Stock Market PoC – Architecture & Implementation Notes

This document outlines the architectural decisions, performance optimizations, and testing strategy for the Stock Market PoC implementation.

**An assumption**: The documentation specifies a 15-minute time window for the Volume Weighted Stock Price (VWSP) calculation. However, no time window was defined for the GBCE All Share Index. For consistency, the same 15-minute time window has been applied to the VWSP values used in the GBCE calculation.

# Performance Optimizations

The core calculation functions, `volume\_weighted\_stock\_price` and `calculate\_gbce\_index`, are designed with real-world scalability in mind. In real stock market systems, millions of trades may be processed in real time. To optimize for this:

* Trades are grouped by stock symbol and stored in efficient `deque` structures, allowing append and fast time-window filtering. In our example 15 minutes.
* VWSP calculations iterate only over relevant trades in reverse order and stop early once outside the time window.
* GBCE Index calculation leverages the VWSP function and is ready for parallelization using a thread pool for multi-stock processing.

# Concurrency

All functions have async-await concept for application responsiveness. In real stock market systems, if an endpoint request thread takes time on calculation, it will still be available for other requests.

# Reusability and Flexibility

The GBCE Index function is implemented using the `volume\_weighted\_stock\_price` function to ensure consistency. For future flexibility, the VWSP calculation supports a configurable `minutes` parameter to define the time window dynamically.

# Clean and Scalable Layered Architecture

The code follows a clean architecture and microservice-aligned design with clear separation of concerns:

* - `models/` – DTOs for request/response validation (e.g., `Trade`)  
  - `services/` – Core business logic  
  - `api/routes/` – FastAPI route handlers  
  - `data/` – In-memory data store, designed to be swappable with a real database/repository layer

# Exception Handling

Raised clear, custom exceptions like StockNotFoundError and InvalidTradeTypeError. The microservice business logic will raise known exceptions and python-FastAPI will catch and return user-friendly HTTP responses. Since there are no 3rd party calls and optional logic TRY-Catch blocks haven’t implemented to be able to clean and exception-driven structure.

# Requirements

Before running the application requirements need to be installed

* pip install fastapi
* pip install uvicorn
* pip install pydantic
* pip install pytest
* pip install pytest-asyncio

# Unit Tests

Unit tests are implemented in `tests/test\_stock\_service.py`, covering business logic such as:

* - VWSP calculation (with/without trades)  
  - P/E ratio and dividend yield  
  - GBCE index calculations  
  - Trade recording and filtering

# Running the Application & Tests

To run unit tests:

pytest tests/

To launch the API application:

uvicorn app.main:app --reload

The application is implemented as a REST API. Once running, visit Swagger API documentation at:

<http://127.0.0.1:8000/docs>

A screenshot of a computer

AI-generated content may be incorrect.

# Documentation and Maintainability

* - Swagger UI is automatically generated from python-FastAPI and Pydantic models.  
  - Each service-layer function includes clear and consistent docstrings, explaining purpose, parameters, and return values.

# Future Improvements

* GBCE Index calculation can have **parallelization** using a thread pool for multi-fetch the VWSP function for every symbol for performance.
* Integration of TradeRepository into routes.py with **dependency injection** for more modularity and robustness
* **Caching** (Redis or in-memory) support for VWSP or GBCE for some required use-cases