TradeJounal Backlog

Numbered Agentic Product Backlog: TradeJournal MVP

Sprint 1: Project Foundation & Core Authentication 뻐 다

Epic: Project Foundation

- **T1: Task:** Initialize the project repository and file structure.

  Goal: Set up the complete project scaffolding.
  Agentic Prompt:
  1. Execute git init.
  2. Create a frontend directory and run npm create vite@latest . -- --template react-ts.
  3. Create a backend directory. Inside backend, create a Python virtual environment and install fastapi, uvicorn[standard], sqlalchemy, psycopg2-binary, python-jose[cryptography], and passlib[bcrypt].
  4. Inside backend, create the following empty directories: app/routers, app/models, app/schemas, app/crud, app/core, app/database.
  5. Inside frontend/src, create the following empty directories: pages, services, hooks, contexts, components.
- **T2: Task:** Create the local database configuration.

  Goal: Define and run the local PostgreSQL database using Docker.
  Agentic Prompt:
  1. Create a docker-compose.yml file in the project's root directory.
  2. **Content:** Configure a PostgreSQL 15 service. Set environment variables for the user, password, and database name. Map port 5432 to the host and configure a local volume for data persistence.
  3. Execute docker-compose up -d and verify the container is running.

Epic: User Authentication

- **US1: User Story:** "As a new user, I want to sign up for an account so that I can start logging my trades."

- **T3: Task:** Define the User database model and Pydantic schemas.

  Goal: Create the SQLAlchemy model and Pydantic schemas for the User.
  Agentic Prompt:
  1. Create backend/app/models/user.py. Inside, define a SQLAlchemy model class User with fields for id, email, password_hash, and created_at.
  2. Create backend/app/schemas/user.py. Inside, define Pydantic schemas: UserCreate (email, password), UserUpdate, and UserInDB (includes password_hash). Also create a User schema for API responses that excludes the password hash.
- **T4: Task:** Implement password hashing utilities.

  Goal: Create a security utility for password handling.
  Agentic Prompt:
  1. Create backend/app/core/security.py.
  2. **Content:** Import CryptContext from passlib.context. Create a pwd_context instance using "bcrypt". Define two functions: verify_password(plain_password, hashed_password) and get_password_hash(password).
- **T5: Task:** Implement the user registration endpoint.

  Goal: Create the API endpoint for user creation.
  Agentic Prompt:
  1. Create backend/app/routers/auth.py.
  2. **Action:** Implement a POST endpoint at /register that accepts a UserCreate schema.
  3. **Logic:** Check for email uniqueness. Hash the password using get_password_hash. Create and save the new User. Return the user data based on the response schema.
  4. Acceptance Criteria: Endpoint rejects passwords shorter than 8 characters.

- **US2: User Story:** "As an existing user, I want to log in to my account so that I can access my trade data."
  - **T6: Task:** Implement the user login endpoint.

    Goal: Create the API endpoint for user authentication and token generation.
    Agentic Prompt:
    1. In backend/app/routers/auth.py, add a new endpoint.
    2. **Action:** Implement a POST endpoint at /login/token that accepts an OAuth2 password request form.
    3. **Logic:** Authenticate the user by comparing the provided password with the stored hash using verify_password. If successful, create and return a JWT access token. If not, raise a 401 Unauthorized

exception.

- **T7: Task:** Create the frontend Sign-Up and Login pages and routing.

  Goal: Build the React components for user authentication.
  Agentic Prompt:
  1. Create frontend/src/pages/SignUpPage.tsx and frontend/src/pages/LoginPage.tsx. Build them as simple forms based on the UI/UX doc.
  2. In your main App.tsx, set up react-router-dom to render these components at /signup and /login routes.

- **US3: User Story:** "As a logged-in user, I want to log out so that I can securely end my session."
  - **T8: Task:** Implement frontend authentication state and protected routes.

    Goal: Manage user session on the frontend and protect dashboard routes.
    Agentic Prompt:
    1. Create frontend/src/contexts/AuthContext.tsx. Use React Context to provide the user's auth token throughout the app. Implement login and logout functions that manage the token in localStorage.
    2. Create a ProtectedRoute.tsx component that checks for the auth token. If it's missing, redirect to /login.
    3. Wrap the dashboard and trade log routes in App.tsx with this ProtectedRoute. Add a "Logout" button to your main navigation that calls the logout function from the context.

Sprint 2: Core Trade Management 📈

Epic: Trade Logging & Journaling

- **US4: User Story:** "As a logged-in user, I want to manually enter a completed trade..."
  - **T9: Task:** Define the Trade database model and schemas.

    Goal: Create the SQLAlchemy model and Pydantic schemas for Trades.
    Agentic Prompt:
    1. Create backend/app/models/trade.py. Define a Trade model with all required fields (ticker, prices, quantity, etc.) and a foreign key to the users.id.
    2. Create backend/app/schemas/trade.py. Define Pydantic schemas for TradeCreate, TradeUpdate, and Trade (for API responses).
  - **T10: Task:** Implement the trade creation endpoint.

    Goal: Create the API endpoint for logging a new trade.

Agentic Prompt:

1. Create backend/app/routers/trades.py.
2. **Action:** Implement a POST endpoint at /trades. This route must be protected.
3. **Logic:** Accept a TradeCreate schema. Calculate the P&L on the backend. Save the new Trade instance, linking it to the current user's ID from the JWT. Handle the incoming timezone data to convert and store entry_timestamp and exit_timestamp as UTC in the database.
4. **Acceptance Criteria:** Endpoint validates that entry_price, exit_price, and quantity are positive numerical values. Verify that user-provided text fields (e.g., journal_notes) are handled by the ORM to prevent SQL injection.

- **US5: User Story:** "As a user, I want to view a list of all my past trades..."
  - **T11: Task:** Implement the trade viewing endpoint.

    Goal: Create the API endpoint for fetching all of a user's trades.
    Agentic Prompt:

    1. In backend/app/routers/trades.py, add a new protected GET endpoint at /trades.
    2. **Logic:** Query the database for all trades where the user_id matches the authenticated user's ID. Return the list.
  - **T12: Task:** Create the frontend Trade Log component.

    Goal: Build the React page to display the list of trades.
    Agentic Prompt:

    1. Create frontend/src/pages/TradeLogPage.tsx.
    2. **UI:** Implement a data table with columns styled according to the UI/UX doc. Add a "View" button to each row. The table columns should be interactive, allowing the user to sort the entire list of trades. The default sort order should be by Exit Date in descending order (most recent first).
    3. **Logic:** On component mount, call the /api/trades endpoint and populate the table.
- **US6: User Story:** "As a user, I want to view the specific details of a single trade and add journal notes..."
  - **T13: Task:** Implement single trade view/update endpoints.

    Goal: Create API endpoints to get and update a single trade.
    Agentic Prompt:

    1. In backend/app/routers/trades.py, add two new endpoints.
    2. **Action 1:** A protected GET endpoint at /trades/{trade_id}.
    3. **Action 2:** A protected PUT endpoint at /trades/{trade_id} that accepts a TradeUpdate schema to update the journal_notes.

4. **Logic:** For both, ensure the requested trade belongs to the authenticated user before proceeding.

- **T14: Task:** Create the frontend Trade Detail/Edit modal.

  Goal: Build the modal for viewing and journaling a trade.
  Agentic Prompt:

  1. Create frontend/src/components/TradeDetailModal.tsx.
  2. **UI:** The modal should display all trade details in a read-only format, with a large text area for the journal_notes.
  3. **Logic:** When the "View" button in the TradeLogPage is clicked, this modal should open, fetch the specific trade's data from /api/trades/{trade_id}, and populate its fields. A "Save Notes" button should trigger the PUT request.

Sprint 3: Dashboard & Deployment 🚀

Epic: Performance Dashboard

- **US7: User Story:** "As a user, I want to see high-level performance metrics on my dashboard..."
  - **T15: Task:** Implement the dashboard statistics endpoint.

    Goal: Create the API endpoint for calculating and returning user performance stats.
    Agentic Prompt:

    1. Create backend/app/routers/dashboard.py.
    2. **Action:** Implement a protected GET endpoint at /dashboard/stats.
    3. **Logic:** Using SQLAlchemy, perform aggregate queries on the trades table for the authenticated user to calculate Total P&L, Win Rate, Average Win, and Average Loss. Return these in a JSON object.
  - **T16: Task:** Create the frontend Dashboard component.

    Goal: Build the main Dashboard UI.
    Agentic Prompt:

    1. Create frontend/src/pages/DashboardPage.tsx and a reusable frontend/src/components/StatCard.tsx.
    2. **UI:** In DashboardPage, create a grid layout.
    3. **Logic:** On component mount, fetch data from /api/dashboard/stats and pass the data to four instances of the StatCard component.
- **US8: User Story:** "As a new user with no trades, I want to be guided on what to do next..."
  - **T17: Task:** Implement the Dashboard's "empty state".

    Goal: Make the dashboard welcoming for new users.

Agentic Prompt:

1. Modify frontend/src/pages/DashboardPage.tsx.
2. **Logic:** Add conditional rendering. If the stats API returns that the total number of trades is zero, display a welcome message and a large "Log Your First Trade" button instead of the stat cards.

Epic: Production Deployment

- **T18: Task:** Configure hosting and environment variables.

  Goal: Prepare the application for deployment.
  Agentic Prompt:

  1. Create a .env.example file in the backend directory. Include placeholders for DATABASE_URL, JWT_SECRET_KEY, and ALGORITHM.
  2. Create a vercel.json file in the frontend directory to configure build settings and rewrites for API proxying.
  3. Create a Dockerfile in the backend directory to containerize the FastAPI application for deployment on services like Render.