ChatGPT

Tradersync-Style Trading Journal – Project Design Document

Overview

We aim to develop a **Tradersync-style trading journal** application using Python with a Windows GUI. The application will import trade records from TraderSync CSV exports, enrich those records with additional data from various sources, and present the consolidated information in a user-friendly table. The design prioritizes **speed, efficiency, and modularity**, ensuring the system can handle tens of thousands of records and be easily extended with new features in the future (e.g. strategy backtesting, chatbot integration, etc.). Data will be persisted in a local **SQLite** database for reliability and quick access, and the GUI will allow users to customize which columns are visible, as well as sort and filter the data interactively.

Key goals and features include:
- **CSV Import & Update:** Read trades from TraderSync-exported CSV files, adding new entries and updating existing ones (using trade date & symbol as identifiers) without duplication.
- **Data Backfill from APIs:** Enrich each trade with relevant market data (daily OHLCV, premarket/after-hours prices from Polygon.io) and fundamentals (share float from a provided float CSV, plus company info like country/industry from an external API) [1] .
- **Centralized Storage:** Use a SQLite database to store the combined dataset (trades + backfilled data), enabling persistence between sessions and efficient queries. SQLite can easily handle the scale (~40,000 records) of our journal with minimal overhead [2] .
- **Interactive GUI:** Provide a Windows desktop GUI (using a Python GUI framework like PyQt/PySide) to display the trading journal as a table. Users can choose which columns to show, sort the table by any column, and view less-critical fields in a separate metadata panel for the selected trade.
- **Modular Architecture:** Structure the application into independent modules – e.g. CSV ingestion, data fetching services, database layer, and UI – following best practices so components are easily maintainable or replaceable. This will facilitate future enhancements like advanced analytics, strategy tagging, backtesting, or AI assistant integration.

Requirements

Functional Requirements

1. **Import Trades from CSV:** The user can import one or multiple TraderSync CSV files containing trade logs. The system parses each file and adds trades to the journal. Duplicate trades (matching on Symbol and Date) should be recognized and not re-added; instead, their data can be updated if needed.
2. **Automated Data Backfill:** For each imported trade, if certain data fields are missing (e.g. daily prices, float, fundamentals), the system automatically retrieves them from the appropriate sources:
3. **Daily Prices:** Use Polygon.io API to get the stock's daily Open, High, Low, Close, Volume for the trade date, plus previous day's close and any pre-market or after-hours prices [1] [3] .

4. **Float (Shares Outstanding):** Look up the stock's float from the provided float CSV file (mapping ticker to float) [1].
5. **Fundamentals:** Optionally fetch static info like company **Country** and **Industry** (e.g. via an API such as FinancialModelingPrep or FinViz – whichever is easiest to integrate and has accessible data) [1]. These might be fetched on demand or in batch for new symbols.
6. **Other Flags:** (Future) Fields like recent or pending reverse splits, "Hidden Chinese" status, etc., could be determined via external services or manual input [4], but will be considered later.
7. **Database Storage:** All trade data and the backfilled fields are stored in a local SQLite database. The database schema will include tables for trades and possibly auxiliary tables for static data (like symbol info) or daily prices. Data should persist between sessions so the user doesn't need to re-import or re-fetch each time.
8. **GUI Table View:** The main interface presents the trades in a table (spreadsheet-like view) where each row is a trade and columns include both the original trade fields (date, symbol, side, setup, P/L, etc.) and the backfilled data (price changes, volume, float, etc.). The user can:
9. **Select Columns:** Choose which columns to display in the table vs. which to keep hidden (they would still be available in the metadata side panel). This helps focus on important fields while keeping the UI uncluttered.
10. **Sort/Filter:** Sort the table by any column (ascending/descending). For example, sort by date, by profit %, by float size, etc. (Each column will use the appropriate sort logic: numeric, date, or alphabetical). We may also allow filtering/search (e.g. by symbol or date range) to quickly find specific trades.
11. **View Metadata Panel:** When a trade row is selected, a side panel shows the less frequently used fields or extended info for that trade. For instance, fundamental data or notes could appear here rather than as columns in the main table unless the user chooses to display them.
12. **Performance:** The app should handle ~10,000 trades per profile (up to ~40,000 total across profiles) smoothly. Data retrieval and UI interactions (sorting, toggling columns) should be efficient. As an example, SQLite can easily handle this number of rows (practically it can handle billions) with proper indexing [2], and a well-designed UI model can display tens of thousands of rows without lag [5]. We will use strategies like bulk-fetching data for multiple trades at once and efficient GUI table models to ensure speed.
13. **Multiple Profiles:** Support multiple "profiles" or trading accounts, each with their own journal (approximately 3–4 profiles). This could be handled either by separate database files or by a profile identifier on each trade within one database. The user should be able to switch profiles to view their separate journals.
14. **Extensibility:** The system's architecture should accommodate future features, such as: strategy setup management (categorizing trades by setups and analyzing them), historical backtesting (using the data to simulate strategy performance on historical data), and even a chatbox powered by ChatGPT for querying or analyzing the journal data. These are outside the initial scope but influence the design (ensuring we can integrate additional data sources, analysis modules, or UI components later without a complete rewrite).

Non-Functional Requirements

- **Modularity & Maintainability:** The codebase will be organized into modules (or services/classes) with single responsibilities – e.g., a module for importing CSV, a module for fetching market data, a database access layer, and the UI layer. This makes it easier to update or replace components (for example, swapping Polygon for another data API) [6] [7]. We will follow SOLID principles where applicable and keep logic decoupled.

- **Usability:** The UI will be clean and intuitive, resembling familiar trading journal layouts. Important fields (Date, Symbol, Side, Setup, P/L, % change, etc.) will be readily visible, whereas less critical fields (detailed fundamentals, special flags) will be accessible but not always on display. The design will help users quickly scan and sort trades, similar to an Excel or Tradersync interface.
- **Performance:** Ensure that importing and data fetching are done in an efficient manner. For example, if thousands of trades are imported, we won't call the Polygon API for each trade sequentially in a naive loop (which would be slow and potentially hit rate limits). Instead, we will optimize by grouping requests (fetch data for a range of dates or multiple symbols at once if the API allows) and by caching results for repeated use. Also, the GUI will utilize Qt's Model/View architecture to handle large tables efficiently in memory [5], and potentially use background threads for long-running tasks (like API calls or heavy computations) to keep the UI responsive [8].
- **Data Integrity:** The import process will validate and parse data carefully (e.g. date formats, numeric fields) and handle errors (like malformed CSV or API unavailability) gracefully, informing the user as needed. We'll also ensure that backfilled data aligns with the correct trade (matching by symbol and date) and won't overwrite user-recorded values (e.g., we won't modify the trade's recorded entry/exit prices, we only add supplemental columns).
- **Persistence:** All data and user preferences (like which columns are visible) should be saved. SQLite will store the core data, and we can use a simple config file or an extra table for user settings (such as column visibility, last selected profile, etc.). This way, when the user closes and reopens the app, their data and settings persist.
- **Security:** API keys (for Polygon and any other services) will be stored securely (not hard-coded). We might use a config file or environment variable for the Polygon API key provided (e.g. `QjD_Isd8mrkdv85s30J0r7qeGcApznGf`). No sensitive personal data is involved aside from trades, but we still ensure the app doesn't expose the API key or corrupt the data.
- **Scalability:** Although the expected data volume is moderate, the design (especially the modular data fetching and database) will allow scaling up if needed – for example, handling more profiles or integrating more data points – without major refactoring.

Data Sources and Integration

A core aspect of this project is aggregating data from multiple sources into the journal. Below are the sources and how they integrate:

- **TraderSync CSV (Trade Data):** The starting point is the CSV export from TraderSync, containing the raw trade logs. Typical fields included are trade date, symbol, position size, entry/exit prices, P/L, trade side (long/short), trade setup name, etc. These fields are ingested as-is into the database (they form the base "Trades" table). The CSV provides essential trade info and some user annotations (like setup or notes). In our design, this data is considered the ground truth that the user inputs.

- **Polygon.io (Market Prices & Volume):** We use Polygon's REST API to fetch daily market data for each trade's ticker and date. Polygon's `Open-Close` endpoint can give us the stock's open, high, low, close, and volume for a given date, as well as pre-market and after-hours prices [3]. This covers fields like "% Change +" (percent change from previous close to today's high) and "% Change Hi to Lo" (intraday range percentage), which we can calculate from those prices. Polygon also offers a "previous day's close" endpoint [9] which gives the prior day OHLC – useful for gap calculations. We will likely use the daily aggregates endpoint to get a range of data in one call (for efficiency). For example, if we have trades for a symbol spanning many dates, we can fetch that symbol's daily data for all needed dates in one query [10] [11], then extract the specific days. The Polygon API integration will be via their REST endpoints using the provided API key. The design will abstract this into a **MarketDataService** class so that if needed, it could be swapped out (for example, using another data source or a local price database in the future).

- **Float Data (Shares Float):** We have an attached CSV (float file) that contains each stock's float (number of shares available for trading). This is essentially static fundamental data. We will import this file (or directly load it into a table) and use it to backfill the "Float" field in our trades. Since float is by symbol (not date-specific), we may create a **SymbolInfo** table in SQLite with one row per ticker, storing float and perhaps other static info (country, industry, etc.). On import or when a new symbol appears, we query this dataset to get the float and store it [1]. This way, float is filled in once and reused for all trades of that ticker. The float file will be read at startup or on import, and kept in memory (e.g., a dict) or in its own DB table for quick lookup.

- **Fundamentals (Country, Industry, etc.):** For additional company info like country of incorporation or sector/industry, we can use an external API. Options include:

- *FinancialModelingPrep (FMP)*: Provides a convenient API for company profile by symbol, returning fields like country, industry, etc. (Would require an API key, but it's free for moderate use).
- *FinViz*: The FinViz website lists country and industry for tickers; we could scrape the HTML page for the ticker (though this is a bit less robust).

- *DilutionTracker (DT)*: Possibly a source for flags like reverse splits or warrants (though may require subscription).
  Given the instruction to use "whichever has the most accessible information online" [1], we will likely choose FMP for a straightforward JSON API, or FinViz if we prefer not to rely on an API key for this. In the initial version, we might only fetch these fundamentals when a new symbol is encountered, and cache them in the **SymbolInfo** table. This data is not time-sensitive, so it can be fetched once. The design will include a **FundamentalDataService** that handles this, so it's abstracted away from the main logic.

- **Other Potential Data/Flags:** The provided field list mentions items like "Armistice", "Most recent RS <1y", "Pending RS", "Hidden Chinese", "Swan Danger", "News (filings)?" [12]. These are specialized fields likely derived from niche datasets or manual input. For example, "Most recent RS <1y" might check if the stock had a reverse split in the last year (could get from corporate actions or SEC filings), "Armistice" might indicate if the company has financing from Armistice Capital (could be manually tracked), "Hidden Chinese" might be a user-defined flag if the company is essentially Chinese (perhaps determined via the country info), etc. Our design will account for adding these fields later. We might have a configuration or a plugin-like system to add custom data enrichment steps. Initially, we will include placeholders or at least ensure the schema can be extended with additional columns, and possibly allow manual editing of such fields in the UI if needed.

Data Flow: The process of integrating these sources is illustrated in the flow chart below. New trades go through an import parser, get stored in the DB, then a backfill process populates missing fields by calling Polygon for prices, looking up float from the float file, and querying fundamentals APIs. Finally, the enriched data is saved and displayed in the GUI table.

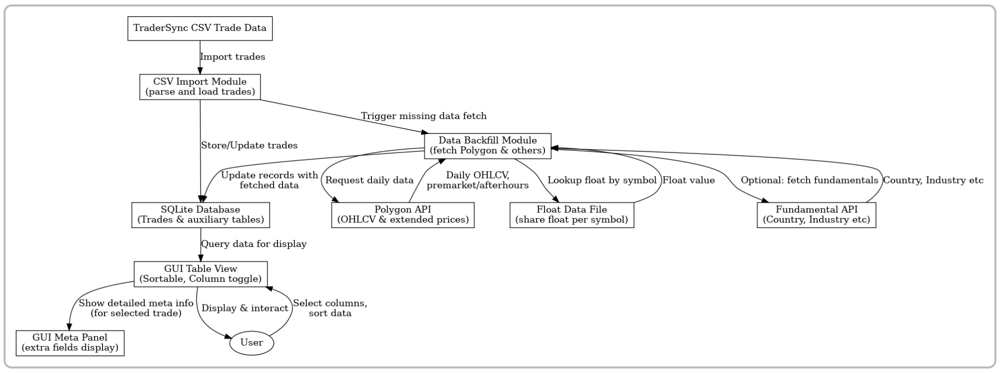

Figure: High-level data flow for the trading journal. Trades from TraderSync CSV are imported into the system. A backfill module retrieves market data from Polygon (daily OHLCV, pre-market/after-hours prices) and looks up static data like float and fundamentals from files/APIs. All data is stored in a SQLite database. The GUI then queries the database to display a consolidated table of trades. The user can interact with the GUI to sort data, toggle columns, and view additional metadata in a side panel.

System Architecture

To achieve modularity and clarity, we will structure the application into layers and components, each responsible for a specific part of the functionality. Below is an outline of the proposed architecture:

- **Importer Module:** This component handles reading the TraderSync CSV files. It will parse each row, possibly using a CSV library or pandas. It maps CSV columns to our internal data model fields (e.g., “Open Date” -> trade date, “Symbol” -> symbol, “Entry Price” etc.). The importer will then interact with the Database layer to insert or update records. It will use the trade’s unique key (likely a combination of Date, Symbol, and maybe an Entry Time or a trade ID if available) to detect existing entries. If an incoming trade matches one in the DB, we update any fields that might have changed (or simply skip if we assume no duplicates from the CSV). New trades get inserted. After importing, it triggers the **Backfill Module** for any trades that are new or that have missing data fields (for example, if the CSV had no info about daily high/low, which it wouldn’t).

- **Data Backfill Module:** This is a crucial part that enriches the trade entries with external data. It coordinates the calls to external services: it might take a list of trades (with their date & symbol) that need data and then:

- Determine which symbols and dates we need market data for (e.g., collect unique symbol/date pairs that are not yet in the DB’s price table).
- Query the Polygon API for each required symbol’s daily data. To be efficient, if multiple trades share the same symbol, it can fetch a range of dates in one call (Polygon’s aggregates API allows specifying a date range [10]). The module then picks out the specific date’s OHLCV from the results for each trade. We’ll obtain values like daily volume, open, high, low, close, and previous close (either from a separate call or by fetching one extra day in the range). Using these, the module can calculate the “% Change +” (which I interpret as the stock’s % gain from previous close to that day’s **high**) and “% Change Hi to Lo” (the % drop from the high of day to the low of day). These calculations will be done once data is retrieved.

- Look up each trade's symbol in the float data. We can load the float CSV into a dictionary `float_map[symbol] = float_value` when the app starts. The backfill simply does `float = float_map.get(symbol)`. This yields the float, which we then store in the trade's record or the symbol info table.

- If fundamentals (country, industry) are configured to fetch, check if we have that info for the trade's symbol in the **SymbolInfo** table. If not, call the Fundamentals API service to retrieve it, then store it. Mark that symbol as having fundamentals fetched to avoid repeat calls.
  The backfill module likely runs in a background thread or as a batch process after import, so the UI doesn't freeze during data fetch. It updates the database with the new information. By isolating all external data logic here, we make it easy to maintain and test (for example, we can mock Polygon API for testing this module).

- **Database Layer:** We will use SQLite (via Python's `sqlite3` or an ORM like SQLAlchemy for convenience). The database will have at least two main tables:

- `Trades` table: storing each trade record (with fields like date, symbol, side, setup, entry/exit, P/L, etc., plus columns for all the backfilled data such as daily open, high, low, close, volume, premarket high/low, afterhours high/low, float, country, industry, etc.). This could be a **wide table** with many columns, which is fine for up to 40k rows. Alternatively, we could normalize some data:
- `SymbolInfo` table: one row per symbol, containing static info (float, country, industry, perhaps last float update date if floats can change, etc.). This avoids repeating those fields in every trade row and makes updates easier (e.g., if we get a new float value for a symbol, update one place). The Trades table can reference this via a foreign key (symbol). However, for simplicity, and since float and country won't change often, we might also duplicate for quick filtering (denormalized). This is a design choice to weigh – performance vs. redundancy – but given 40k rows, duplication isn't a big issue.
- We might also have a `DailyPrice` table keyed by (date, symbol) for OHLCV data if we prefer not to store daily prices directly in Trades (especially if a symbol was traded multiple times on the same day, we wouldn't want to store the same OHLCV twice). But unless that scenario is common, keeping those fields in Trades is acceptable for simplicity.

- `Profiles` table: if using one database for multiple profiles, a Profiles table (and a profile_id in Trades) would segregate the data. Or we keep separate DB files per profile. The design will consider using a separate file per profile for simplicity (so the user can open a profile's journal file). SQLite's file-based databases make this easy.
  We will also implement a Database Access module or use an ORM so that other parts of the app do not execute raw SQL all over the place. This central layer can provide functions like `get_trades(profile_id, filters)`, `insert_trade(trade_obj)`, `update_trade_data(trade_id, data_dict)`, etc. This encapsulation means we could swap SQLite for another DB in future or add caching without altering the business logic.

- **GUI Layer:** For the user interface, we plan to use a modern Python GUI framework, likely **PyQt6/ PySide6** (Qt) or possibly a high-level toolkit like **PySimpleGUI** if speed of development was a priority. However, PyQt/PySide is preferred for performance and flexibility given the need for a complex table and interactive features. In Qt's model-view design, we will utilize a **QTableView** widget backed by a custom **QAbstractTableModel** or QStandardItemModel to display the trades [5]. The model will fetch data from the database (or we might load all trades into a pandas DataFrame or list of dicts in memory for the session for faster UI interaction – 40k records in memory is fine). We'll enable sorting by setting up a QSortFilterProxyModel, which allows the user to click column headers to sort rows. We will implement column hiding/showing by toggling the visibility or by regenerating the model based on user's selection of fields. Qt's flexibility allows dynamic columns. We might present a dialog or list of checkboxes for the user to select which columns to show. The **Meta Data Panel** could be implemented as a side panel (e.g., a QDockWidget or just a group of labels in a layout) that updates when a different row is selected. This panel would display the fields that are not currently in the table. For instance, if the table is showing only "Important Fields" (per the CSV's categorization) [13], the meta panel can show the "Fundamentals" fields. The GUI will also include menu options like "Import CSV" (to trigger the import workflow), "Refresh Data" (to manually trigger backfill again if needed), and perhaps profile switching and settings. Best practices in GUI design will be followed: the UI interactions (like import or refresh) will call controller functions that run heavy tasks in background threads and then update the model, to keep the app responsive [8].

- **Controller/Orchestration:** We may implement a controller layer (or just use the main application class) that ties everything together. For example, when the user clicks "Import CSV", the controller uses the Importer module to parse the file and store trades, then calls the Backfill module to update data, and finally refreshes the GUI view. If the user toggles columns or sorting, the controller updates the model or view state accordingly. This layer will contain the high-level logic of the application (but will remain thin, delegating actual work to the modules described above).

- **Logging & Error Handling:** The application will include logging (to a file or console) especially for the background data fetching. Any errors from the Polygon API or file parsing should be caught and logged, and the UI should notify the user in a friendly way (e.g., "Data for some trades could not be fetched due to network error" rather than crashing). This is part of making the app robust.

In summary, the architecture separates concerns clearly: *Importing*, *Data enrichment*, *Data storage*, and *Presentation/UI*. Each part can be developed and tested independently, and replaced or upgraded as needed. For instance, if we later decide to use a different data provider than Polygon, we would only modify the MarketDataService without changing how the rest of the app imports trades or displays the table.

Database Design

Using SQLite, the schema might look like the following (initially):

Table: Trades (if not fully normalized)

Field	Type	Source	Description
id (PK)	INTEGER	–	Unique identifier for each trade (autoincrement).
profile_id (FK)	INTEGER	–	References a profile if multiple profiles in one DB.
date	DATE	TraderSync CSV	Trade date (we'll likely use the **Close Date** or entry date from CSV) [14].
symbol	TEXT	TraderSync CSV	Ticker symbol of the stock [14].
side	TEXT	TraderSync CSV	"Long" or "Short" position [14].
setup	TEXT	TraderSync CSV	The trading setup or strategy tag (user-defined in TraderSync).
size	INTEGER	TraderSync CSV	Position size (number of shares or contracts).
entry_price	REAL	TraderSync CSV	Entry price of the trade.
exit_price	REAL	TraderSync CSV	Exit price of the trade.

profit_loss	REAL	TraderSync CSV	Profit or loss in dollars for the trade.
return_pct	REAL	TraderSync CSV	Return in % for the trade (if provided by TraderSync).
hold_time	TEXT	TraderSync CSV	Duration of trade (e.g. “02:18:44 hr” or “01:53 min”) – may store as text or convert to numeric (minutes).
notes	TEXT	TraderSync CSV	Any notes on the trade.
daily_open	REAL	Polygon API	The stock’s opening price on that date [15] .
daily_high	REAL	Polygon API	High price of the day [16] .
daily_low	REAL	Polygon API	Low price of the day [17] .
daily_close	REAL	Polygon API	Closing price of the day [18] .
daily_volume	INTEGER	Polygon API	Total volume for the day [19] .
prev_close	REAL	Polygon API	Prior day’s closing price (to compute gap).
premarket_high	REAL	Polygon API (extended)	Highest price in pre-market (if obtainable).
premarket_low	REAL	Polygon API (extended)	Lowest price in pre-market.
afterhours_high	REAL	Polygon API (extended)	Highest price after hours.
afterhours_low	REAL	Polygon API (extended)	Lowest price after hours.
percent_change_up	REAL	Computed	“% Change +”: e.g. (daily_high – prev_close) / prev_close * 100%. This can be computed on the fly or stored for convenience.
percent_hi_to_lo	REAL	Computed	“% Change Hi to Lo”: (daily_low – daily_high) / daily_high * 100% (a negative percentage).
volume_float_ratio	REAL	Computed	Perhaps we compute Volume as % of Float (to see relative volume). This would be (daily_volume / float) * 100%.
float_shares	INTEGER	Float file	Float (number of shares) for the stock [1] .
country	TEXT	Fundamentals API/Source	Country of the company (e.g. “USA”, “China”) [1] .
industry	TEXT	Fundamentals API/Source	Industry or sector of the company [1] .
armistice_flag	BOOLEAN	(user or DT API)	Example: flag if company is associated with Armistice (optional future field).
recent_rs_flag	BOOLEAN	(calc or DT API)	Flag if a reverse split occurred in last year (optional future).
pending_rs_flag	BOOLEAN	(DT API)	Flag if a reverse split is authorized/pending.
hidden_chinese_flag	BOOLEAN	(heuristic)	Flag if “Hidden Chinese” (possibly based on country or other info).
swan_flag	BOOLEAN	(user/fin data)	“Black swan danger” flag (criteria to be defined, optional).
news_exists	BOOLEAN	Manual or API	Whether significant news/filings were present (user can mark Yes/No).
news_link	TEXT	Manual or API	URL to news/filing if applicable.

(Fields in bold are those not originally in the TraderSync CSV but added via backfill. The design above is flexible – during implementation we might adjust which fields are stored vs. computed on the fly. For example, percent changes can be derived when needed rather than stored, but storing them can speed up sorting by that metric.)

If we opt for a `SymbolInfo` table instead of duplicating float, country, industry in each trade, that table might look like:

Table: SymbolInfo

symbol (PK)	float_shares	country	industry	last_updated_date	…

And Trades table would just have `symbol` as foreign key to SymbolInfo. When displaying or exporting, a join would retrieve those fields. The choice depends on convenience vs. normalization. For initial simplicity, we might include them directly in Trades to avoid complex joins when querying for the UI, given the relatively small dataset.

We will add necessary **indexes** to the database, for example: an index on `symbol` in SymbolInfo, and on `date` or `symbol,date` in Trades to speed up lookups (e.g., if we often query trades for a particular symbol or date range). SQLite can handle our data volume well, but proper indexing ensures that sorting or filtering by symbol/date in queries remains near-instant [20] .

User Interface Design

The UI is critical for a good user experience. We plan a **main window** that contains:

- **Menu Bar / Toolbar:** Options like *Import Trades* (to load CSV), *Refresh Data* (re-run backfill in case some data was missing or updated), *Profile* (to switch or manage profiles), and possibly *Settings* (to configure API keys or toggle features).
- **Trades Table:** The centerpiece is the table of trades. We will implement it using Qt's QTableView with a custom model. Each column corresponds to a field (date, symbol, side, P/L, etc.), and each row is one trade. The table will support:
- **Sorting:** The user can click on a column header to sort by that field. We'll utilize QTableView's sorting features (with QSortFilterProxyModel) to handle this efficiently, rather than manually sorting and resetting the table each time. Qt can handle large tables well if the model is implemented properly (only accessing data as needed for display) [5] . For example, we may subclass QAbstractTableModel's `data()` to retrieve values from an internal list or on-demand from the database.
- **Column Show/Hide:** There will be many columns (as seen in the schema). Not all will be shown by default. We might start by displaying the **Important Fields** as identified (Date, Symbol, Side, Setup, % Change, Volume, Float, etc.) [21] . Less important or advanced fields (like fundamentals or flags) would be hidden initially. The user can open a "Column Selector" dialog listing all fields with checkboxes to show/hide them. When the user applies changes, the table view updates to add or remove those columns. Qt supports dynamic column count, so our model can be built based on an active field list.
- **Row Selection & Meta Panel:** When a user selects a row (trade) in the table, the side **Meta Data Panel** shows additional details about that trade that might not be visible in the table. For instance, if "News link" or "Notes" are not in the table, the panel would display those. The panel could also show calculated metrics or even charts (e.g., perhaps a small sparkline of price or marking if the trade hit certain levels). Initially, it will likely be a simple form or text view listing the hidden fields' values and perhaps some summary statistics (like R-multiple, which was in CSV, etc.). This panel helps keep the main table uncluttered while still allowing access to all info.
- **Scrolling & Performance:** 40k rows is a lot to show at once, but Qt is optimized for virtual scrolling. We will ensure the table uses **lazy loading** (Qt does this by default with the model/view; it doesn't render all rows at once) and perhaps set uniform row heights to speed up rendering. We should avoid using something like QTableWidget for this volume, as it would require creating 40k QTableWidgetItem objects – instead, using QTableView with a model is the right approach [22] . In testing, we'll verify that the table can scroll smoothly. If performance issues arise, options include

pagination (show e.g. 1000 at a time) or further optimization of data access (like not calculating complex values on the fly during scrolling).

- **Editing/Manual Input:** It's not explicitly requested, but we might allow editing of some fields (like adding Notes, or toggling flags like "News?" yes/no). This would mean making some cells editable. We'll decide if that's in scope; if not initially, the design will at least not preclude it.

- **Status Bar:** We can use a status bar to show brief info, like "Imported 256 trades from CSV" or "Data fetch in progress…". This keeps the user informed about background actions.

Look & Feel: We will aim for a clean, spreadsheet-like look. Columns will be clearly labeled (possibly with short names, and tooltips for full description if needed). We can group columns logically or color-code them (e.g., fundamentals in a different shade) to distinguish categories of data. For example, perhaps use a light background for the fundamental columns to differentiate from the main trade metrics. We'll also ensure the app window can be resized and the table will adapt (maybe horizontal scroll for many columns). The user's column width adjustments should persist if possible (we can save the section sizes in settings).

Step-by-Step Development Plan

To implement this project, we will proceed in stages, verifying each component's functionality before integrating the next. Below is a step-by-step plan:

1. Project Setup and Skeleton:
- Initialize a Python project (we can use a virtual environment). Set up version control (e.g., Git) for the code.
- Create the basic Python package structure with modules: e.g. `importer.py`, `data_fetcher.py`, `database.py`, `ui.py`, etc., or organize by package (e.g. `services/`, `models/`, `ui/`).
- Install necessary libraries: for example, `pandas` (to help with CSV parsing if needed), `requests` (for API calls), `PyQt6` or `PySide6` (for GUI), and possibly `SQLAlchemy` for DB ORM (or just use built-in `sqlite3`). Also, if using FMP API, we might not need an extra lib (just use requests), but if using FinViz, we might use BeautifulSoup for HTML parsing.

2. CSV Importer Module:
- Implement the CSV reading functionality. We can start by reading the sample TraderSync CSV provided to ensure we correctly parse the format. The CSV has headers and data as shown in the example [23] [24]. We'll likely use Python's `csv` module or pandas to handle quoted fields and commas in numbers (notice "30,000" shares is quoted in CSV).
- Map the columns: determine which CSV columns correspond to which fields in our database. For instance, "Open Date"/"Close Date" might be the same date for day trades; we might take one of them as the trade date. Fields like "Return %" can map to our profit_loss or return_pct. We create a Trade data class or dictionary to hold parsed values.
- Connect to SQLite: Using `sqlite3` or ORM, set up the database file and create tables if not exist (on first run). We can write a small function to create tables per the schema.
- Write logic to upsert trades: for each parsed trade, either insert or update if it exists. Since initially there's no data, it will just insert. But on subsequent imports, we need to avoid duplicate insertion. We can define "exists" as same symbol and date (and perhaps same entry price to differentiate multiple trades on same stock in one day). This logic can be refined as needed (TraderSync might have a unique trade ID we can use if present).

- Test this module by calling it with the example CSV and then querying the DB to see that trades are stored correctly. At this stage, we'll only import the core fields from CSV (no backfilled data yet, so those columns might be null or default).

3. Data Fetcher (Backfill) Module:
- Using Polygon's API: Write a function to retrieve daily data for a given symbol & date (using the `/v1/open-close/{symbol}/{date}` endpoint, which returns JSON with open, high, low, close, volume, preMarket, afterHours, etc. [25] [3]). Use the API key for authentication (Polygon uses an `Authorization: Bearer <API_KEY>` header or a `apiKey` param). Start with single requests to ensure we can parse the response. Then extend to batch: for example, implement a function `get_daily_data(symbol, start_date, end_date)` using the aggregates API [10] to fetch a range of days in one go. This can return a dictionary of date -> prices. This is efficient if, say, the user imports a year's worth of trades for one symbol – one call gets all daily bars. We must heed Polygon's rate limits and plan accordingly (perhaps pause or spread calls if many symbols).
- Float lookup: Load the float CSV into memory (e.g., at app start or when first needed). Implement a simple function `get_float(symbol)` that returns the float from the dataset (or None if not found). The float file likely has a symbol column and float column; ensure to strip any whitespace and match case correctly (tickers are uppercase). All TraderSync symbols seem to be in uppercase in the sample.
- Fundamentals: Choose an API (say FMP). Implement `get_fundamentals(symbol)` that calls the API and parses the JSON to get country and industry. For example, FMP's profile endpoint might give a JSON array; we parse the first element's "country" and "industry". If no API is chosen initially, we can stub this out or use the float file as a placeholder for country/industry if it contained that (not likely). Alternatively, use FinViz by fetching `https://finviz.com/quote.ashx?t=SYMBOL` and parse out the relevant fields (country is often embedded in text). That might be more scraping work, so FMP is cleaner. Since the user indicated to use the easiest source, we assume an API like FMP.
- Once the sub-functions (prices, float, fundamentals) are ready, implement the main backfill logic: it will accept a list of trades (or better, query the DB for trades that need data – e.g., those with null daily_open or similar). It then:
a. Groups trades by symbol for efficiency. For each symbol, gather all dates needing data.
b. For each symbol, call the Polygon aggregate API once for the min to max date range needed. Parse results and update each trade's daily_open/high/low/close/vol. Also, call Polygon's previous day API for the day before each trade date (if not already included in aggregates) to get prev_close – or note that the open-close endpoint might directly provide previous close as part of afterHours (depending on how they define it). If not, we manually fetch or compute from consecutive data.
c. Compute percent_change_up and percent_hi_to_lo for each trade. This can be done now or later in UI, but doing it now and storing saves computation each time the table is sorted by that column.
d. Look up float for each trade's symbol (maybe do this once per symbol and apply to all trades of that symbol). If the float is missing for a symbol, we could mark it as "Unknown" or allow user to input it later. But with the provided file we likely have floats for relevant tickers (the example trades like NXTT, QMMM, etc., presumably are in the float file).
e. Fetch fundamentals for each new symbol encountered. This can happen here or lazily when needed in UI. It might be fine to do here so that country/industry are immediately available for filtering/sorting. But to avoid slowing down import too much, we could spawn separate threads or tasks for fundamentals if the list is long. Initially, do it straightforwardly for completeness.
f. Update the database: execute UPDATE statements to fill in the fetched fields for each trade (or if using an ORM, update objects and commit). Use transactions for batch update for performance. We should also insert into SymbolInfo table if using that.

- Test this module by manually calling it after an import, with a known symbol and date, and verify the DB gets updated correctly. For example, after importing the sample CSV which has trades on Sep 11, 2025 for multiple symbols, run backfill for those. (We may simulate API calls if offline or use sample data for testing.) Ensure that percent changes are computed correctly and floats match.

4. Develop the GUI (UI Module):
- Start with a basic PyQt application window. Verify that we can create a window on Windows with a table widget. We'll then implement the model for our trade data. We might use a simplified approach first: load all trades from DB into a pandas DataFrame or list of dicts, and then use Qt's model to display that. There are a couple of ways to do this:
* Use a QTableWidget (easier but not as scalable; we likely avoid this for final, but can prototype quickly to see the data).
* Implement a custom QAbstractTableModel. This is the preferred way [26] [27]. We define `rowCount`, `columnCount`, and `data(index, role)` to supply data. We also define `headerData` to name the columns. The model will hold a reference to the data (which could be the DataFrame or a list). Initially, we can make the model read-only (since editing is not a priority).
- Define the default set of columns to display. For example, a list like: `visible_columns = ["date", "symbol", "side", "setup", "profit_loss", "return_pct", "percent_change_up", "percent_hi_to_lo", "daily_volume", "float_shares"]` as these cover the important fields (date, trade info, outcomes, daily change, volume, float). The rest will be hidden or in meta. The model's `columnCount` and data methods will consider only `visible_columns`.
- Set up the QTableView: create it, assign the model, and enable sorting (`table.setSortingEnabled(True)`). We might also want to adjust presentation: e.g., format the date, format floats to 2 decimals, add a percentage sign for percent columns, etc. Qt's model can handle display role vs raw data (we can implement that in `data()` to format values). Alternatively, we keep raw in model and use a QStyledItemDelegate for formatting if needed. Simpler: format in the model for display role.
- Implement the Meta Data Panel: This could be a QWidget containing labels for each meta field. For flexibility, we might make it a scrollable text area or form since there can be many fields. When a table selection changes (connect to QTableView's `selectionModel().currentChanged` signal), we retrieve the selected trade's full data (either from the model or directly query DB) and populate the panel. For example, if the selected row's data is available in our internal list, we can easily get the dictionary and then set text for each label (like `notes_label.setText(trade['notes'])`, etc.). We will show all fields here or at least those not in visible_columns.
- Column selection UI: Add a button or menu item "Select Columns" which opens a dialog. The dialog can list all possible fields with checkboxes. We know all fields from the DB schema or we can maintain a master list. The user ticks which to show. On dialog confirm, we update `visible_columns` list and then either reset the model or tell the view to hide/show columns accordingly. If using QAbstractTableModel, the number of columns can change – we'd need to emit `beginResetModel()` / `endResetModel()` around changing the column list so the view updates. Or we could hide columns in QTableView by index without altering model – but since we want reordering as well, adjusting model might be cleaner. We'll ensure this operation is not too slow (resetting model for 40k rows is okay if done occasionally). We'll also store the user's selection (maybe in a config file or QSettings) so next run it remembers.
- Sorting: Verify that clicking headers sorts the data properly. If using QSortFilterProxyModel on top of our model, it will handle numeric vs text sorting if the data is stored as the correct type. We must ensure our model returns the correct type for comparisons (e.g., return float or Decimal for numeric columns, not formatted string). The proxy will sort based on data role. We might need to implement `lessThan` if custom sort needed (like for dates or when mixing text and numbers), but Python's built-in should handle

date objects and numbers fine. If any column doesn't sort correctly, adjust accordingly (for instance, dates might be stored as strings in DB, but we can convert to `QDate` or similar for sorting).
- Profile support: If separate DB files for profiles, implement a way in UI to open a profile (which would reinitialize the model with data from the selected DB). Possibly a dropdown or menu to switch profile, or open file dialog. This is straightforward: on profile change, close current DB, open new DB, query all trades, update model.
- Test the UI with sample data: After running import and backfill on our sample CSV, open the UI and see that trades appear with correct values, sorting works, and selecting a trade shows meta data. Fine-tune any layout or performance issues (e.g., if loading 40k rows is slow, consider only loading visible portion or adding a loading spinner while model is resetting).

5. Modularize and Clean Up:
- At this stage, we have all pieces working in a basic form. Now we refactor/improve: ensure that modules (Importer, Backfill, UI) are loosely coupled. For example, the UI shouldn't directly call SQL; it should go through the database layer. The Importer should perhaps use a Trade model object or namedtuple for clarity. We also add error handling: wrap API calls in try/except and handle failures (like if Polygon is down or returns 429 rate limit, we may retry after delay or skip and mark that trade for later).
- Implement caching where beneficial: e.g., if the user imports another CSV with trades in the same date range, we might have fetched those daily prices already – if we cached them or kept them in the DB (maybe our DailyPrice table), we can reuse rather than calling API again. The design might simply always check DB first for a given symbol/date's OHLCV before calling API. So incorporate that logic: Backfill checks if data is missing for a trade (either indicated by null fields or by absence in DailyPrice table) before fetching.
- Optimize database usage: If using sqlite3 directly, use executemany for batch inserts, and parameterized queries to avoid SQL injection issues (especially if we ever incorporate user input beyond controlled CSV). Possibly turn on WAL (Write-Ahead Logging) mode for better concurrent read/write if needed [28] (like UI reading while backfill writing, though likely we sequence these operations to avoid conflicts).
- Ensure the SQLite DB file is stored appropriately (e.g., in the user's Documents or application directory). If profiles = separate files, decide naming (maybe profile name as filename).

6. Advanced Features (Future Extensions Planning):
- Although not implementing now, we design with future in mind. For **setup management**, we might later allow users to define setups and tag trades or analyze performance by setup. Our data model already stores the setup name from TraderSync; we could later add a Setup table for more info. No change needed now, just keep note.
- For **historical backtesting**, if we wanted to test a strategy, we could use the price data we already gather. Perhaps we'd need minute-level data or more, which polygon can provide via different endpoints. We might structure the code to easily plug in a Backtester module. No immediate action, but our modular design (especially separating data retrieval and logic) will help when implementing such features.
- **ChatGPT integration:** If in the future we add a chatbot in the app for queries ("Which was my most profitable setup last month?" etc.), we can design an interface to feed relevant data to the OpenAI API. We might store an API key for OpenAI, and have a UI panel for chat. This will be a separate component that utilizes the data from DB. Designing now mainly means not making assumptions that block adding an interactive console or chat window. Our architecture has a main window – we could add a tab or an additional dialog for the chatbot. The important part is that our data access layer can easily retrieve answers to questions for the bot. We might even pre-compute some stats to help with quick responses.
- For now, we might just outline these possibilities in documentation and ensure the code structure (e.g., using dependency injection or config files) is flexible to accommodate them.

7. Testing and Validation:
- Prepare test cases: e.g., import a sample CSV and verify that the number of trades in DB matches, that the fields are correctly populated (especially edge cases like special characters in notes or different date formats).
- Test data fetching with a few known values – perhaps cross-check the Polygon results with another source or ensure they make sense (for example, if a trade on a known big mover, check the % change computed is correct).
- Test performance: create dummy 10k rows (maybe duplicate the sample trades many times or generate synthetic trades) and see how the app handles it. Measure import time, backfill time (mindful of API usage – perhaps simulate or use a smaller range for test), and UI loading/sorting speed. Optimize if any step is too slow (e.g., if backfilling 10k trades sequentially is slow, implement multithreading or asynchronous API calls in batches).
- User acceptance: if possible, get feedback on the UI layout – ensure it's intuitive. Perhaps adjust column naming (user-friendly names vs internal names). Provide ability to export the table or some summary if needed (Tradersync might have some reports; not in scope now, but keep data accessible for that).

8. Deployment Preparation:
- Once fully tested, we can package the application. Since it's a desktop app, we might use PyInstaller or a similar tool to create an executable for Windows. The SQLite DB and config files would reside alongside or in a known folder.
- Write documentation for the user (how to import, what each field means – the table provided by the user [29] is a good reference, and we can incorporate those descriptions into tooltips or a help section).

9. Future Iterations:
- Based on the modular design, future development can be iterative. For example, to add a "news scraper" feature that automatically fetches if there was a news on the trade date, we can create a NewsService module that uses an API or web scrape (like a feed or SEC filings) to flag trades with news. This module would plug into the backfill process or run as a separate action.
- If adding analytics (like performance charts, win rates, etc.), we'd either query the DB or use pandas on the dataset. This could be integrated into the UI as additional tabs or pop-ups (e.g., "Show Statistics" could display summary stats, maybe using matplotlib or a simple text report). The key is our data is centralized and in a structured form to enable this.

Throughout development, we will continuously refer to modern best practices and ensure clean separation of concerns. By following this plan, we will build a robust trading journal application that not only meets the current requirements but is also adaptable for many enhancements to come.

Conclusion

This design provides a comprehensive roadmap for creating the trading journal application. By combining **TraderSync data** with **market and fundamental data**, stored in a **reliable SQLite database**, and presented through a **dynamic GUI**, the tool will greatly assist in analyzing trades. The modular architecture (importer, backfill services, database, UI) ensures each part can be developed and tested in isolation and makes future modifications straightforward [30]. We also leveraged known frameworks and APIs to reduce complexity: for instance, using Polygon.io for one-stop access to daily prices and volumes [3], and Qt's model-view for efficient table handling [5]. The chosen technologies (Python, SQLite, PyQt, REST APIs) are well-documented and widely used, ensuring that we can find support and examples as needed during development. By following this design and step-by-step plan, we will create a fast, extensible, and user-friendly trading journal that can evolve with the user's needs.

Sources: The field mappings and data sources were confirmed by the provided specification [1], Polygon.io's API documentation for daily OHLCV data [3], and SQLite's noted capacity for large numbers of records [2], among others, to ensure feasibility and correctness of the approach. The use of Qt model/view architecture is guided by best practices for handling large tables in GUIs [5], which aligns with our performance goals.

---

[1] [4] [12] [13] [14] [21] [29] Trading Journal - Trades.csv
file://file-3zZSfbRRrXqrGbQXVHZedb

[2] sql - Can SQLite handle 90 million records? - Stack Overflow
https://stackoverflow.com/questions/3160987/can-sqlite-handle-90-million-records

[3] [15] [16] [17] [18] [19] [25] Daily Ticker Summary (OHLC) | Stocks REST API - Polygon
https://polygon.io/docs/rest/stocks/aggregates/daily-ticker-summary

[5] [8] [26] [27] Display tables in PyQt6, QTableView with conditional formatting, numpy and pandas
https://www.pythonguis.com/tutorials/pyqt6-qtableview-modelviews-numpy-pandas/

[6] Boost Product Development Efficiency with Modular Design Principles
https://moldstud.com/articles/p-increasing-efficiency-with-modular-design-in-product-development

[7] [30] Modular Design - FasterCapital
https://fastercapital.com/keyword/modular-design.html

[9] Previous Day Bar (OHLC) | Stocks REST API - Polygon
https://polygon.io/docs/rest/stocks/aggregates/previous-day-bar

[10] [11] Custom Bars (OHLC) | Stocks REST API - Polygon
https://polygon.io/docs/rest/stocks/aggregates/custom-bars

[20] Optimizing a large SQLite database for reading - Jacob Filipp
https://jacobfilipp.com/sqliteoptimize/

[22] Large number of rows for QTableView and QAbstractTableModel
https://www.qtcentre.org/threads/67415-Large-number-of-rows-for-QTableView-and-QAbstractTableModel

[23] [24] trade_data (2).csv
file://file-KwL6XN1QuyuDxfvihZ6i8J

[28] SQLite Optimizations for Ultra High-Performance - PowerSync
https://www.powersync.com/blog/sqlite-optimizations-for-ultra-high-performance