  - **Functionality:** Fetch data via Axios (`GET /entity/{userId}`), display in MUI `DataGrid` (`List.jsx`) with sorting/pagination/selection, handle adding new items via controlled forms (`Form.jsx`) with MUI inputs (`TextField`, `Select`, `DatePicker`), handle single and bulk deletes (`DELETE /entity/{id}`, `POST /entity/bulk/delete`) with confirmation dialogs. Use shared notification component. Fetch related data if needed (e.g., Accounts for dropdowns). Planning Manager centralizes delete logic for budgets/goals. Goal list includes progress bar visualization. Account Manager handles 409 Conflict on deleting linked accounts.
  - **API:** Standard CRUD endpoints for each entity (`/expenses`, `/income`, `/accounts`, `/recurring`, `/budgets`, `/goals`).
- **Settings (`src/modules/Settings.jsx`):**
  - **Functionality:** Manages multiple settings sections (Profile, Password Change, Licence, Data I/O, DB Config) within one component using MUI Cards. Fetches initial profile/licence data. Handles profile updates, in-app password change (with current password verification), licence key update (client format check + backend validation), JSON data export (triggers backend download), JSON data import (with critical confirmation dialog, uploads file), and database configuration update (warns about restart).
  - **API:** `GET/PUT /users/{userId}`, `PUT /users/{userId}/password`, `GET/PUT /users/{userId}/licence`, `GET /export/all/{userId}`, `POST /import/all/{userId}`, `PUT /settings/database`.
- **CSV Import (`src/modules/ExpenseImport/ExpenseImport.jsx`):**
  - **Functionality:** Provides separate UI sections for Expense and Income CSV imports. Uses hidden file inputs, client-side file type check, uploads file via `FormData` and Axios, displays backend processing results (success/errors/counts).
  - **API:** `POST /expenses/import/{userId}`, `POST /income/import/{userId}`.
- **Reporting (`src/modules/ExpenseReports.jsx`, `src/modules/CustomReports.jsx`):**

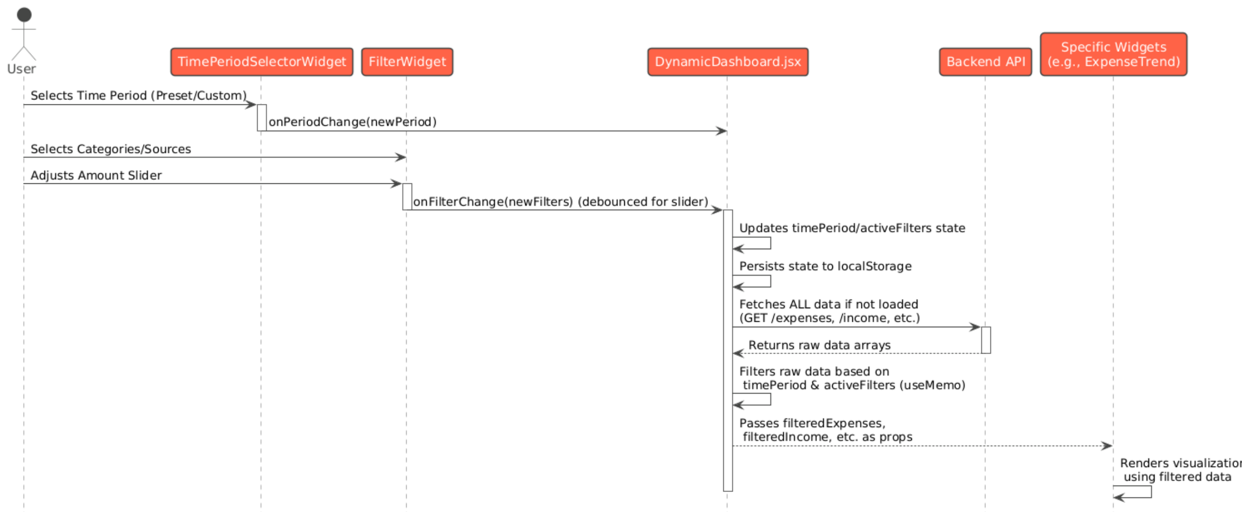

Figure 4: Dashboard Filtering Data Flow

  - **Standard Reports (`ExpenseReports.jsx`):** Fetches consolidated data (`GET /reports/.../all`). Uses client-side libraries (`react-csv`, `xlsx`, `jspdf/jspdf-autotable`) to generate and trigger downloads for Excel, PDF, and individual CSVs.
  - **Custom Reports (`CustomReports.jsx`):** (Licence required) Provides UI for selecting data types, date range (`TimePeriodSelectorWidget`), and output format. Sends parameters to backend (`POST /reports/.../custom`) which generates and returns the file for download. Access gated by Sidebar based on licence status.

- **Dynamic Dashboard (`src/modules/DynamicDashboard/`):**
  - **Components:** `DynamicDashboard.jsx` (main), `AddWidgetDialog.jsx`, `TimePeriodSelectorWidget.jsx`, `FilterWidget.jsx`, `WidgetWrapper.jsx`, numerous specific `widgets/*.jsx`.
  - **Functionality:** Fetches all required data types. Manages widget layout (`react-grid-layout`) and active widgets state, persisted to `localStorage`. Filters data based on time period and user filters (`useMemo`). Renders widgets within `WidgetWrapper` (provides frame, title, remove button). Widgets use `Recharts` for visualization. Allows adding/removing widgets. Quick Add widget allows direct data entry.
  - **API:** Fetches data from multiple endpoints (`/expenses`, `/income`, `/budgets`, etc.) on load. Quick Add uses `POST /expenses`, `POST /income`.

3.3 Desktop Shell Design (Electron)

Electron bridges the gap between the web-based frontend and a native desktop experience, managing the application window and the backend process.

3.3.1 Main Process (`electron.js` - Detailed Interaction)

The main Electron process script (`electron.js` or similar) orchestrates the application lifecycle and communication.

- **Window Creation:** Uses Electron's `BrowserWindow` API to create the main application window (`mainWindow`). Key options include setting dimensions (`width`, `height`), frame visibility (`frame: true/false`), and importantly, `webPreferences`. Recommended secure defaults `nodeIntegration: false` and `contextIsolation: true` are used, preventing the renderer process (React app) from directly accessing Node.js APIs. A preload script could be specified via `preload: path.join(__dirname, 'preload.js')` to expose specific Node.js/Electron APIs to the renderer securely if needed, although direct backend communication via HTTP is the primary method here.
- **Loading UI:** Detects the environment (`isDev = process.env.NODE_ENV === 'development'`).
  - In development, loads the URL from the React development server (e.g., `mainWindow.loadURL('http://localhost:3000')`).
  - In production (packaged app), loads the local `index.html` file from the React build output directory using `mainWindow.loadURL(`file://$path.join(__dirname, '../build/index.html')`)` (adjust path based on build structure).
- **Backend Process Spawning:** This is a critical function handled within the main process.
  - **Path Determination:** Locates the backend executable. In development, it finds the Python interpreter and the path to `seta-api/app/main.py`. In production, it constructs the path to the packaged backend executable (e.g., `seta_api_server.exe` or `seta_api_server`) located within the application's resources directory (copied there by Electron Builder via `extraResources`). The exact path might involve `app.getAppPath()` and relative paths.
  - **User Data Path:** Determines the appropriate directory for user configuration and the local database using Electron's `app.getPath('userData')`. This provides a standard, OS-specific, writable location.
  - **Spawning:** Uses Node.js's `child_process.spawn` function.
    * Passes the command (python interpreter or direct executable path) and any necessary arguments.
    * Crucially, passes the determined user data path to the backend process via the `env` option: `env: { ...process.env, SETA_USER_DATA_PATH: userDataPath }`. This ensures the backend uses the correct location for its configuration and local database.
  - **Process Monitoring:** Attaches listeners to the spawned backend process's `stdout` and `stderr` streams to log backend output for debugging. Listens for the `close` or `exit` event to know when the backend terminates.
- **Lifecycle Management:** Handles Electron app lifecycle events:
  - `ready`: Triggered when Electron has finished initialization. Used to create the `BrowserWindow` and spawn the backend process.
  - `window-all-closed`: Quits the application when all windows are closed (standard behavior, except on macOS).
  - `activate`: Re-creates the main window if the app is activated when no windows are open (macOS specific).
  - `will-quit`: Triggered just before the application quits. A handler is attached here to explicitly kill the backend child process (`backendProcess.kill()`) to ensure it shuts down cleanly when the Electron app closes.

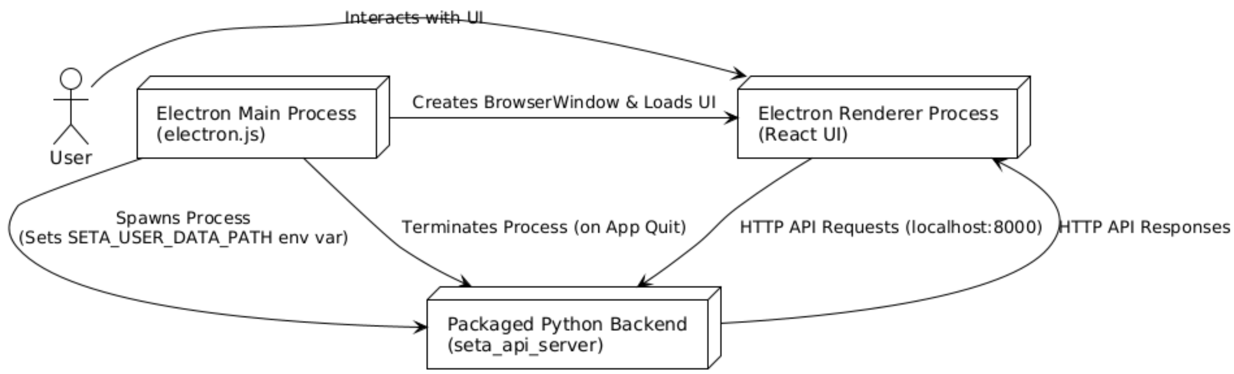

Figure 5: Electron Process Interaction

3.3.2 Packaging (`electron-builder`)

Electron Builder is used to package the application for distribution.

- **Configuration:** Managed primarily via the `"build"` key within the frontend's `seta-ui/package.json` file, or optionally via a dedicated `electron-builder.yml` file. Specifies application ID, product name, output directories, target platforms, icons, etc.

- **Build Script:** The `npm run electron:build` script (defined in `package.json`) typically first executes the React build (`react-scripts build` or similar) to generate the static frontend assets, and then invokes the `electron-builder` command line tool. Arguments like `-mac`, `-win`, `-linux` can specify target platforms.

- `extraResources`**:** This configuration directive within the `"build"` key is essential for including the pre-compiled backend. It instructs Electron Builder to copy files or directories into the packaged application's resources folder. It must be configured correctly to copy the *entire* output directory of the native PyInstaller build (e.g., from `../seta-api/dist/seta_api_server`) into a known location within the app package (e.g., `app/backend`). The path used in `electron.js` to spawn the backend must correspond to this packaged location. Example configuration snippet:

```
"build": {
  "extraResources": [
    {
      "from": "../seta-api/dist/seta_api_server",
      "to": "app/backend",
      "filter": ["**/*"]
    }
  ],
  // ... other build settings ...
}
```

- **Platform Targets:** Configured to build distributables for macOS (`.dmg`), Windows (`.exe` installer), and Linux (`.AppImage`). Electron Builder handles the specifics for each platform.

- **Code Signing:** For distribution, code signing is necessary to avoid security warnings and enable features like auto-updates. Electron Builder supports signing via configuration options and environment variables (e.g., `CSC_LINK`, `CSC_KEY_PASSWORD` for Windows/Mac; Apple ID credentials for macOS notarization). These are typically stored as secrets in the CI/CD environment (e.g., GitHub Actions secrets) and accessed during the automated build process. Implementation requires obtaining valid certificates. (we hope CUHK support us to buy certificates for code sign)

- **Auto Update:** Electron Builder generates platform-specific update manifest files (e.g., `latest.yml`, `latest-mac.yml`) which are uploaded alongside the installers during the release process. These files are used by `electron-updater` (if integrated into the Electron app) to check for and download updates.

4 Database Design

The application's data persistence layer relies on a relational database managed via SQLAlchemy ORM.

- **ORM (SQLAlchemy):** Provides a high-level, object-oriented interface to the database, abstracting away raw SQL queries for most operations. It maps Python classes defined in `app/models.py` to database tables and object attributes to columns. This improves code readability, maintainability, and portability between supported database systems.

- **Schema Definition (`app/models.py`):** This file contains the canonical definition of the database structure. It uses SQLAlchemy's declarative mapping style:

  - Defines classes inheriting from `Base`.
  - Specifies table names (`__tablename__`).
  - Defines columns with types, primary/foreign keys, constraints (unique, nullable), and indexes.
  - Establishes relationships between tables using `relationship()`, including back-references (`back_populates`) and cascade behaviors (`cascade`, `ondelete`).

  A visual representation of the schema is provided in Figure **??**.

- **Supported Databases:**

  - **SQLite (Default):** Chosen as the default for its simplicity and suitability for a single-user desktop application. It requires no separate database server installation, storing the entire database in a single file (`seta_local.db`) located within the application's user data directory. The schema is automatically created by SQLAlchemy (`Base.metadata.create_all`) on the first run if the file doesn't exist.
  - **PostgreSQL (Optional):** Supported as an alternative for users who prefer or require a more robust, server-based relational database. This option requires the user to have a running PostgreSQL instance and provide a valid connection string (via environment variable `DATABASE_URL` or the custom setting in `seta_config.json`). Schema management for PostgreSQL typically relies on Alembic migrations.

- **Migrations (Alembic):** The project includes Alembic configuration (`alembic.ini`, `alembic/` directory) for managing incremental changes to the database schema over time. While not strictly necessary for the auto-created SQLite database, Alembic is the standard tool for applying schema updates in a controlled manner, especially for PostgreSQL deployments. Developers would use Alembic commands (`alembic revision -autogenerate`, `alembic upgrade head`) to create and apply migration scripts when the SQLAlchemy models in `app/models.py` are modified. The Alembic environment needs to be configured correctly to point to the target database URL.

5 Deployment and Build Process

SETA is designed for deployment as a cross-platform desktop application, integrating the Python backend and React frontend into a single package. The process relies on native builds for each target platform, automated via GitHub Actions.

- **Backend Build (PyInstaller):**
  - The Python backend (`seta-api`) is compiled into a platform-specific executable bundle using PyInstaller.
  - PyInstaller analyzes the Python code (`app/main.py` and its imports), collects dependencies, and bundles them with the Python interpreter into a distributable format (typically a folder containing the executable and supporting files - `--onedir` mode).
  - The `seta_api_server.spec` file provides configuration for PyInstaller, including specifying the main script, application name, included data files (`-add-data` for the `app` module, `alembic` folder, `alembic.ini`), and options like `-noconsole` (for Windows GUI apps).
  - **Native Requirement:** This PyInstaller build step **must** be executed on the target operating system (macOS, Windows, Linux) to ensure the bundled executable and libraries are compatible with that OS.
- **Frontend Build (React):**
  - The React frontend (`seta-ui`) is built into static assets (HTML, CSS, JavaScript bundles) using the standard React build process (e.g., `npm run build` invoking `react-scripts build`). This produces an optimized set of files ready for deployment.
- **Electron Packaging (Electron Builder):**
  - Electron Builder takes the static React build output and bundles it with the Electron runtime environment.
  - **Backend Integration:** Crucially, it copies the *natively pre-compiled* backend executable bundle (output from PyInstaller) into the application's resources directory using the `extraResources` configuration in `package.json`.
  - It generates platform-specific installers and packages (`.dmg` for macOS, `.exe` installer for Windows, `.AppImage` for Linux).
  - **Native Requirement:** Like the backend build, the Electron packaging step should ideally be performed on the target OS to ensure correct handling of native dependencies and packaging conventions.
- **GitHub Actions Automation (`.github/workflows/release.yml`):**
  - **Trigger:** The workflow is triggered automatically when a Git tag matching the pattern `v*.*.*` (e.g., `v1.2.0`) is pushed to the repository.
  - **Parallel Jobs:** It runs three parallel jobs, one on each major OS runner provided by GitHub Actions (`macos-latest`, `windows-latest`, `ubuntu-latest`).
  - **Build Steps per Job:**
    1. Check out the source code corresponding to the pushed tag.
    2. Set up Node.js and Python environments.

3. Install backend Python dependencies (`pip install -r requirements.txt`).
4. Run PyInstaller to build the backend **natively** on the runner's OS.
5. Install frontend Node.js dependencies (`npm install` in `seta-ui`).
6. Run Electron Builder (`npm run electron:build - -[mac|win|linux]`) to package the frontend, embedding the **native** backend built in the previous step.
7. Upload the resulting packaged application file(s) (e.g., `.dmg`, `.exe`, `.AppImage`, `.yml` update file) as a build artifact specific to that job/OS.

– **Release Creation Job:** A final job runs *after* all three build jobs have succeeded.

  * It downloads the build artifacts (the packaged apps for all platforms) from the completed build jobs.
  * It uses an action like `ncipollo/release-action@v1` to create (or update) a GitHub Release associated with the triggering Git tag.
  * It uploads all the downloaded artifacts to the assets section of that GitHub Release, making them available for users to download.
  * Optionally generates release notes based on commits since the last tag.

This process ensures that users receive natively compiled versions of the application optimized for their operating system, all managed through an automated CI/CD pipeline triggered by version tagging.

6 Security Considerations

Security was considered throughout the design and implementation process, focusing on protecting user data and authentication credentials.

- **Password Security:** User passwords are never stored in plaintext. Upon signup or password change, the backend hashes the provided password using a strong, standard hashing algorithm (`hashlib.sha256`) before storing the hash in the database (`users.password_hash`). During login, the provided password is hashed using the same algorithm and compared against the stored hash (`verify_password` function).

- **Token Security:**

  – Email verification and password reset tokens are generated using Python's `secrets.token_urlsafe(32)`, which produces cryptographically secure, URL-safe random strings.

  – These tokens are stored temporarily in the database (`users.verification_token,users.password_reset_to` with unique constraints and indexes.

  – Verification tokens are invalidated (set to `None`) immediately after successful email verification.

  – Password reset tokens have an explicit expiry timestamp stored in the database (`users.password_reset_token_expiry`), typically set to 1 hour after generation. The backend verifies the token's existence and checks if it has expired (`expiry < datetime.now(timezone.utc)`) before allowing a password reset. The token is invalidated after successful use.

- **API Access Control & Validation:**

  – **CORS:** `CORSMiddleware` in FastAPI is configured to restrict which origins can make requests to the API, preventing unauthorized web pages from interacting with it.

  - **Input Validation:** FastAPI's integration with Pydantic automatically validates incoming request bodies and parameters against the defined schemas (`BaseModel` subclasses). This prevents many common injection-style attacks and ensures data integrity before it reaches the business logic. Invalid requests result in automatic 422 Unprocessable Entity responses.
  - **Authorization:** Most API endpoints operate within the context of a specific `user_id`. Backend logic consistently filters database queries by the authenticated user's ID (e.g., `db.query(models.Expense).filter(models.Expense.user_id == user_id).all()`) to prevent users from accessing or modifying data belonging to others.

- **Licence Key Validation:** Licence key checks are performed exclusively on the backend (`validate_licence_key`, `require_active_licence` dependency). The validation logic (checking against the `ACCEPTED_LICENCE_KEYS` set) is not exposed to the frontend. While the current keys are provided freely, this server-side check establishes the pattern for potential future secure licence validation.

- **Dependency Management:** Using standard package managers (`pip` with `requirements.txt`, `npm` with `package.json/package-lock.json`) helps manage dependencies. Regularly updating dependencies is crucial to patch known vulnerabilities. Tools like `pip-audit` or `npm audit` can be used to scan for known issues in dependencies.

- **Filesystem Access:** The backend's filesystem interaction is primarily limited to reading/writing the configuration file (`seta_config.json`) and the local SQLite database file (`seta_local.db`) within the designated user data directory (`SETA_USER_DATA_PATH`). Electron's default security settings (`contextIsolation: true`, `nodeIntegration: false`) restrict the frontend renderer process from arbitrary filesystem access.

- **Code Signing (Limitation):** As noted, the application builds are currently unsigned. Implementing code signing for macOS and Windows builds is a necessary future step to enhance user trust, bypass OS security warnings, and enable seamless auto-updates via `electron-updater`. This requires obtaining developer certificates and integrating signing into the build process (Electron Builder configuration and CI/CD secrets).

7 Limitations and Future Considerations

While SETA provides a robust set of features, certain limitations exist, and several areas offer potential for future enhancements.

- **Multi-Currency Support:** Currently assumes a single currency (often hardcoded or defaulted to 'USD' in places). Proper multi-currency support would require storing currency information per account/transaction and handling exchange rates, adding significant complexity.

AI Tools Usage

Gen AI tools(aistudio.google.com) is used to generate clean and readable latex code and PlantUML diagram for better representation.
Most of the Source code is generated by AI(aistudio.google.com), and is carefully reviewed, commited and modified by all our group mates