ChatGPT

Beginner-Friendly Trading Bot Project Structure (using Alpaca API)

Introduction

This project is a modular **Python trading bot** that runs locally, designed for a data analyst new to trading. It uses the **Alpaca API** for real-time market data and paper trading (simulated trades). The bot supports monitoring **5–20 stock tickers plus BTC/ETH**, and is organized for clarity and easy expansion. The initial focus is read-only (data collection and signal generation) with the ability to add trading execution logic shortly after. The design favors simplicity (no cloud deployment needed) while following sensible practices to allow growth.

Initial Setup and Prerequisites

Before coding, ensure the following setup is complete:

- **Alpaca Account & API Keys:** Sign up for a free Alpaca trading account and obtain your API Key ID and Secret Key (use a **paper trading** account for safety) [1] [2]. In Alpaca's dashboard, enable paper trading and copy your API credentials. Keep these keys private (do not commit them to code) [2].
- **Local Environment:** Install Python (3.8+ recommended) and create a virtual environment for the project. Activate the environment and install required libraries:
- `alpaca-trade-api` (official Alpaca Python SDK) [3]
- Any other needed packages (e.g., `pandas` for data handling, `schedule` for timing tasks, etc.). Save dependencies in a `requirements.txt` for easy installation.
- **API Key Configuration:** Configure your Alpaca API credentials for the bot. **Option 1:** Create a config file (as described below) to store the API Key, Secret, and API endpoints. **Option 2 (preferred for security):** Set environment variables for the keys and endpoints (e.g., `APCA_API_KEY_ID`, `APCA_API_SECRET_KEY`). The Alpaca SDK will read these if set [4]. For paper trading, ensure the API calls use the paper endpoint – either by setting `APCA_API_BASE_URL="https://paper-api.alpaca.markets"` as an environment variable or by specifying `base_url` in code [5]. This directs the SDK to the paper trading server so all orders are simulated, not live.

With the environment ready and API keys configured, you can proceed to structure the trading bot project.

Project Structure

The project is organized into a folder with clear modules, each handling a distinct aspect of the bot. This separation of concerns makes the bot easier to understand and extend [6]. Below is a possible folder and file layout:

```
trading_bot/
├── config/
│   └── config.py          # Configuration (API keys, settings, ticker list)
├── strategies/            # Strategy modules (trading logic)
│   └── example_strategy.py # Sample strategy (e.g., simple moving average
crossover)
├── data_fetcher.py        # Module for market data retrieval from Alpaca
├── trader.py              # Module for executing trades (paper trading logic)
├── main.py                # Main script to orchestrate the bot
├── requirements.txt       # Python dependencies (alpaca-trade-api, etc.)
└── README.md              # Documentation and usage instructions
```

Each component is described in detail below.

config.py (Configuration Module)

This script contains configuration constants and setup logic. It should store **API credentials** (API Key ID, Secret Key) and global settings such as the list of symbols to watch, mode flags, etc. For example, you can define a `API_KEY`, `API_SECRET`, and `BASE_URL` (set to `"https://paper-api.alpaca.markets"` for paper trading) [7]. It's wise to load these from environment variables for security [4] – for instance, using `os.getenv("APCA_API_KEY_ID")` – so that sensitive keys aren't hard-coded. This module can also include a **ticker list** (e.g., `WATCHLIST = ["AAPL", "MSFT", "ETH/USD", "BTC/USD"]`) which mixes stocks and crypto pairs. Alpaca supports crypto symbols with a slash (e.g. `"BTC/USD"` appears as the tradable symbol for Bitcoin [8]), so you can include BTC and ETH alongside stocks in the watchlist. Other settings like default timeframe for data (e.g., 1-minute bars) or trade parameters (like paper trading toggle, position size rules) can be defined here for easy tweaking. In summary, `config.py` centralizes all user-specific settings and secrets needed by the bot.

data_fetcher.py (Market Data Retrieval)

This module handles connections to Alpaca's **Market Data API** to fetch real-time price information for the assets in the watchlist. It utilizes the Alpaca Python SDK to request data. On initialization, it can create an Alpaca REST API client, for example:

```
import alpaca_trade_api as tradeapi
api = tradeapi.REST(API_KEY, API_SECRET, BASE_URL, api_version='v2')
```

using the credentials from `config.py`. The module provides functions to get the latest market data for multiple symbols. For instance, one function `get_latest_data(symbols)` could request recent price bars or quotes for a list of tickers. Alpaca's SDK allows querying multiple symbols at once – e.g. using `api.get_bars(symbols, timeframe='1Min', limit=1)` to get the latest 1-minute bar for each symbol [9]. This returns data (often as a Pandas DataFrame or dictionary) for each ticker. For example, you might call:

```
symbols = ["AAPL", "MSFT", "BTC/USD"]
bars = api.get_bars(symbols, timeframe='1Min', limit=1)
```

which would yield the most recent minute candle for Apple, Microsoft, and Bitcoin/USD [10]. The `data_fetcher` should parse these results into a convenient format (e.g., a Python dict or DataFrame mapping each symbol to its price, volume, etc.). If real-time streaming is required, this module could be extended to use Alpaca's WebSocket for live updates, but initially a simple polling approach (fetching every X seconds) is easier for a beginner. The goal of `data_fetcher.py` is to provide up-to-date market data on demand to other parts of the bot, abstracting away the API calls. Any errors (like rate limits or connection issues) should be caught and logged here, with maybe a retry mechanism since Alpaca has call limits (e.g., 200 requests/minute). This module remains **read-only** – it does not place any trades, only gathers data.

strategies/ (Strategy Modules)

This folder will contain one or more strategy scripts, each implementing a trading strategy or logic for generating buy/sell signals. For a beginner-friendly bot, you might start with a single file, e.g. `example_strategy.py`, which can serve as a template. The strategy module defines how the bot decides to trade based on market data. For instance, it might compute technical indicators or apply simple rules. A basic example strategy could be a **Moving Average Crossover**: calculate short-term and long-term moving averages and signal a buy when the shorter MA crosses above the longer MA (and signal a sell on the opposite). The strategy file could contain a function or class, e.g. `def generate_signals(data)` where `data` is the latest prices (perhaps a dict or DataFrame from `data_fetcher`). This function would analyze the data and return a **signal** for each asset – for example, a dictionary like `{"AAPL": "BUY", "MSFT": "SELL", "BTC/USD": "HOLD"}` or perhaps a list of trade instructions. Initially, the strategy might be very simple or even just a placeholder that always returns "HOLD" (no action) until real logic is added. The key is that this module is separate from data gathering and execution: it encapsulates decision-making. This makes it easy to add or swap strategies later. You can create additional strategy files (e.g., `rsi_strategy.py`, `mean_reversion.py`) and have the bot load the one you want to test. The modular design allows experimenting with multiple strategies by adding new modules without altering the core system. Each strategy module should ideally follow a common interface (e.g. each provides a `generate_signals` function or a Strategy class with a `.decide()` method) so that the main program can call them interchangeably. By isolating strategy logic, a user can focus on refining their algorithm in one place. (For now, keep strategies simple and well-documented, since the user is just starting out with trading logic.)

trader.py (Trade Execution Module)

The `trader.py` module is responsible for handling **paper trade orders** and simulating execution. In the initial read-only phase of the project, this module can operate in a dry-run mode: instead of placing real orders, it can log the intended trade actions or update a simulated portfolio state. This ensures no accidental real trades while testing logic. Once confident, the user can enable actual paper trading through this module by integrating Alpaca order APIs. The module will use the Alpaca REST client (same API object as in data_fetcher, or a new one) to submit orders to the paper trading endpoint. For example, to **place a buy order** for a stock, one would call Alpaca's `submit_order` function with appropriate parameters:

```
api.submit_order(symbol, qty, "buy", "market", "gtc")
```

This would execute a market buy for the given quantity (on the paper account) [11]. Similarly, a sell order can be placed by calling `api.submit_order(symbol, qty, "sell", "market", "gtc")` [12]. The `trader.py` module should include functions like `execute_signal(symbol, signal)` or `place_order(symbol, side, qty)` which take a trade signal (from the strategy) and carry out the appropriate action. For instance, if the strategy says "BUY" for AAPL, the trader module decides how many shares to buy (perhaps a fixed quantity or based on available cash and some position sizing strategy) and then submits the order via Alpaca's API [13]. Initially, a simple approach is to use a fixed quantity (e.g., buy 1 share or a small notional amount for crypto) for any buy signal and sell all holdings of a symbol on a sell signal. The module can check current positions using `api.get_position(symbol)` [14] if needed, to avoid duplicate orders or to know how much to sell. Since we are in paper mode, these API calls and orders will not risk real money. It's crucial to ensure the `BASE_URL` is set to the paper API so that orders go to Alpaca's **paper trading server** [5]. The trader module should handle API exceptions (e.g., if an order fails) and log the outcomes. In summary, `trader.py` converts strategy decisions into actual trade operations, either simulated or via Alpaca's paper trading API. By centralizing trade execution here, you can easily toggle between simulation and real paper trading by changing a flag or function implementation in this module (for example, a variable `DRY_RUN = True` could make `execute_signal` just print actions instead of calling `submit_order`). This layer could also enforce basic **risk management** – e.g., not allocating more than a certain percentage of capital per trade, or skipping trades if the account is low on buying power – which can be added as the user gains experience.

main.py (Orchestrator Script)

This is the **entry point** of the trading bot. Running `main.py` will launch the bot's routine. Its role is to tie together all the other modules in a coherent workflow. Typically, `main.py` will:

1. **Load Configuration:** Import `config.py` to retrieve API keys, list of tickers, and any settings. Possibly configure the environment (e.g., set up logging format or read command-line arguments if any).
2. **Initialize Components:** Create instances of the Alpaca API client (using keys and base URL from config) and pass them to helper modules if needed. For example, you might instantiate a data fetcher object, or simply ensure global API access in modules by importing the configured API object. If using object-oriented design, you could instantiate a `DataFetcher` class, a `Strategy` class, etc., but a simpler functional approach is fine: just import the functions from `data_fetcher.py` and `trader.py`.
3. **Retrieve Market Data:** Call the appropriate function in `data_fetcher` to get the latest prices for the watchlist (e.g., `market_data = get_latest_data(WATCHLIST)`). Ensure this returns the necessary data structure (prices or indicators needed by the strategy).
4. **Generate Trade Signals:** Invoke the strategy logic. For example, import `example_strategy.generate_signals` and do `signals = generate_signals(market_data)`. The returned `signals` might indicate for each symbol whether to buy, sell, or do nothing.
5. **Execute Trades (Paper):** Pass these signals to the trader module. For each symbol with a buy/sell signal, call the execute/order function. For instance:

```
for symbol, action in signals.items():
    if action == "BUY":
        trader.execute_buy(symbol)
    elif action == "SELL":
        trader.execute_sell(symbol)
```

Under the hood, `execute_buy`/`execute_sell` would place a paper trade via Alpaca or log the intent. If the strategy says "HOLD", do nothing for that symbol.

6. **Logging & Output:** Throughout the above steps, `main.py` should log key events and results. For example, log the fetched prices, the signals generated, and any trades executed. This can be as simple as printing to console, but using Python's `logging` module is recommended so that timestamps and levels (INFO, ERROR) are recorded and logs can be saved to a file (see Logging section below).
7. **Scheduling the Next Run:** Decide how the bot repeats. The main script could be designed to run **once** and exit (suitable if you use an external scheduler like cron to run it periodically), or it could loop internally. If using a loop, you might have something like:

```
while True:
    ... (perform steps 3-5) ...
    time.sleep(60)  # wait 1 minute before the next cycle
```

to continuously update every minute. Make sure to handle exceptions inside the loop so the bot doesn't crash silently. Alternatively, as discussed below, using an OS scheduler is more robust for long-running deployments [15].

The `main.py` script thus orchestrates one full cycle of data -> decision -> action. For a beginner phase, you might run it manually during market hours to observe behavior. As you add trading logic, you could then automate its execution on a schedule.

Logging and Scheduling (Optional Enhancements)

Logging: Implementing a logging system is highly recommended for a trading bot. Logging provides a record of what decisions were made and what actions were taken, which is invaluable for debugging and learning. You can configure the Python `logging` module at the start of `main.py` (or in a separate `utils.py`) to output to a file (e.g., `logs/bot.log`). Each cycle, log the timestamp, fetched prices, generated signals, and any orders submitted. For example, after executing trades, log lines like "`[2025-12-12 09:30:00] BUY 10 AAPL at $150 (paper)`". These logs let you verify the bot's behavior over time and identify any issues (e.g., if a strategy is generating too many signals or if orders are failing). Alpaca's API responses can also be logged. Make sure to also log exceptions or errors (network issues, etc.). A robust logging setup will help monitor the bot when it runs unattended [16].

Scheduling: There are two main ways to keep the bot running regularly: an **internal loop** or an **external scheduler**. An internal loop (using `while True` with delays) is straightforward, but there's a risk: if the script encounters an error and stops, the loop ends and the bot ceases running. A more fail-safe approach is to use the OS to schedule regular runs of the script [15]. For instance, using a cron job (on Linux/Mac) or

Task Scheduler (on Windows) to execute `python main.py` at desired intervals (e.g., every minute or every 5 minutes). With this approach, each run is independent – if one execution crashes, the next scheduled run will still start fresh. It also avoids memory buildup since the process restarts each time. For a local bot dealing with stocks, you might schedule it to run only during market hours (e.g., 9:30am to 4pm ET on weekdays) to avoid unnecessary activity when markets are closed. Alpaca's clock API (`api.get_clock()`) can be used in code to check if the market is open [17], but scheduling can enforce this externally too. For crypto (BTC/ETH), which trades 24/7, you might run the bot continuously or on a separate schedule for those assets. In summary, **for simplicity**, you can start with a basic loop in `main.py` that runs every minute. As you scale up, consider migrating to a cron-style schedule for reliability [15]. Remember to add some randomness or handling to avoid all actions happening exactly on the minute (which can align with many bots and cause API bursts). For example, sleeping 60 seconds plus a small random offset can distribute calls more evenly within rate limits.

How the Bot Works (Putting It All Together)

Once everything is set up, using the bot involves a few steps: 1. **Configure**: Edit `config.py` to add your API keys and the tickers you want to track (stocks like "AAPL" or "TSLA", and crypto like "BTC/USD" – ensure crypto is enabled on your Alpaca account if using those [18] [19]). Double-check that `BASE_URL` points to the paper trading API for safety. Adjust any other settings (e.g., strategy parameters) as needed.
2. **Install & Run**: Activate the Python virtual environment, install the requirements (`pip install -r requirements.txt`), then run `python main.py`. The bot will begin its cycle: fetching market data and printing or logging outputs. You should see log messages or console prints of prices and signals. On the first run, if using a dry-run mode, it will likely just report what it *would* trade. This is your chance to verify the strategy is behaving as expected.
3. **Paper Trading**: To enable actual paper trades, ensure that the Alpaca account has paper trading enabled and flip the bot from dry-run to active. This might mean changing a flag in `trader.py` (e.g., `DRY_RUN = False` or using a command-line argument to `main.py`). With that, when the strategy generates a "BUY" signal, the bot will call Alpaca's API to execute a paper trade. You can monitor your Alpaca paper dashboard to see orders and positions updating in real-time, confirming that the bot is trading virtually.
4. **Monitoring**: Let the bot run during the day (or continuously if looping). Keep an eye on the logs. If using scheduling (cron), make sure logs accumulate each run. Look for any errors or suspicious behavior (like too frequent trades or missed signals) and adjust the code as necessary. Because the structure is modular, you can refine one part (say, tweak the strategy or fix a data parsing bug) without overhauling the entire bot.
5. **Next Steps**: As you gain confidence, you can expand the bot. Add new strategy modules and perhaps a mechanism in `config.py` to choose which strategy to use. Implement more sophisticated risk controls (e.g., stop-loss in the strategy logic or bracket orders via Alpaca's API [13]). Include **error notifications** (for example, email or SMS if the bot crashes or a trade fails) for production readiness. You could also incorporate **backtesting** offline to test strategies on historical data before going live, though that can be separate from this real-time bot. The current architecture will support growth: you might, for instance, split `trader.py` into separate modules for order management and portfolio management as complexity grows, or add a database to record trade history. For now, the focus is on a clean, working baseline that is easy to understand.

Summary

This project layout emphasizes clarity: configuration, data fetching, strategy decision-making, and trade execution are each in their own module. This modular design is beginner-friendly and mirrors how larger trading systems separate concerns [6]. By running locally with Alpaca's paper trading API, you can safely learn how automated trading works. The **folder structure and script roles** described above act as a blueprint for building the trading bot. Each script's purpose is well-defined, making the code easier to navigate and expand. With this setup, a user can gradually increase the bot's sophistication – for example, adding more indicators to the strategy or handling more tickers – without needing to restructure the entire codebase. The Alpaca API provides the real-time data and trading functions needed, while the project's organization ensures the user can experiment and learn systematically. Start with this basic framework, and soon you'll be able to implement and test your own trading ideas in code, one module at a time, all on your local machine with full control. Happy coding and trading! [15] [20]

---

[1] [2] [3] [7] [9] [10] Python for Stock Market Analysis: Alpaca API - Quassarian Viper
https://q-viper.github.io/2022/05/01/python-for-stock-market-analysis-alpaca-api/

[4] [5] [13] [17] alpacahq.alpaca-trade-api-python.md
https://github.com/DLR-SC/repository-synergy/blob/115e48c37e659b144b2c3b89695483fd1d6dc788/data/readme_files/alpacahq.alpaca-trade-api-python.md

[6] [20] GitHub - steveman1123/stonkBot: The first go at a stock trading bot using Alpaca API
https://github.com/steveman1123/stonkBot

[8] [18] [19] Crypto Trading
https://docs.alpaca.markets/docs/crypto-trading-1

[11] [12] [14] [15] [16] Step-by-Step to Build a Stock Trading Bot
https://alpaca.markets/learn/stock-trading-bot-instruction