ChatGPT

PocketSage Overhaul & Refactor Plan

Objective: Rebuild PocketSage from the ground up with a simpler, more maintainable architecture while aligning with all requirements in the Specs. We will strip down the current Flet-based UI and implement a minimal **desktop-first, offline** web app using Flask and Jinja2, consistent with the original project vision [1] [2]. The new design prioritizes clarity and simplicity (inspired by HomeBank's lightweight approach [3]), focusing on core flows without unnecessary complexity. All data will remain local (SQLite database under an `instance/` folder), with optional SQLCipher encryption for privacy [4] [5]. We will retain the underlying user/auth logic (single local "guest" user by default) and admin operations, but **simplify the experience for a single-user desktop context** (no login screen, auto-use a default profile [6]). Development will be **schema-first**, defining a clear SQLModel data model (with Alembic migrations for schema changes [7]) before building out each module's functionality. The application will be structured into modular Flask blueprints for each domain: **Ledger**, **Habits**, **Liabilities**, **Portfolio** (optional), **Admin**, and **Reports** [8]. We outline below a comprehensive roadmap for each module, followed by specific file-level TODO markers to guide implementation. The plan addresses current known issues (e.g. form buttons not updating the UI, backup/export reliability) and includes stubs for future features mentioned in the PDFs. The tone and detail here reflect a top-tier engineering plan, ensuring **readable code, UI clarity, and alignment with requirements**.

Ledger Module (Transactions & Budgets)

The **Ledger** is the heart of the app, handling income/expense tracking and budgeting. We will implement a streamlined transaction register and basic budgeting features per user requirements (UR-1, UR-2, UR-3) [9]. The UI will be modeled after classic personal finance ledgers (e.g. HomeBank/Mint), with a table of transactions and filtering controls. Specifically, we'll create a **Ledger** Flask blueprint with routes for listing transactions, adding/editing entries, and managing categories/budgets. Key functionality:

- **Transaction CRUD:** Implement full Create/Read/Update/Delete for transactions. Each transaction will have date, description (memo), category, and amount (with positive for income, negative for expenses). Form inputs will be validated server-side (e.g. required fields, numeric amount) – aligning with **FR-8** for input validation [10]. Upon form submission, the app will redirect back to the ledger list view so the new/updated transaction is immediately visible (this ensures the UI refreshes after a button press, fixing the current issue where added entries do not appear without manual refresh). We'll display clear inline error messages for any validation issues (meeting **NFR-17** on form feedback clarity [11]).

- **Ledger UI & Layout:** Design a simple, scrollable table view for transactions, showing key columns (Date, Description, Category, Amount, maybe Account). A summary bar above the table will show the current period's totals (e.g. income, expenses, net balance) and budget status. Users can filter the list by date range and category via a filter bar or dropdown (similar to Mint's filtering by time/category [12]). For MVP, all transactions can be shown in one combined ledger (we'll assume a single primary account), but the data model supports multiple accounts for future use. The UI should be clean and uncluttered – e.g. similar to HomeBank's ledger register which emphasizes simplicity and ease of use

[3]. We won't implement multi-account UI now (we'll just seed a default account and assign all transactions to it), but the code will be structured to allow adding account filters later.

- **Categories Management:** Provide the ability to manage transaction categories (per **FR-9** [10]). We will seed a set of common categories on first run (e.g. "Food", "Rent", "Salary") and allow the user to add/edit/delete categories via the UI. This could be a simple form or modal accessible from the Ledger page. All transactions link to a category (optional for income entries). We'll ensure category names are unique per user and possibly use a slug field for consistency (the model already includes `name` and `slug` [13]). Category CRUD operations will be simple and update the list of categories used in transaction forms (e.g. a dropdown in the transaction form). This addresses UR-1 and UR-3's need for categorization and budgeting by category [9].

- **Budgets & Alerts:** Implement basic monthly budgeting (supporting **FR-13** [14] and UR-3). For MVP, we will allow the user to set a single overall monthly budget or per-category budgets (the exact scope depends on the spec: since FR-13 mentions "budget definitions and threshold notifications", we will at least support defining a monthly budget amount for the overall expenses, or possibly for each category). The app will then compute the total expenses per month and indicate if the budget is exceeded. In the UI, this can be shown as a progress bar or a simple highlight (e.g. green if under budget, red if over). We will **stub** the "threshold notification" part – e.g. log a warning or prepare a placeholder in code where in the future we might trigger a notification if the budget limit is crossed (FR-52 is future scoped for budget alerts [15]). For now, a visual cue on the ledger or reports page will suffice. All budget data will be stored in a new `Budget` SQLModel (if not already defined) linking a user (and possibly category) to a monthly amount.

- **Rollup Summaries:** Calculate and display rollups such as monthly income, expense, and net cash flow, as required by **FR-10** [10]. The ledger page will show these summaries for the current month by default (and allow the user to switch months or date ranges). We'll implement a utility in the service layer to compute these totals efficiently each time the ledger view is requested (or cache them per month). This summary also supports UR-2 (viewing cash flow summaries) [9].

- **Spending Chart:** Integrate a simple spending visualization directly into the ledger view, fulfilling **FR-11** [14]. For example, a pie chart of expenses by category for the current month can be generated with Matplotlib and displayed as an image on the ledger page. We'll use an accessible color palette (meeting **NFR-18** on chart readability [11]) – for instance, using high-contrast colors or patterns for each category slice. The chart should update whenever transactions change. This gives users an immediate visual breakdown of spending, addressing UR-2's call for spending visualizations [16]. The chart generation logic will be implemented in a dedicated service (e.g. `reports_service` or `ledger_service`) and can be reused on the Reports page.

- **CSV Import/Export:** Provide the ability to import transactions from a CSV and export transactions to CSV for backup (aligned with **FR-30** on import and UR-7 on export [17] [18]). The import function will be designed to be **idempotent** – avoiding duplicate entries by using a unique `external_id` or hash for each transaction (as per **NFR-11** [19] and FR-30). We'll include a field `external_id` in the Transaction model (already present [20]) to store an identifier from imported data and use it to skip or update existing records on re-import. Exports will produce a CSV of all transactions (or filtered by date) that the user can save. We will add UI buttons for "Import CSV" and "Export CSV" on the ledger page, using a file picker for import and prompting download for export. Any errors or successes from these operations should be clearly communicated (e.g. a flash message or a results page), to satisfy UR-7's requirement for export and feedback [17]. Additionally, we'll prepare the backend to support **file system watcher** integration for CSV imports (optional **FR-32** [18]): i.e., a background thread (using `watchdog`) that can detect when a new CSV file is placed in a watched directory and automatically trigger an import. We will **stub** this in the code (the `watcher.py` service exists with TODOs [21]) – not enabled by default, but ready to use if configured (this addresses UR-35 in the requirements for an automated file watcher actor [22]).

- **Optimistic Locking & Data Integrity:** To prevent lost updates and ensure data integrity (important for multi-user or future sync scenarios), we plan to implement optimistic locking on critical records in the ledger, per **FR-12** and **NFR-10** [14] [23]. Specifically, we will add a `last_updated` timestamp and/or a version number column to the `Transaction` model. This field will be checked during updates – if an update finds that the record was modified by another process (version mismatch), it can warn or reject the change. While multi-user editing is not a current scenario (desktop app is single-user), this lays groundwork for robustness and satisfies the requirement for auditability (each transaction's updates are timestamped – **NFR-23** [24]). We will mark in the model definition where to include the version field and ensure the update logic in the service checks it. (This may be a low-priority item for MVP, but a TODO will be added for completeness.)

- **HomeBank-Style Ledger Best Practices:** Based on patterns from HomeBank/GnuCash, our ledger will adopt a **single-window** approach with a navigation panel and the transaction register on the same screen (HomeBank's wishlist even suggests a unified window with nav tree [25]). We will keep navigation simple (top nav links to each module) and within the ledger page, we'll avoid overly complex interactions. For example, adding a transaction will likely be a straightforward form (possibly a modal) instead of an in-line edit grid, reducing user error. We also consider Mint's approach – allowing quick filtering and a clear view of spending by category and time period [12] – to ensure the ledger isn't just raw data but provides insight. The design goal is an intuitive flow: the user opens the app and immediately sees their recent transactions and summary for the month, with obvious controls to add a new transaction or adjust budgets. This resonates with the **desktop-first, easy-to-use** philosophy (fast startup, minimal clicks) [26].

Planned Implementation Tasks (Ledger):

- [ ] **Data Model:** Ensure the `Transaction` SQLModel has all needed fields. *Add* `updated_at` *(timestamp) and* `version` *(int) fields* to `Transaction` for optimistic locking and audit trail (ref. NFR-10) [27]. Also confirm `Category` model is present and link transactions to categories (this is already in place [20]).

```
# TODO: In src/pocketsage/models/transaction.py, add fields for
last_updated timestamp and version number for FR-12 / NFR-10.
```

- [ ] **Blueprint & Views:** Create a `ledger` Flask blueprint (e.g. `src/pocketsage/routes/ledger.py`) with routes: `/ledger` (GET – list transactions, with filter params), `/ledger/new` (GET/POST – add form), `/ledger/<id>/edit` (GET/POST – edit form), and `/ledger/delete` (POST – deletion). Use Jinja2 templates (e.g. `templates/ledger_list.html`, `templates/`

`ledger_form.html`). Include context data for totals and charts. After adding/editing a transaction, redirect to the list view so the UI updates immediately.

```
# TODO: Implement Flask routes for Ledger (list, add, edit, delete) with
form handling and filtering (FR-7, FR-8).
```

- [ ] **Template (Ledger List):** Build a Jinja2 template to display the transactions table and filter controls. Use a loop to show each transaction row, formatting date and amount (e.g. red for negative amounts). Show a summary section on top (month totals) and the spending chart image (as an `<img>` tag). Provide buttons/links for "Add Transaction" and maybe "Manage Categories" or "Import/Export CSV".

```
<!-- TODO: Design ledger_list.html to show transactions table, filters, and
summary (UR-2, FR-10). -->
```

- [ ] **Template (Transaction Form):** Create a form for adding/editing transactions (`ledger_form.html`). Fields: date (default today), description, amount, type (income/expense toggle or infer from sign), category (dropdown), and perhaps account (default account). Show validation errors near fields (NFR-17) [11] . On successful submit, route back to ledger list.

```
<!-- TODO: Create ledger_form.html with WTForm fields for date,
description, amount, category, etc. (FR-8) -->
```

- [ ] **Category Management UI:** Implement a simple interface to add/edit categories. This could be a section on the ledger page or a separate modal/dialog. Possibly use a small form with just "Name" (and optional type: income vs expense) and an Add button. The category list can be shown for reference with edit/delete options. Ensure that deleting a category is prevented if it's in use by transactions (or require reassignment).

```
# TODO: Provide category CRUD endpoints or modal in ledger page (FR-9).
```

- [ ] **Budget Logic:** Implement budget storage and checking. Create a `Budget` model (if not existing in `models/budget.py`) with fields like `month`, `amount`, and maybe `category_id` (null for overall budget vs specific category budgets). In the ledger service, add a function `check_budget(status)` that compares total expenses vs budget and returns a flag or message if over budget (FR-13) [28] . Display the budget status on the UI (e.g. "Budget: $X of $Y used" and highlight if over). For now, we assume a single overall budget amount can be set via an Admin or Settings page (or even a config), with finer per-category budgets as a later enhancement. Mark a TODO for adding notifications when thresholds are crossed (FR-52 future) [15] .

```
# TODO: Add budget model and service to track monthly budgets and usage
(FR-13).
# TODO: In budget service, stub a notification trigger when budget
threshold exceeded (FR-52).
```

- [ ] **Filtering & Pagination:** In the ledger list route, support filtering by date range and category. Use query parameters or form inputs for filters. Leverage SQLModel queries to filter transactions by the current user, date, category, etc. Implement pagination if the list grows large (maybe limit 50 per page with "Next" button), to satisfy performance needs (NFR-6 expects 10k transactions load under 3s [29] – pagination will help). Mark any advanced filtering (text search, tag filtering) as future enhancements.

```
# TODO: Implement query filters (by date span, category) and pagination in
ledger list view (FR-7).
```

- [ ] **Transaction Import/Export:** Utilize the existing import utility (likely `services/import_csv.py`) to handle CSV parsing. Add a route `/ledger/import` to accept an uploaded file (via Flask's file upload) and pass it to the import service. Ensure the import service performs deduplication using `Transaction.external_id` (the PDF NFR-11 emphasizes no duplicate transactions on re-import [19]). After import, flash a success or error message indicating how many records were added or skipped. For export, add a route `/ledger/export` that queries all transactions (or those in current filter) and returns a CSV file (set appropriate response headers for download). Also integrate these with the Admin "Backup" feature (which will call all modules' export routines – see Reports/Admin sections).

```
# TODO: Wire up CSV import (POST file) and export (GET download) routes for
transactions (FR-30, UR-7).
# TODO: In import service, ensure idempotent upsert by external_id to avoid
dupes (NFR-11).
```

- [ ] **Spending Chart Generation:** Implement a function to create the spending breakdown chart. Likely in `services/reports.py` or `services/ledger_service.py`, use Matplotlib to generate a pie chart of expenses by category for a given month. Save it to a PNG in a temp folder or serve directly as response (e.g. by encoding to base64 or saving under `instance/charts/`). We will ensure the chart uses distinct colors (perhaps a predefined palette) and is accessible (NFR-18) [11] . This chart function will be called whenever the ledger page loads (or possibly cached per month). Mark a TODO to refine the chart (e.g. legends, dynamic sizing) as needed.

```
# TODO: Implement ledger spending chart (Matplotlib) and save PNG for
embedding (FR-11).
```

- [ ] **UI Refresh Fix:** Ensure that after adding/editing a transaction or performing an import, the ledger page data is up-to-date. In Flask, this is naturally handled by re-querying on page load, so we inherently resolve the Flet UI bug where the list didn't refresh. We will, however, double-check that any client-side caching (if introduced) is invalidated. Mark a note in the form handler code to emphasize that the redirect or re-render must happen to reflect changes.

```
# TODO: After form submission, redirect or reload ledger page to reflect
new data (fixes UI update issue).
```

Habits Module (Daily Habits Tracking)

The **Habits** module lets users track daily habits and view streaks/visualizations, fulfilling requirements UR-4 and UR-5 (habit tracking influencing spending) and the detailed functional requirements FR-14 through FR-18 [30] [31]. We will create a Flask blueprint for habits (`/habits` routes) with pages to list habits, toggle daily completion, and view stats. The focus is on simplicity: a list of habits with an easy "done/not done" checkbox for today, plus ability to add new habits and see streak information.

- **Habit CRUD:** Implement creation, reading (list), updating (e.g. renaming or archiving), and (if needed) deletion of habits (though deletion might be replaced by archiving to preserve history). The Habit model already exists with fields for name, description, reminder_time etc. [32]. We'll provide a form to add a new habit (fields: name, optional description or reminder time). New habits default to active (`is_active=True`). In the habits list view, each habit will have a toggle control for today's completion. We won't implement complex schedules beyond daily cadence for now (the model has a `cadence` field for potential weekly habits, but UR-4 implies daily habits primarily [33]). Archiving (FR-18) will be implemented as setting `is_active=False` on a habit; archived habits won't show in the active list, but can be listed under an "Archived Habits" section and possibly reactivated.

- **Daily Toggle & Entries:** For daily tracking, we use the `HabitEntry` model (which records a habit ID and date for each completion) [34]. When the user toggles a habit as done for today, the app will create a HabitEntry for today's date (if not already exists). If toggling off (undo), it will delete that day's entry. This logic will be in the habits service or directly in the route handler. We'll ensure this is atomic and idempotent per day. Immediately after toggling, the UI should update the habit's streak count (the number of consecutive days up to today). This addresses **FR-14** (persist entries with streak recalculation) [30] and UR-11 (quick daily toggle) [35].

- **Streak Calculation:** Implement a function to compute the current streak and longest streak for each habit. This can be done by querying the HabitEntry table for recent dates. For efficiency, we might maintain a cached streak count on the Habit model or compute on the fly. Given our scale is small (daily entries), on-the-fly calculation is fine: e.g. fetch all entries for the habit, sort by date, then compute consecutive days sequences. We will update the UI to show the **current streak** and possibly the **best streak**. This fulfills FR-14 (streak logic) [30]. After each toggle, we recalc the streak and update the display via page refresh or dynamic update (in Flask, a simple approach is to reload the habits page portion; we might use a small AJAX call for a better UX, but full page refresh is acceptable MVP).

- **Habit List UI:** The habits page will list all active habits in a table or card list. Each habit row shows: name, maybe a short description, a "Today" checkbox or toggle button, current streak (e.g. " 5 days" icon), and possibly an archive button. The design should be minimal: e.g. a table with columns [Habit | Streak | Done Today | Actions]. Checking/unchecking the "Done Today" box will submit a form (or an AJAX call) to toggle that habit's entry. We'll use clear visual feedback – for instance, if a habit is done today, maybe highlight the row or show a checkmark icon. This addresses UR-11/UR-12 (quick toggle and immediate streak update) [36] . We will also consider adding a small **calendar heatmap** for each habit to visualize the history (FR-16) [30] . For MVP, a simplified approach could be a one-line mini calendar (e.g. last 7 days of M/T/W/... with colored dots or checkmarks). However, since the requirements explicitly call for weekly/monthly heatmap visualization, we plan to generate a chart or graphic showing habit completion over time. A straightforward implementation: use Matplotlib to create a 30-day heatmap (e.g. a 5x6 grid for last month, with colored squares for days done vs not done). This could be shown on the habits page or consolidated in Reports. We'll at least stub this feature now: possibly generate a static image for each habit's recent activity (like a 30-day streak calendar) and show it in the habit detail or as a tooltip. If time permits, implement one and reuse it.

- **Reminders (stub):** The spec mentions optional habit reminders (UR-10 optional, FR-17) [37] [31] . For now, we will not implement actual notification dispatch (no external emails or system notifications to keep the app offline and simple), but we'll include a stub in the code. The Habit model has a `reminder_time` field [38] ; we will allow the user to set a time (HH:MM) if they want a daily reminder. We then plan a placeholder in the scheduler: e.g. if a habit has a reminder_time, we could log a message at that time or mark that a notification would have been sent. In the future, this could integrate with an OS notification or email if configured. We will annotate this in code as a TODO (so QA/DevOps knows where to hook in actual notifications later, satisfying FR-17's future scope [30] and FR-51 in the Notifications section [15]).

- **Habit Influence on Spending:** One interesting aspect mentioned is that habits may "influence spending" or have notes linking to spending trends [39] . This sounds like an Advisor Lite feature: e.g. if you have a "No Coffee" habit and you keep it, maybe show that your spending on "Coffee" category decreased. Implementing this fully is complex and likely beyond MVP scope. However, we can **stub** an integration point: for example, allow associating a category or note with a habit (like linking a habit to a category it might impact). We won't process it now, but we can document that such a link exists. The *Project Overview* hints at "habit→spend notes" [39] , which suggests at least showing a note that, for instance, after 30 days of a habit, the user saved $X. We will not implement the calculation now, but might include a dummy placeholder text (e.g. "This habit might be saving you money!" if appropriate). This falls under "Advisor Lite" which is optional – so a low priority, but noted.

- **UI Integration:** Add a link in the navigation bar to the Habits page. Also, ensure the Dashboard (if any) or Reports can show a summary of habits (like number of habits done today). For example, on a dashboard, we might display "Habits done today: X/Y" to give a quick overview (this aligns with UR-4 and the Dashboard summary mentioned in the TODO.md [40]).

Planned Implementation Tasks (Habits):

- [ ] **Data Model:** Review `Habit` and `HabitEntry` models. They look sufficient (HabitEntry acts as join table of Habit and date) [34] . We might add an index on HabitEntry `occurred_on` if not already, for query speed. Also possibly add a `longest_streak` field on Habit to cache the all-time best streak (not strictly needed, can compute when needed). Mark a TODO to compute and store longest streak when appropriate (maybe as part of a daily job or when habit is toggled off after a streak break).

```
# TODO: Consider caching longest_streak in Habit for quick access (optional
optimization).
```

- [ ] **Blueprint & Routes:** Create a `habits` blueprint (`src/pocketsage/routes/habits.py`). Routes: `/habits` (GET – list active habits, with form to add new and controls to toggle), `/habits/new` (POST – create habit), `/habits/<id>/toggle` (POST – mark done/undone for today), `/habits/<id>/archive` (POST – set is_active False), and maybe `/habits/archived` (GET – view archived habits). The toggle route will handle creating or deleting a HabitEntry for today. Use Flask's session or context to identify the current user's habits.

```
# TODO: Implement Habits blueprint routes for listing, adding, toggling,
and archiving habits (FR-14, FR-18).
```

- [ ] **Habit List Template:** Create `templates/habits_list.html`. This page will list each habit with: Name (possibly hyperlink to more details if needed), Current Streak count, and a checkbox or button for "Done Today". Also include a form at the bottom or top to add a new habit (name, optional description/reminder). If using a simple approach, we might handle the toggle via form submission (each checkbox could be a form). We will ensure that after toggling, the page refreshes and the streak count updates accordingly. Also, if a habit was not done today yet, the checkbox is unchecked; if done, it's checked. Provide a way to archive (could be a small "Archive" button next to each habit, which posts to archive route). Possibly also a link to view habit history or a small inline chart.

```
<!-- TODO: Build habits_list.html to display habits, streaks, and today's
toggle (UR-11, FR-14). -->
```

- [ ] **Streak Calculation Service:** Implement a helper in the habits service (e.g. `services/habit_service.py`) to calculate streaks. For a given habit, fetch all entries (or at least the most recent contiguous sequence up to today). An efficient method: if today is completed, count backwards day by day until a day is missing or habit start. If today is not done, streak is 0 (or break). Longest streak can be computed by scanning the full entry list once – but for MVP, we might compute it on the fly each time or update it when entries are added/removed. We will ensure this logic is well-tested for edge cases (e.g. no entries at all => streak 0, breaking streak after an inactive day, etc.).

```
# TODO: Implement compute_streak(habit_id) that returns current_streak and
longest_streak (FR-14).
```

- [ ] **Toggle Handler:** In the toggle route, use the streak function. For example, after toggling, we might recalc and flash a message like "Habit X: current streak = Y days". Or simply rely on the refreshed page. Also handle concurrency (though unlikely – single-user app).

```
# TODO: In habits toggle route, insert or delete HabitEntry for today and
recompute streak.
```

- [ ] **Visualization (Heatmap):** Implement a rudimentary habit completion visualization for FR-16 [30]. Perhaps create a function to generate a small calendar image for a habit. Alternatively, for simplicity, in the template we can visually represent last 7 or 30 days using colored Unicode blocks or emoji (✔ for done, ☐ for not done). If using Matplotlib, we could draw a grid (dates vs completion). Time permitting, we'll do a 7-day week x 5-week calendar for last month. This image can be shown when clicking on a habit or directly in the list if space allows. We'll add a placeholder image or HTML table for now.

```
# TODO: Generate a 30-day heatmap or calendar for habit completion (FR-16)
– possibly as a placeholder image.
```

- [ ] **Reminders Stub:** In the application scheduler (see Admin section for scheduler setup), add logic to handle habit reminders. For each habit with `reminder_time` set, we might schedule a daily check at that time. Since actual notifications are out of scope, we can log a message or set a flag that a reminder would be triggered. This satisfies the design of FR-17 in principle without user-facing functionality, and leaves room for future implementation (FR-51) [15].

```
# TODO: In scheduler, stub a daily job to handle Habit reminders at
specified times (FR-17, future feature).
```

- [ ] **Archive and Reactivate:** Implement the archive route to set `is_active=False`. Archived habits won't appear on the main list. Provide a view or section to list archived ones (maybe on a separate page or a collapsed section in the habits page). A reactivate button would set `is_active=True` and bring it back. This covers FR-18 [41].

```
# TODO: Implement archive/reactivate logic for habits (FR-18).
```

- [ ] **Link to Ledger (future):** Add a note in the code about potential integration point with ledger spending. E.g., in the Habit model or service, note that if a habit is linked to a spending category, one could correlate streaks with spending changes. This is not implemented now but aligns with the product vision (Advisor Lite).

```
# TODO: [Future] Link habits to spending patterns (e.g., associate habit
with a category to analyze impact).
```

Liabilities Module (Debts & Payoff Planner)

The **Liabilities** module helps users manage debts (loans, credit cards) and plan payoffs, addressing UR-5 (manage liabilities, see payoff projections) [33] and the detailed features FR-19 through FR-24 [42]. We will build a **Debts** blueprint with routes for viewing all liabilities, adding a new liability, recording payments, and viewing payoff simulations (snowball/avalanche). This module will enable users to see how long until they are debt-free under different strategies.

- **Liability CRUD:** Allow users to add, edit, and remove liabilities (loans or debts). The Liability model includes fields for name, balance, APR, minimum payment, due day, etc. [43]. In the UI, we'll provide a form to input these details for each debt account (e.g. "Chase Credit Card – $5,000 balance, 18% APR, $100 min payment"). Liabilities will be listed on the main Debts page with summary info: name, current balance, interest rate, and a status (maybe next due date or simply active). Users can edit these if needed (e.g. update an interest rate or correct a balance). Deletion will likely only be allowed if no payments recorded or simply we can allow deletion if the user wants (with a confirmation).

- **Snowball/Avalanche Calculation:** Implement the two debt payoff strategies – **snowball** (pay smallest balance first) and **avalanche** (pay highest interest first), per FR-20 and FR-21 [42]. We will create a service function that, given the list of liabilities and an extra payment amount (or just using current minimums), projects a payoff schedule. This function will produce outputs like: for each month going forward, which debt to pay extra, remaining balances, total interest paid, and a projected debt-free date. We will incorporate logic to avoid infinite loops (e.g. if interest is higher than payment, etc.) and ensure once a debt is paid off, its freed payment is applied to the next debt (rollover). The output will allow us to display summary stats: e.g. "Under Snowball, you will be debt-free by Jan 2030, paying $X in interest. Under Avalanche, debt-free by Nov 2029, interest $Y." We'll also generate a **payoff timeline** (FR-22) [44] – likely a line chart showing combined balance over time declining to 0, possibly with markers when each debt is paid. We will use Matplotlib to create this chart (X-axis time, Y-axis total balance or individual balances). For MVP, a simple combined timeline is fine. This chart will be displayed on the Debts page or in Reports.

- **Liability List UI:** The Debts main page will list all liabilities in a table format. Each row: Name, Balance, APR, Minimum Payment, and maybe a computed "Payoff Date (if only min payments)". We will also show totals (total debt across all liabilities). At the top or bottom, include buttons or links to simulate payoff strategies. For example, a toggle or radio buttons to select Snowball vs Avalanche, and a "View Payoff Plan" button. When clicked, we will display the results (possibly on the same page below or on a separate detail page). The results include: projected debt-free date and total interest (for chosen strategy), maybe a small table listing each debt's payoff month, and the timeline chart. This addresses UR-18 (debt-free date and timeline) [45].

- **Recording Payments:** Allow the user to input actual payments made, beyond just updating the balance manually. FR-23 requires recording actual payments and recalculating balances [44]. We will implement a mechanism to record a payment: e.g. on the Debts page, each liability could have a

"Record Payment" button. Clicking it opens a form to input amount and date of payment. Submitting will create a transaction in a **PaymentHistory** or directly adjust the Liability's balance. Since the model doesn't yet define a separate Payment entity (though a comment in model suggests it as a future addition [46]), we have two approaches: (1) simply subtract the payment from the liability's balance and log it in a basic way (like appending to an in-memory log or printing – not ideal), or (2) leverage the existing Transaction model by creating a transaction with a special category like "Debt Payment" and linking it via `Transaction.liability_id` (the Transaction model has a field for liability linkage [47]). Approach (2) is cleaner: we can treat debt payments as transactions (transfer of money out). We will do the following: when a user records a payment, update the Liability's balance (balance = balance - amount), create a Transaction entry (amount = -X, category = maybe an "Interest" or the liability's own category, or we create a new Category "Debt Payment", and set `liability_id` to the debt). This way, the payment is reflected in both the debt module and the ledger. We'll note that this integration is optional for MVP; we can stub it if time is short, but ideally we implement it to keep data consistent. We will ensure after a payment, we recalc the payoff projections with the new balance.

- **Export/Amortization:** Provide an option to export the payoff schedule or current liabilities to CSV (related to FR-23 and FR-24). For each liability, users might want an amortization schedule (especially for loans) – e.g. a CSV listing each payment period, interest, principal, remaining balance. We can generate a basic schedule for the chosen strategy and allow download. This might be a stretch for MVP, but we'll include a TODO. At minimum, ensure the data is available for export via the general backup (which will include liabilities and perhaps a summary). FR-24 is about displaying the debt-free projection and schedule summary [44], which we will do on screen; providing a downloadable version covers the export angle.

- **UI/UX Considerations:** We will keep the Debts UI straightforward. HomeBank doesn't have a built-in debt planner like this, but other financial apps (e.g. Mint's goals or undebt.it) have similar flows. The main complexity is making the payoff simulation understandable. We'll include textual explanations: e.g. "Using the Snowball method (pay smallest balance first with any extra funds), your debts will be paid off by <date>." Perhaps also highlight which strategy is faster or saves interest. For MVP, we won't do a detailed comparison, but we'll implement both calculations so the user can choose. We'll ensure the UI updates when switching strategy (maybe using JavaScript to swap the chart and summary without full reload, if possible; otherwise a quick form submit is fine).

- **Edge Cases:** If a user's minimum payments are not sufficient to ever pay down a debt (e.g. interest accrual outpaces payments), our calculation should detect that and warn (to avoid infinite loop). We can put a check: if in an iteration the balance doesn't decrease or time exceeds some large limit, break and inform the user that payoff isn't achievable under current inputs. This ties into FR-23's note to guard against tiny payments causing infinite loops [48].

Planned Implementation Tasks (Liabilities):

- [ ] **Data Model & Repos:** Confirm the `Liability` model structure (it's already defined with necessary fields [43]). We may add a separate model for PaymentHistory in the future, but for now, use transactions for recording payments. We should also ensure the `Liability.__tablename__` is set (it is, "liability" [49]). Possibly add a model for payoff strategy or schedule if needed, but more likely we compute on the fly.

```
# TODO: Consider adding a PaymentHistory model or utilize Transaction for
debt payments (FR-23).
```

- [ ] **Blueprint & Views:** Create `debts` blueprint (`src/pocketsage/routes/debts.py`). Routes: `/debts` (GET – list all liabilities and summary), `/debts/new` (GET/POST – add a liability), `/debts/<id>/edit` (GET/POST), `/debts/<id>/delete` (POST), `/debts/<id>/pay` (GET/POST – record a payment), and `/debts/plan` (GET or POST – calculate payoff plan for chosen strategy). The list view will call the planner service to get default strategy info (maybe Snowball by default).

```
# TODO: Implement Debts blueprint with routes for listing liabilities,
adding/editing, recording payments, and viewing payoff plans (FR-19,
FR-23).
```

- [ ] **Debts List Template:** Create `templates/debts_list.html`. It will show a table of current debts. For each liability: Name, Balance, APR, Min Payment, and actions (Edit, Delete, Record Payment). At the bottom or top, include controls for payoff simulation: e.g. a form with a radio selection (Snowball or Avalanche) and a button "Calculate Payoff". Also show results from the last calculation: e.g. "Snowball payoff date: Dec 2028 (total interest $N)". Include the payoff chart image on the page. If the user changes the strategy selection, submitting will refresh the results.

```
<!-- TODO: Build debts_list.html to list liabilities and display payoff
summary and chart (UR-18, FR-22). -->
```

- [ ] **New Liability Form:** Create `templates/debt_form.html` for adding/editing a liability. Fields: Name, Initial Balance, APR (%), Minimum Payment, Due Day (if needed, or skip), and perhaps an optional extra payment field (for user's planned extra monthly contribution, though that can just be considered in the plan rather than stored). Validate that name is provided, balance is numeric, APR is percentage, etc. On submit, create the Liability and redirect to list.

```
<!-- TODO: Create debt_form.html for adding/editing a liability (FR-19). --
>
```

- [ ] **Payoff Planning Logic:** Implement functions `calculate_snowball(liabilities)` and `calculate_avalanche(liabilities)` in a service module (e.g. `services/debt_planner.py`). These will simulate month-by-month payments. Input: list of liabilities (with balances, APR, min payment) and an optional total monthly budget for debt (if user has extra money to allocate – we can assume they pay at least the sum of mins). If no extra payment specified, we use exactly the mins (then payoff date is when each naturally ends). For snowball: each month, pay all mins, allocate any extra to the **smallest balance** remaining. For avalanche: allocate extra to **highest APR**. We output: total months to payoff, total interest, and possibly a list of payoff events (like [(debt_name, payoff_date), ...]). Also generate data points for chart: e.g. total remaining balance each month. Implement careful interest calculation: interest accrues monthly = APR/12 * balance. Subtract payments. We might simplify compounding by assuming monthly compounding with the balance at month's start (close enough). This satisfies FR-20 and FR-21 logic [42].

```
# TODO: Implement debt payoff calculation for snowball & avalanche
strategies (FR-20, FR-21).
```

- [ ] **Payoff Timeline Chart:** Using the output of the planner, produce a Matplotlib line chart (X-axis months, Y-axis total balance). Mark the debt-free point on the chart. Optionally, we could stack each debt to show how individual balances drop, but that might be too detailed. A single line of total debt works. Save this chart as `debt_payoff.png`. This addresses FR-22 [44].

```
# TODO: Plot payoff timeline chart as PNG for the selected strategy
(FR-22).
```

- [ ] **Payment Recording:** In the `/debts/<id>/pay` route, handle recording a manual payment. The form will ask for Amount (and Date, default to today). On submit: find the liability, reduce its balance by Amount. Also, create a new Transaction entry (with category perhaps "Debt Payment" or the liability's own name, and link via `liability_id`). If the liability's balance goes to 0 or below, mark it as paid off (we could leave it but perhaps set a flag or simply note that balance=0 means paid). Provide feedback to user ("Payment recorded, new balance $X"). This covers FR-23's core (reconcile balances with actual payments) [44]. If the payment exceeds remaining balance (user fully pays off), we can set balance to 0 and maybe archive the liability (or just leave it with 0 balance).

```
# TODO: Handle payment form submission: update liability balance and log a
transaction for it (FR-23).
```

- [ ] **Debt-Free Date & Summary:** After each plan calculation, compute the date when all balances reach zero. Also compute total interest paid across all debts in the plan. Display these in a summary paragraph (e.g. "Debt-free by July 2027, Total Interest: $5,200"). This is fulfilling FR-24 (display debt-free projection date) [44]. If multiple strategies are available, allow the user to see both (maybe by switching radio options).

```
# TODO: Display projected debt-free date and total interest for the chosen
strategy (FR-24).
```

- [ ] **Export Schedule (optional):** Add a route or option to download an amortization schedule CSV for a selected strategy. This CSV could list each month's payments and balances per debt. Mark this as a low-priority TODO to implement if time allows, since FR-24 mainly requires display which we have.

```
# TODO: (Optional) Provide CSV export of payoff schedule for each liability
(stretch goal related to FR-23/24).
```

- [ ] **Integration with Reports:** Ensure that the Reports module (discussed below) can include a section on debts – e.g. the payoff timeline chart and a summary of total remaining debt. Possibly the unified export will include current liability info and maybe the chart image. Mark a note in Reports to incorporate debt data.

```
# TODO: Integrate debt summaries into the Reports page (FR-41/42 connection
to debts).
```

Portfolio Module (Investments – Optional)

The **Portfolio** module is optional (UR-6 and UR-7 mention it as an opt-in) [50] . It allows users to import and view investment holdings and see allocation and performance at a basic level. We will implement this module in a way that it can be disabled if not needed (e.g. controlled by a configuration flag). If enabled, a **Portfolio** blueprint will handle routes under `/portfolio`. Key features to implement in alignment with FR-25 through FR-29 [51] :

- **Holdings CRUD & Import:** Users can add investment holdings manually or via CSV. A **Holding** represents a stock or asset position (the SQLModel `Holding` is defined with fields: symbol, quantity, average price, market_price, etc. [52]). We will create a page listing all current holdings with their details. The user can add a holding through a form (symbol, quantity, purchase price, current price, maybe date acquired). Alternatively, the user can import a CSV file containing multiple holdings. FR-25 requires accepting a CSV upload of holdings [51] . We will implement a CSV import similar to transactions: allow the user to choose a file (perhaps in a specific format with columns like Symbol, Quantity, Cost Basis, Current Price). The import function will parse this and for each record, create or update a Holding entry. We need to avoid duplicates – likely by symbol or by a combination (symbol + account). The plan is to treat *symbol* as a unique key for now (assuming one account for simplicity). If a holding for that symbol already exists, we might update it (e.g. adjust quantity or average price) or skip. We will log or display how many holdings were added/updated. This covers FR-25 (CSV upload) and partially FR-26 (persist holdings and snapshots – we interpret snapshots as maybe storing historical values; not in MVP beyond current values) [51] .

- **Portfolio List UI:** The `/portfolio` page will list each holding in a table: Symbol, Quantity, Average Price, Market Price, Market Value (Quantity * Market Price), and Unrealized Gain/Loss (market value minus cost basis). We will compute these on the fly for display (FR-28 gain/loss) [53] . At the bottom, show totals (total portfolio value, total gain). Provide controls to add a new holding (a form) and to import CSV (a file input). Also possibly a button to export the holdings to CSV (FR-29) [53] . If the user has multiple accounts for investments, we might group by account, but since multi-account is not fully supported in UI, we'll assume one account or treat all holdings together.

- **Allocation Chart:** One of the main features is an allocation visualization – typically a pie or donut chart showing percentage of portfolio per holding (or per asset category, but we have no category concept, so per holding is fine). FR-27 calls for a donut chart of allocation [51]. We will implement this using Matplotlib: a pie chart of each holding's market value share of total. If there are many holdings, perhaps just show top N and group others as "Other" for clarity. The chart can be displayed on the Portfolio page (and also included in Reports). Ensure an accessible color palette as well (NFR-18) [11]. The chart updates whenever holdings data changes (after import or adding a holding).

- **Manual Updates & Price Refresh:** The MVP will not integrate live market data (no external API, consistent with offline requirement [26]), so any price updates must be manual. The user can edit a holding to update its `market_price`. We'll provide an Edit form for each row or a quick inline edit for current price. In the Roadmap TODOs, adding live price or manual inputs was noted as a future improvement [54]. We'll explicitly *not* implement live price fetch (to keep offline), but we make it easy for the user to input the current price from their own research. We might add a "Refresh All" button that simply resets all market prices to default (not very useful), so likely skip that.

- **Account Support (Optional):** The data model includes an `Account` relation for holdings [55], meaning a user could have multiple brokerage accounts. For MVP, we assume a single account or we ignore the account grouping. We will seed a default portfolio account for the user so the foreign key is satisfied. If in the UI we want to show accounts, we might just list the account name (like "Main Portfolio") as a header. This is low priority, as UR-6/UR-7 didn't explicitly require multi-account. Mark a TODO that multi-account portfolio management could be added later.

- **Export & Backup:** Implement the ability to export the current holdings to CSV (FR-29) [53]. This can be a simple route that returns a CSV with all holdings fields. Also ensure that the Admin "Backup" (in next section) includes portfolio data (e.g. including portfolio CSV or including it in the DB backup).

- **Performance Consideration:** Portfolio data sets will typically be small (maybe tens of holdings), so performance is not a big issue. If users were to track many assets or frequent price changes, we might consider storing historical snapshots (time series) for performance tracking (this is hinted by FR-26 "allocation snapshots time series" [51] and stretch goals about time-series tracking). We will not implement time series now, but we note that `Holding` might be considered a snapshot itself at a certain date. If needed, we could create a separate model for daily snapshots of total portfolio or asset values. We add a placeholder in code as a reminder for this extensibility.

Planned Implementation Tasks (Portfolio):

- [ ] **Data Model:** Review `Holding` model (already defined) [52]. Ensure `__tablename__` is set ("holding" is set) and relationships to `Account` and `User` are in place (they are [55]). If any issues exist (the TODO.md mentioned fixing Holding ↔ Account mapping [56]), verify that and correct if necessary (e.g. ensure `Account.holdings` relationship exists and works, which it does in `account.py` [57]).

```
# TODO: Verify Holding <-> Account relationship is functioning, adjust if
needed (from TODO notes).
```

- [ ] **Blueprint & Routes:** Create `portfolio` blueprint (`src/pocketsage/routes/portfolio.py`). Routes: `/portfolio` (GET – list holdings and show summary/chart), `/portfolio/new` (GET/POST – add holding), `/portfolio/<id>/edit` (GET/POST – edit holding, e.g. update price or quantity), `/portfolio/delete` (POST – remove holding), `/portfolio/import` (POST – handle CSV file upload), and `/portfolio/export` (GET – download CSV of holdings). We will guard all these routes such that they only execute if portfolio module is enabled (perhaps controlled by a config flag or simply always available but with no data if user never uses it).

```
# TODO: Implement Portfolio blueprint for listing holdings, add/edit,
import CSV, and export CSV (FR-25, FR-29).
```

- [ ] **Portfolio Template:** Create `templates/portfolio.html` to display the holdings table and controls. The table columns: Symbol, Quantity, Avg Price, Market Price, Market Value, Gain/Loss. Compute Market Value = Quantity * MarketPrice, Gain = (MarketPrice - AvgPrice) * Quantity. We can compute these in template or pass precomputed values from the route. At the top, show total portfolio value and total gain (summing all holdings). Include a section for the allocation chart (e.g. an `<img src="/portfolio/chart.png">` if we have an endpoint for the chart image, or embed base64). Controls: “Import CSV” (file input with form posting to /portfolio/import) and “Export CSV” (link to /portfolio/export). Also an “Add Holding” form inline or a separate page.

```
<!-- TODO: Build portfolio.html to list holdings, show totals, and display
an allocation chart (FR-27, FR-28). -->
```

- [ ] **CSV Import Handler:** In the import route, parse the uploaded CSV. We can use Python’s csv module or pandas if available, but for simplicity csv. Expect columns: symbol, quantity, avg_price, market_price (we will document required format). For each row, normalize the symbol (uppercase ticker for stocks), find if a Holding with that symbol exists for the user. If exists, decide merge strategy: either update the holding (perhaps recalc average price if adding more quantity? but that gets complex, likely the CSV is a full snapshot so we might replace the entry). For now, simplest: if exists, we **skip or update** (we’ll update the quantity and prices to whatever the CSV says, assuming CSV is intended as a full refresh). If not exists, create a new Holding. Provide feedback: e.g. “3 holdings added, 2 updated”. Use the same `external_id` concept if applicable, but holdings are uniquely identified by symbol so that suffices as key. Mark that in a multi-account scenario, we’d include account info in CSV and key by (account, symbol).

```
# TODO: Implement CSV import for holdings: parse file, upsert Holding
entries (FR-25, NFR-11 approach to avoid dupes).
```

- [ ] **CSV Export Handler:** Gather all holdings for current user, and output a CSV with headers (Symbol, Quantity, AvgPrice, MarketPrice, Value, Gain). This fulfills FR-29 [53]. This can be used in the backup zip as well.

```
# TODO: Implement export of holdings to CSV for download (FR-29).
```

- [ ] **Allocation Chart:** Implement a function to generate the allocation chart (pie chart) using Matplotlib. The input will be the list of holdings with their market values. Create a pie chart with slices proportional to each holding's value. Label slices with symbol (and maybe percent). If there are many small slices, we might combine or skip labels to keep it readable (accessible colors are important, we'll use a set of distinguishable colors or Matplotlib's default but ensure contrast). Save as `portfolio_allocation.png`. This is the fulfillment of FR-27 [51]. The chart will be shown on the portfolio page and also likely on the Reports page.

```
# TODO: Generate portfolio allocation pie chart (Matplotlib) and save to
static image (FR-27).
```

- [ ] **Gain/Loss Calculation:** The portfolio page should display unrealized gain or loss per holding and total. The calculation: `(market_price - avg_price) * quantity`. We will compute that and possibly color it red/green in the UI (green for gain, red for loss). This meets FR-28 (compute gain/loss) [53]. Ensure we handle negative quantities or missing market_price gracefully (e.g. if market_price not provided, treat current value as unknown or equal to avg_price).

```
# TODO: Calculate unrealized gains for each holding and display in UI
(FR-28).
```

- [ ] **Account Grouping (Future):** If we plan to support multiple investment accounts, mark a TODO to implement a filter or grouping by account on the portfolio page. For now, maybe just show account name if needed. The Account model is there [58], and user could theoretically add multiple, but no UI for it now. Possibly seed one account called "Portfolio" and assign all holdings to it.

```
# TODO: [Future] Support multiple accounts in portfolio (group holdings by
account, add account management UI).
```

- [ ] **Periodic Snapshot (Future):** As a stretch idea, note that FR-26 mentions time series snapshots of allocation [51]. We won't do this now, but we can add a cron job or admin action later to capture a snapshot of holdings values at a point in time (for historical tracking). We'll leave a comment or stub indicating this can be implemented (e.g. a `PortfolioSnapshot` model or re-using the Holding model with timestamps).

```
# TODO: [Future] Implement periodic snapshots of portfolio value for
performance tracking (FR-26).
```

Admin Module (Admin & Settings)

The **Admin** module provides maintenance features, configuration, and overall application management for the power user or internal admin. This includes seeding demo data, backup and restore, managing user accounts, and system settings (encryption, scheduling). We will implement an **Admin** blueprint with routes under `/admin` for these tasks, aligning with FR-37 to FR-40 [59] and other system requirements.

- **Demo Data Seeding:** Implement a route/button to **seed demo data** (FR-37) [60]. This will populate the database with sample entries for each module (transactions, categories, habits, entries, debts, holdings) so that a new user or a demo session can have data to explore. The current codebase likely has a `scripts/seed_demo.py` or similar. We will integrate this into the app: pressing "Seed Demo" will call a service function (e.g. `admin_tasks.seed_demo()`) which inserts a canned set of data. We must ensure this can run idempotently – if it runs again, it shouldn't duplicate data (perhaps it wipes existing user data or uses upsert logic). Given this is mostly for demo, a simple approach is to wipe the DB and re-add (with user confirmation). Alternatively, we check if sample categories exist etc., and only add missing. The requirement UR-9 and FR-37 emphasize a restore from backup or seed for learning [61] [60], so we'll definitely include this.

- **Backup (Export) & Restore:** Provide an **Export/Backup** function that bundles all user data into a single archive (FR-38 and FR-50) [59] [62]. This likely means: export all tables as CSV and all charts as PNGs, compress into a zip. The MVP approach: when user clicks "Backup", the server will generate a zip file containing:

- CSV of transactions, CSV of categories, CSV of habits & entries, CSV of liabilities, CSV of holdings.
- Any relevant charts (perhaps latest spending pie, payoff chart, allocation chart, habit heatmap) as images.

- Possibly a JSON or text README of what's included. We will then either download this zip to the user's chosen location or save it under `instance/exports/` and inform the user. (If using Flask as a web app, likely we just prompt download). We should also implement a **Restore** function to import from such a backup. Restore might be more complex (we have to read the zip, parse files, and replace the database contents). For MVP, we can stub the restore: possibly just mention that to restore, one might manually replace the database file or similar. But since UR-9 specifically says restore from backup for learning [61], we should attempt a basic restore: e.g. if user uploads a backup zip, we can drop all current data and load from the CSVs. We'll add caution (and likely require confirmation because it overwrites data). If implementing, we will reuse import logic from each module to re-ingest the CSVs. If not fully implementing now, we'll note that as a next step and ensure a TODO is placed. The backup should be stored in a safe location – our config's DATA_DIR likely has a `backups/` folder. We'll keep the last few backups (the existing code mentions retention of last 5 archives [63]). We will implement rotation (delete older backups) to satisfy that requirement.

- **User Management:** In a desktop app context, multi-user is not a priority, but we have to keep the logic. The current design uses a **local user** with role "admin" by default and optionally a "guest" user for ephemeral sessions [64] [65]. We will maintain the `User` model and the notion of roles ("user" vs "admin"). The Admin page can list users (if we ever create more than one) and allow actions like password reset or deletion (mentioned as a TODO in the Roadmap [66]). For MVP, we likely only have one user (the local admin), so user management UI is minimal. But we will include an indication of the current user and role (like showing “Logged in as: [username] (Admin)” somewhere, even if not logging in actually). If we keep the guest concept (for quick start without data persistence), we can provide a “Reset Guest Data” action that purges the guest user data (the current code had something like `purge_guest_user` [67]). We’ll incorporate that if relevant: e.g. if the user runs in a no-login guest mode and then wants to actually create a user, they can clear the guest data.

- **Settings (Theme, Encryption):** The Admin or Settings page should allow toggling some application settings:

- **Theme** (light/dark): The current app supported a theme toggle [6] . We can persist this in a settings table or config file. Provide a UI control to switch theme (this is minor but nice).
- **Encryption**: If SQLCipher is available, allow the user to encrypt the database. This might be tricky to do on the fly. At minimum, indicate whether the DB is encrypted or not (maybe on the Settings page, “Database: Encrypted [Yes/No]”). Possibly a button to “Enable Encryption” which if clicked will re-create or migrate the DB under encryption. However, that might be beyond MVP; more realistically, encryption is decided at app startup (with an env var). We will implement support such that if `POCKETSAGE_ENCRYPT=1` in .env, we use an SQLCipher URI or execute `PRAGMA key` on connection (this fulfills **FR-3** [7] and **NFR-1** [68] for encrypted DB). But we won’t try to toggle it live through the UI for now – we will just show the status.

- **Auto-Backup**: Possibly allow enabling automatic backups (e.g. a checkbox “Backup my data every night”). The scheduler already has logic to schedule nightly backup if configured [69] . So expose a setting for `auto_backup_enabled`. If turned on, the scheduler (APScheduler) will perform the backup each day at 3 AM by default [70] . The user should be informed where backups go (likely `instance/backups/auto/`). We’ll add a note on the Admin page about this.

- **Other Environment Config**: Use .env for things like data directory, etc. This is more backend, but we ensure the app reads `POCKETSAGE_*` variables at startup (FR-2) [71] . Not much UI for this, but maybe display current config values for debugging.

- **Logging & Monitoring:** Ensure that structured logging is set up (FR-5) [72] . We have `setup_logging` likely in code. We will continue to use a rotating file handler (writing logs to `instance/logs/app.log` with rotation) as was intended. The Admin page might have a section showing recent log messages or an option to download the log for troubleshooting. Not strictly required by spec, but FR-39 does say “Admin UI progress and error feedback” [59] – which includes showing errors from operations. We will for example catch exceptions in seed/backup and show a message on the UI (“Backup failed: [reason] – see logs for details”).

- **CLI Commands:** Provide command-line interface commands for key admin actions (FR-40) [59] . Since we use Flask, we can add custom Flask CLI commands, e.g. `flask pocketsage seed-demo`, `flask pocketsage export-data`, etc. This is mainly for developer/ops use (like packaging scripts). We will implement these using Flask CLI or click in our `app.py` or a separate commands module. This allows running these tasks without the GUI, aligning with 12-factor operability (NFR-24, NFR-25) [24] . We’ll note to update README accordingly.

- **Packaging with PyInstaller:** Although packaging is more devops than admin, we plan for distribution via PyInstaller one-file executables (FR-47) [73] . We will create a spec or script for

PyInstaller that includes the Flask app, and ensure data files (like default .env or templates) are bundled. We should test the build on major OS (NFR-19 cross-platform) [11]. This won't be done in the UI, but is part of the project deliverable. We'll include instructions or a script (`scripts/build_desktop.[sh|bat]`) as mentioned in TODOs [74]. For the purpose of this plan, just note that after development, we will verify PyInstaller packaging.

- **Scheduler (Background Jobs):** We already have an APScheduler-based background scheduler in the code [75] [76]. We will integrate this into the Flask app (likely started on app startup). The scheduler will handle tasks like the nightly backup (as coded) and potentially others (log rotation is already scheduled at 4 AM [77], reminders stub will be added). Ensure that when the app exits, the scheduler stops (the Flet app had on_close; in Flask, if running in a console, we might not worry, but if packaged, maybe handle signals). This meets FR-6 (init background tasks) [72] and the scheduling requirements (UR-31, FR-31 for scheduling jobs) [78]. We will keep these tasks mostly headless (no UI needed, except maybe indicating on the Admin page that auto-backup is on, etc.).

- **Fix Export/Backup Feedback:** Specifically, the user noted backup/export failures not giving feedback. In our new implementation, after the user clicks "Backup", we will try to generate the zip and then either download it or show a link to it. If it fails (exception), we'll flash an error message. We'll also log the error with details. This should address the current problem where perhaps nothing was shown. We will test the backup thoroughly with different data volumes to ensure it succeeds or at least fails gracefully (NFR-26 safe failure handling) [79].

- **Encryption & Security:** On startup, we will enforce that a non-default secret key is set for Flask (NFR-2) [80]. If `FLASK_SECRET_KEY` is still the default placeholder, we will refuse to run (or print a critical warning) – instructing the user to set a proper secret in the .env. Also, if encryption is desired, we ensure the database URL uses `sqlite://` with SQLCipher driver or apply PRAGMA key. Since the spec emphasizes no external network usage (NFR-3) [80], we will double-check that our code doesn't attempt any external API calls (and we won't include any telemetrics). The only potential external call might be if someone tried to fetch stock prices – we are not doing that.

- **Modularity & Extensibility:** Ensure that all these admin and config features are done in a clean, modular way. The blueprint approach already compartmentalizes functionality (NFR-21 modular architecture) [81]. For example, the Portfolio module can be entirely turned off if not needed. We'll document toggles (like an env flag to disable portfolio or watcher).

Planned Implementation Tasks (Admin & Core):

- [ ] **Blueprint & Template:** Create `admin` blueprint (`src/pocketsage/routes/admin.py`). The `/admin` page (template `admin.html`) will serve as a control panel. It will display sections for **Data Management** (seed, backup, restore), **Settings** (theme, auto-backup toggle, encryption info), **User Management** (list users, maybe just current user info), and **System** (version, environment). Buttons: "Seed Demo Data", "Backup Data", "Restore from Backup". If restore is implemented, that may be a form file upload. For seed and backup, those trigger actions immediately. Under Settings, have checkboxes or toggles for things like "Auto Backup nightly" and "Dark Theme" (the latter could also be a UI toggle in a navbar). We'll handle those via simple form posts or via a bit of JavaScript to toggle and call an endpoint.

```
# TODO: Implement Admin blueprint with routes for seed_demo, backup,
restore, and settings updates (FR-37, FR-38, FR-39).
```

- [ ] **Demo Seed Function:** Integrate the demo seeding logic. Possibly use existing `scripts/seed_demo.py` via import, or rewrite within a service function. This should create a set of categories (e.g. "Rent, Food, Salary, Entertainment"), a bunch of transactions (mix of income and expenses over a couple months), a few habits (with some entries to show streaks), a couple of debts with balances, and a few portfolio holdings. Essentially populate each module's tables so the app looks "used". This helps demonstrate the features (important for the planned Thanksgiving demo [82]). The seed should be idempotent: we can choose to **wipe and reseed** to avoid duplication. So likely the seed function will *delete existing user data* (except maybe keep user account) before inserting new. That could be surprising if a user accidentally clicks it on real data – so guard it with a confirmation dialog ("This will erase current data and replace with sample data. Continue?"). In the Admin UI, we'll handle that either via a JavaScript confirm or a second step page.

```
# TODO: Implement seed_demo() service to populate sample ledger, habits,
debts, portfolio data (FR-37).
# TODO: Warn/confirm before seeding as it may overwrite real data.
```

- [ ] **Backup (Export) Implementation:** Write a service function `create_backup(user_id)` that gathers data from all modules and zips it. Steps: create a temp directory, call each module's export logic (we will reuse the `/ledger/export`, etc., or directly query models to write CSVs). Also generate charts: it would be nice to include charts in the backup zip – e.g. generate a fresh spending pie, debt payoff line, etc., and include those PNGs. Use Python's `zipfile` to bundle everything with a timestamped filename like `PocketSage_backup_2025-11-25.zip`. Save it to `instance/exports/` (ensuring the dir exists). We will maintain at most 5 most recent as noted: after creating, check the directory and remove oldest if >5 files [63]. The route for "Backup" can either send the file to the user (download) or just notify that backup was saved (maybe better to allow user to choose location, but in a web app that's limited – by default it would download to user's browser). Since this is a desktop scenario, perhaps we intend the user to pick a folder on their machine. We could use an `<input type="file" webkitdirectory>` to choose a folder, but that's awkward in web. Instead, we might just trigger a normal file download which the user then moves to desired location. We will clearly inform where the backup went (if saved on disk or just downloaded). Possibly we offer both: a direct download and also save a copy under instance.

```
# TODO: Implement backup export service: generate CSVs for all tables and
bundle with charts into a zip (FR-38, FR-42, FR-50).
```

- [ ] **Restore Implementation (Basic):** Provide an endpoint `/admin/restore` that accepts an uploaded backup zip. This will be a potentially destructive action: it should replace current data with data from backup. Implementation: on restore request, we likely flush all user-specific tables (transactions, categories, habits, entries, liabilities, holdings, etc.) and then read each CSV from the zip and insert rows. We must create objects with proper foreign keys (e.g. categories before transactions to link category_id). This is complex to get 100% right easily; for MVP we may implement a simpler approach: e.g. if the backup zip contains a full SQLite database file, we could just replace our `pocketsage.db` with it. That might be simplest if we choose to backup by copying the actual DB file instead of CSVs. Indeed, an alternative backup strategy is: if not concerned about encryption differences, we could simply copy the SQLite database file and any relevant files (like an images folder). However, the requirements specifically mention CSV and PNG, so they intend a human-portable format [63]. So we'll stick to CSV. We will attempt a partial restore: e.g. require that the backup be from the same schema version (we can embed a version in the backup name or a manifest). If mismatch, we refuse or caution. To implement: after upload, we extract zip to a temp folder, then open each CSV and feed to the respective import logic (we can reuse the import functions we have, e.g. for transactions and holdings). This ensures using the same deduping rules etc. We then regenerate any derived data (like budgets or streaks if needed) or just rely on data consistency. Because of time, we might mark detailed restore as a TODO, but ensure at least the structure is there.

```
# TODO: Implement basic restore from backup: allow user to upload a backup
zip, then import its CSV contents (FR-50).
# TODO: Add safety checks and confirmation, as this will overwrite existing
data.
```

- [ ] **Auto-Backup Scheduling:** Utilize the `BackgroundScheduler` already set up [76]. The code checks a user setting `auto_backup_enabled` [83] and if true, schedules `create_backup` daily at 3:00 AM [69]. We will ensure that this integration remains after refactor: e.g. when the Flask app starts, after initializing DB, we call something like `scheduler = create_scheduler(ctx, auto_start=True)` (like in Flet code [84]). We will adapt that to Flask (maybe via an app factory pattern or a background thread). If not straightforward, we can start the scheduler in a separate thread on app launch. We'll expose a UI control (checkbox "Auto-backup daily at 3 AM") that sets the setting. This satisfies UR-31 (scheduled jobs) [78] and FR-31. Also ensure log rotation job remains (it's scheduled at 4 AM as per code [77]).

```
# TODO: Initiate background scheduler on app startup, with auto-backup job
if enabled (FR-6, FR-31).
```

- [ ] **User Roles & Guest Mode:** In the auth service, they have `ensure_local_user` and `ensure_guest_user` [85] [86]. We will follow that approach: by default, on app launch, if no users exist, create a "local" user with admin role (no password, or a default one). The app will automatically log in this user (since we are skipping login UI). This matches the login-free desktop mode from the current TODO [6]. We keep the possibility of adding real login later, but for now, everything runs under this one user context. In Admin UI, we might show a section "User: local (Admin)" and possibly allow setting a password or creating additional users in future. The requirement UR-8 was full offline usage with privacy [87], so not needing accounts is fine. We will mark in code that multi-user login can be reintroduced in future (the auth TODO already says that [88]). Also ensure that any references to user in data models use this default user's ID. The seed/demo functions should attach data to the correct user. If using guest for demo, perhaps the seed clears guest user's data (there was `reset_demo_database` call in `purge_guest_user` [89]). We'll align with that logic as needed.

- [ ] **Secret Key Enforcement:** In the Flask app configuration, check `app.config['SECRET_KEY']`. If it's the default (like "dev" or empty), print error and exit (or throw). This addresses NFR-2 (no default secret) [24]. In a packaged app, the secret can be generated or set in .env. We note this in documentation.

- [ ] **Logging Setup:** Use Python logging to output JSON-formatted logs to file (maybe using the `logging_config.get_logger` in code). Ensure logs rotate (the scheduler's job at 4 AM does log rotation cleanup of old files [90]). We will keep logs local only (NFR-4, no telemetry) [80]. Possibly provide a way on Admin page to download the log file for debugging.

- [ ] **Testing & CI (DevOps):** (Included here for completeness) – We will set up unit tests for critical functions (calculations, imports) to satisfy FR-43/44/45 and ensure new code works. Also ensure `ruff` and `black` are used (NFR-12 code formatting) [91] – we can include a pre-commit config. The CI pipeline (likely GitHub Actions) will be configured to run tests and packaging (FR-46) [92]. These tasks, while not visible to the end-user, ensure maintainability and quality.

- [ ] **Packaging:** Create PyInstaller spec or script that collects the Flask app and all resources (including .env, templates, static files). Test the built binary on Windows, macOS, Linux (at least smoke test) to satisfy FR-48 and NFR-19 [11]. Also document how to run it.

  ```
  # TODO: Prepare PyInstaller packaging script and verify one-file executable
  generation (FR-47, FR-48).
  ```

- [ ] **Documentation & Help:** Update README.md and a Help section in the UI (perhaps an `/help` route) to reflect the new usage: mention that no login is required, how to use each module, how to backup/restore, etc. This aligns with making the app user-friendly (NFR-16 simple navigation, NFR-17 clear errors, which we are addressing) [93]. Possibly include an "About" in Admin with version info.

By implementing the above, the Admin module will give the user confidence in controlling their data and the app's operations, wrapping up any loose ends of the refactor.

Planned Implementation Tasks (Admin & Settings):

- [ ] **Admin Template UI:** Develop `templates/admin.html` as a dashboard for admin tasks. Include sections:
  - **Data Management:** "Seed Demo Data" (with confirm), "Backup Data" (download or save notice), "Restore Data" (file upload with confirm). After each action, show a status message (success or error).
  - **Settings:** "Theme: Light/Dark" toggle (could use a bit of JS to toggle immediately; also persist in user settings). "Auto-Backup: On/Off" checkbox – triggers enabling/disabling in settings (and scheduler). Show "Data Directory: [path]" and "Encryption: Enabled/Disabled" info.

    - **User Info:** "Current User: <username> (<role>)". If we allow, a button to "Reset Password" or "Delete User" could be there (for future; for now maybe hidden).
    - **System:** show app version, maybe environment (dev/prod), and a link to logs or to open the logs directory.
    - Possibly include a "Logout" button if multi-user was on (not needed now). Keep this page simple and form-driven for actions.

    ```
    <!-- TODO: Design admin.html with sections for data management (seed/
    backup/restore), settings toggles, and user/system info (FR-39). -->
    ```

- [ ] **Backup Execution & Feedback:** Ensure when backup is triggered via UI, the user sees feedback. We might not want to block the server during zipping if data is large. If needed, we could run backup in a background thread and show a "processing..." message. But simpler: when "Backup" is clicked, start creating zip and once done, either auto-download or present a link "Download backup". If the process fails, catch exception and flash error. Write logs either way (e.g. logger.info "Backup completed: file path" or logger.error if failed) for audit (NFR-22 logs) [24] .

  ```
  # TODO: After backup creation, flash success message with file name or
  initiate download (improve UX per FR-39).
  ```

- [ ] **Restore Process:** When user uploads a backup and confirms restore, perform the steps to load data. Because this is sensitive, double-confirm ("Are you sure? This will overwrite your current data."). Ideally, backup should contain everything needed. We'll implement this carefully or mark as future if we can't complete now. If implemented, after restore, we will likely restart the app or at least re-query data for all modules (since DB changed significantly). Possibly easiest is to shut down and prompt user to restart application (or do `flask run` again) after restore. We can mention that in UI after successful restore ("Please restart PocketSage to finish restoring data."). Mark that as instruction.

- [ ] **User Management Hooks:** Add admin routes for user deletion or password reset, but since we have no UI login, this is mostly for future. We can stub: if in some debug mode, allow deletion of guest user (which essentially triggers `purge_guest_user()` from auth service [67] to wipe data). Mark TODO for actual multi-user support down the line.

  ```
  # TODO: Provide user management actions (password reset, delete user) - for
  future phases when multi-user is needed.
  ```

- [ ] **Configuration Enforcement:** On app startup, implement checks:

    - If `SECRET_KEY` is default, abort startup with an error (guide user to set env var) (NFR-2) [80] .
    - If `POCKETSAGE_ENCRYPT` is true, adjust DB connection to use SQLCipher. Possibly use `sqlcipher` URI or run `PRAGMA key='...'` after connecting. The exact method: if using

SQLModel with SQLite, one can attach a key by executing a raw SQL. We'll put that in the engine init if flag is set (FR-3) [7] . Mark a TODO to integrate SQLCipher library if available.
    - Ensure `.env` variables are loaded (FR-2) [71] , e.g. using python-dotenv or our config module. Already `config.py` likely parses environment (we saw POCKETSAGE_ prefix in SR-4 [94]).
    - Also ensure the instance folder and subdirs (exports, backups, logs) exist and have proper permissions (set to private/700 on Unix if possible, as in TODO they mentioned securing directories [63]).
    - Print out or log configuration info (except secrets) on start for transparency (like data directory path, encryption on/off).

    ```
    # TODO: Enforce secure config at startup: check SECRET_KEY, apply
    SQLCipher PRAGMA if enabled (FR-3, NFR-1, NFR-2).
    ```

- [ ] **Logging & Observability:** Use `logging_config.setup_logging()` (if exists) to set up a JSON logger with file rotation (FR-5) [72] . Confirm logs are written under instance/logs. Possibly expose last few log lines on Admin page (just for convenience). Not explicitly required, but helpful for debugging if something fails (could be a hidden debug toggle).

- [ ] **CLI Commands:** In `app.py` or a separate `commands.py`, use Flask CLI to add commands:

    - `flask seed-demo` – calls the seed function.
    - `flask export-data` – runs backup (maybe printing path).
    - `flask create-user` – if we want to allow creating a new user in headless mode. We add these to help testers/CI (FR-40) [59] . Mark in docs that these exist.

    ```
    # TODO: Add Flask CLI commands for key admin operations (seed,
    export, etc.) for operability (FR-40, NFR-24).
    ```

- [ ] **Testing Hooks:** Ensure that test environment can bypass certain features (like not running scheduler, or using an in-memory SQLite for speed). We could add a config flag for testing mode. Also prepare some unit tests for things like payoff calculations, streak calc, etc., as part of QA.

Finally, after implementing all modules above and the admin features, we will have a minimal yet fully functional offline finance tracker. The app will cover all **planned functionality** from the requirements (ledger with budgets and charts, habits with streaks, debts with payoff planning, optional portfolio, and robust admin tools), without including any unrequested features. Any features marked as future (like file watcher automation, detailed notifications, multi-user support, advanced analytics) are stubbed or documented with TODO comments for later development, in line with the specification that future enhancements will be handled later.

Below is a summary of the inline `# TODO` markers to be inserted into the codebase for guiding implementation:

File-wise TODO Insertion Summary:

- **src/pocketsage/models/transaction.py** (near model fields):

```
# TODO: Add 'updated_at' timestamp and 'version' field for optimistic
locking (FR-12, NFR-10).
```

- **src/pocketsage/routes/ledger.py** (define routes):

```
# TODO: Implement /ledger list, add, edit, delete routes with filtering &
pagination (FR-7, FR-8).
```

- **src/pocketsage/templates/ledger_list.html** (top of file):

```
<!-- TODO: Ledger table UI with filters, monthly summary, and spending
chart embed (UR-2, FR-10, FR-11). -->
```

- **src/pocketsage/templates/ledger_form.html** (top of file):

```
<!-- TODO: Transaction form fields and validation error display (FR-8,
NFR-17). -->
```

- **src/pocketsage/routes/ledger.py** (where category management would be handled):

```
# TODO: Add endpoints or modal for Category CRUD (FR-9) – create/edit/
delete categories.
```

- **src/pocketsage/services/budgeting.py** (if existing, or create it):

```
# TODO: Implement budget tracking logic and threshold alert stub (FR-13,
FR-52).
```

- **src/pocketsage/services/import_csv.py** (within transaction import logic):

```
# TODO: Ensure idempotent import by using external_id or hash to skip
duplicates (NFR-11).
```

- **src/pocketsage/services/reports.py** (or ledger_service.py, for charts):

```
# TODO: Generate spending breakdown chart for ledger (FR-11) using
Matplotlib.
```

- **src/pocketsage/routes/habits.py** (define routes):

```
# TODO: Implement /habits routes for list, add, toggle daily entry, archive
(FR-14, FR-18).
```

- **src/pocketsage/templates/habits_list.html** (top):

```
<!-- TODO: Habits list UI with name, streak, and 'done today' checkbox
(UR-11, FR-14). -->
```

- **src/pocketsage/services/habit_service.py** (new service for habits):

```
# TODO: Compute current and longest streak for a habit (FR-14).
# TODO: Schedule habit reminders stub (FR-17, future feature).
```

- **src/pocketsage/models/habit.py** (maybe near end or where appropriate):

```
# TODO: Possibly add longest_streak field or calculate on the fly; consider
timezones for entries (future).
```

- **src/pocketsage/routes/debts.py** (define routes):

```
# TODO: Implement /debts routes for list, add/edit liability, record
payment, and plan payoff (FR-19, FR-23).
```

- **src/pocketsage/templates/debts_list.html** (top):

```
<!-- TODO: Debts list UI showing each liability and payoff summary (UR-18,
FR-22). -->
```

- **src/pocketsage/services/debt_planner.py** (new module):

```
# TODO: Implement snowball and avalanche payoff calculations (FR-20,
FR-21).
# TODO: Guard against scenarios where debts can't be paid off with given
payments.
```

- **src/pocketsage/services/debt_planner.py** (for chart):

```
# TODO: Create payoff timeline chart (PNG) showing total debt vs time
(FR-22).
```

- **src/pocketsage/routes/debts.py** (in payment handling section):

```
# TODO: On payment record, update liability balance and create
corresponding Transaction entry (FR-23).
```

- **src/pocketsage/routes/debts.py** (after payoff calc):

```
# TODO: Display debt-free date and total interest saved in the response
(FR-24).
```

- **src/pocketsage/routes/portfolio.py** (define routes):

```
# TODO: Implement /portfolio routes for list holdings, add/edit, import,
export (FR-25, FR-29).
```

- **src/pocketsage/templates/portfolio.html** (top):

```
<!-- TODO: Portfolio holdings table and allocation chart display (FR-27,
FR-28). -->
```

- **src/pocketsage/services/portfolio_service.py** (or reuse import_csv for holdings):

```
# TODO: Parse imported CSV to upsert Holding entries (FR-25).
# TODO: Compute portfolio allocation percentages and gains (FR-27, FR-28).
```

- **src/pocketsage/services/reports.py** (for portfolio chart):

```
# TODO: Generate portfolio allocation pie chart (FR-27) for use in UI and
reports.
```

- **src/pocketsage/models/portfolio.py** (if adjustments needed):

```
# TODO: Confirm account linking; if multi-account needed in future,
implement account management.
```

- **src/pocketsage/routes/admin.py** (define routes):

```
# TODO: Implement /admin routes: seed_demo, backup, restore, toggle
settings (FR-37, FR-38, FR-39, FR-50).
```

- **src/pocketsage/templates/admin.html** (top):

```
<!-- TODO: Admin panel UI for seed data, backup, restore, settings toggles,
and user info (FR-39). -->
```

- **src/pocketsage/services/admin_tasks.py** (seed and backup functions):

```
# TODO: Implement seed_demo_data(): create demo records for all modules
(FR-37).
# TODO: Implement create_backup(): export all data to zip (FR-38, FR-50).
# TODO: Implement restore_backup(zip_path): import data from backup (FR-50)
with caution.
```

- **src/pocketsage/scheduler.py** (in _run_backup or init):

```
# TODO: After backup job, possibly notify user or update a status somewhere
(future improvement).
```

- **src/pocketsage/config.py** (or app startup code):

```
# TODO: Enforce SECRET_KEY is not default; enforce encryption PRAGMA if
enabled (NFR-2, FR-3).
```

```
# TODO: Load environment configuration for DB path, encryption flag, etc.
(FR-2, SR-4).
```

- **src/pocketsage/auth.py** (at top TODO exists):

```
# TODO: (Reintroduce proper multi-user auth later if needed; currently
using single local user).
```

- **pyproject.toml / pre-commit config** (if any):

```
# TODO: Ensure ruff/black formatting and tests run in CI (NFR-12, FR-46).
```

Each of these TODO comments is crafted to be **GitHub-compatible**, meaning they can be checked and tracked in the repository. By following this roadmap and addressing each TODO item, we will refactor PocketSage into a clean, modular, and extensible codebase that meets all the specified requirements. The end result will be a minimal viable offline finance tracker that is easy to navigate, with all core features (transactions, habits, debts, basic portfolio) implemented and placeholders for future enhancements. This overhaul emphasizes code readability, UI clarity, and robust data handling, positioning PocketSage for future growth and user adoption.

Sources:

1. PocketSage Requirements Specification [95] [42] [59] (functional requirements for ledger, habits, debts, portfolio, admin)
2. PocketSage Project Overview [2] [1] (architecture and module breakdown, offline-first approach)
3. HomeBank vs GnuCash UI Principles [3] (inspiration for simplicity and ease of use in ledger design)
4. Mint App Features [12] (filtering transactions by category/time for report visualization)
5. PocketSage Developer TODOs and Roadmap [96] [6] (current state of features and areas to address in refactor)

---

[1] [2] [4] [8] [26] [39] [82] PocketSage_Project_CEN4010.pdf
file://file_00000000e84471f59aabcaa2bdf4afbd

[3] Alternative for Quicken: HomeBank Finance/Accounting Software Tutorial – Living the Linux Lifestyle
https://livingthelinuxlifestyle.wordpress.com/2018/09/07/homebank-finance-accounting-software-tutorial/

[5] [7] [9] [10] [11] [14] [15] [16] [17] [18] [19] [22] [23] [24] [27] [28] [29] [30] [31] [33] [35] [36] [37] [41] [42] [44] [45] [50] [51] [53] [59] [60] [61] [62] [68] [71] [72] [73] [78] [79] [80] [81] [87] [91] [92] [93] [94] [95] PocketSage-Requirements-Doc.pdf
file://file_0000000086dc71f79062f1bc317417b1

[6] [40] [48] [56] [63] [74] TODO.md
https://github.com/Blood-Dawn/pocketsage/blob/d2b8c0c45edc3278e598b4926d66cf9d286f85fb/TODO.md

12 Features of the Mint Personal Finance App
https://www.thebalancemoney.com/mint-com-manages-accounts-budgets-and-more-online-1293882

13 category.py
https://github.com/Blood-Dawn/pocketsage/blob/66365c297c70c5c5efada0d7e5619b013f4a5cf3/src/pocketsage/models/category.py

20 47 transaction.py
https://github.com/Blood-Dawn/pocketsage/blob/66365c297c70c5c5efada0d7e5619b013f4a5cf3/src/pocketsage/models/transaction.py

21 watcher.py
https://github.com/Blood-Dawn/pocketsage/blob/66365c297c70c5c5efada0d7e5619b013f4a5cf3/src/pocketsage/services/watcher.py

25 Bugs : HomeBank
https://bugs.launchpad.net/homebank

32 34 38 habit.py
https://github.com/Blood-Dawn/pocketsage/blob/66365c297c70c5c5efada0d7e5619b013f4a5cf3/src/pocketsage/models/habit.py

43 46 49 liability.py
https://github.com/Blood-Dawn/pocketsage/blob/66365c297c70c5c5efada0d7e5619b013f4a5cf3/src/pocketsage/models/liability.py

52 55 portfolio.py
https://github.com/Blood-Dawn/pocketsage/blob/66365c297c70c5c5efada0d7e5619b013f4a5cf3/src/pocketsage/models/portfolio.py

54 66 96 ROADMAP.md
https://github.com/Blood-Dawn/pocketsage/blob/d2b8c0c45edc3278e598b4926d66cf9d286f85fb/ROADMAP.md

57 58 account.py
https://github.com/Blood-Dawn/pocketsage/blob/66365c297c70c5c5efada0d7e5619b013f4a5cf3/src/pocketsage/models/account.py

64 65 67 85 86 88 89 auth.py
https://github.com/Blood-Dawn/pocketsage/blob/d2b8c0c45edc3278e598b4926d66cf9d286f85fb/src/pocketsage/services/auth.py

69 70 75 76 77 83 90 scheduler.py
https://github.com/Blood-Dawn/pocketsage/blob/d2b8c0c45edc3278e598b4926d66cf9d286f85fb/src/pocketsage/scheduler.py

84 app.py
https://github.com/Blood-Dawn/pocketsage/blob/d2b8c0c45edc3278e598b4926d66cf9d286f85fb/src/pocketsage/desktop/app.py