# Backend Developer Take-Home Assignment

---

## Objective

Build a **backend service** that tracks and manages a **custom equal-weighted stock index** comprising the **top 100 US stocks** by daily market capitalization.
Each stock must maintain an **equal notional weight**, and the index should update daily to reflect market cap changes and rebalance composition.

Your backend must expose **RESTful APIs** to:

- Construct the index dynamically for any date or date range.
- Retrieve historical performance and compositions.
- Detect and show composition changes over time.
- Export all outputs into **well-formatted Excel files**.

This project assesses your ability to:

- Model real-world financial problems.
- Build modular services using **FastAPI**.
- Design efficient **SQL-based** storage with **Redis caching**.
- Apply backend best practices (APIs, persistence, caching, containerization).

---

## Assignment Requirements

### 1. Data Acquisition (Background Job)

- Fetch daily market capitalization and price data from a public API (e.g., Yahoo Finance, Alpha Vantage, IEX Cloud, Polygon).
- Store the data into **DuckDB** or **SQLite**.
- Evaluate at least **two data sources** and justify your choice.
- Ensure **at least 30 trading days** of history are ingested.
- Data acquisition must run as a **standalone job**, not via API.

### 2. Data Storage

- Use **DuckDB** or **SQLite**.
- Model tables for:
    - Stock metadata
    - Daily stock price and market cap
    - Index compositions and performance

- Query and transform data purely via **SQL**.

## 3. Index Construction API

- **POST /build-index**
  - Inputs: start_date, end_date (optional).
  - Construct the equal-weighted index dynamically for the given dates:
    - Select **top 100 stocks** daily by market cap.
    - Assign **equal weights**.
    - Persist compositions and performance.
    - **Cache results in Redis**.
- **Important**: Index building must happen at **API runtime**, not during ingestion. Ensure you understand the principles behind equal weighting, including how notional weights are assigned and rebalanced.

## 4. Index Retrieval APIs

| Endpoint | Function |
|---|---|
| **GET /index-performance?start_date=&end_date=** | Return daily returns and cumulative returns (cached). |
| **GET /index-composition?date=** | Return 100-stock composition for a given date (cached). |
| **GET /composition-changes?start_date=&end_date=** | List days when composition changed, with stocks entered/exited (cached). |

- Responses must be **JSON**.

## 5. Data Export API

- **POST /export-data**
  - Export:
    - Index performance
    - Daily compositions
    - Composition changes
  - Format: **Excel (.xlsx)** with clean headers and numeric formatting.

## 6. Containerization

- **Dockerize** the entire application.
- Provide a **docker-compose.yml** bringing up:
  - FastAPI service
  - Local database
  - **Redis instance**
- Ensure clean networking between services.

---

## Tech Stack

- **Python**
- **FastAPI**
- **Redis** or similar
- **DuckDB** or **SQLite**
- **Polars** and/or **Pandas**
- **Docker** (with docker-compose)

---

## Deliverables

- GitHub repository with:
    - Full source code.
    - **README.md** documenting:
        - Setup instructions (local + Docker).
        - Running data acquisition job.
        - API usage (sample curl/Postman).
        - Database schema overview.
- Working, clean REST APIs.
- Working Excel exports.
- Short write-up suggesting production/scaling improvements.

---

## Evaluation Criteria

| Category | Focus |
|---|---|
| Domain Understanding | Correct index logic, rebalancing, change tracking |
| Backend Engineering | API design, modular structure |
| Database Design | Schema quality, SQL proficiency |
| Caching Strategy | Correct and efficient use of Redis |
| Code Quality | Clarity, modularity, maintainability |
| Deployment | Working Docker setup |
| Documentation | Clear and complete |