Dynamic Portfolio Dashboard – Challenges & Solutions

Candidate: Vimal Anand

Tech Stack: Next.js (React), TypeScript, Node.js, Tailwind CSS

1. Project Overview

The goal of this assignment was to build a **dynamic portfolio dashboard** that displays real-time stock market data such as CMP, Gain/Loss, P/E Ratio, and Earnings. The application required integration with **Yahoo Finance** and **Google Finance**, both of which do not provide official public APIs.

To address this, I designed a **full-stack architecture** using Next.js, where all external data fetching and business logic are handled on the server side, and the frontend remains lightweight and focused only on rendering.

2. Key Challenges Faced & Solutions

Challenge 1: No Official APIs for Yahoo & Google Finance

Problem:
Yahoo Finance and Google Finance do not expose stable, official APIs. Direct client-side access is insecure and prone to rate limits or IP blocking.

Solution:

- Used **Node.js (Next.js API routes)** as a backend layer.
- Integrated:
  - `yahoo-finance2` (unofficial library) for CMP.
  - `axios + cheerio` scraping for Google Finance P/E ratio and earnings.
- Ensured **all third-party calls happen server-side only**.

Outcome:
Improved security, better control over failures, and compliance with assignment constraints.

Challenge 2: TypeScript Issues with Unofficial Libraries

Problem:
`yahoo-finance2` has incomplete or inconsistent TypeScript typings, causing errors like missing properties.

Solution:

- Avoid using `any`.

- Defined a **minimal domain-specific type** containing only required fields (e.g., `regularMarketPrice`).
- Explicitly cast the response to this safe type.

Outcome:
Maintained type safety while working with unreliable third-party typings.

Challenge 3: Rate Limiting & Performance Issues

Problem:
Frequent polling (every 15 seconds) could easily hit rate limits of unofficial data sources.

Solution:

- Implemented an **in-memory cache with TTL (15 seconds)** on the backend.
- Backend checks cache before calling external sources.
- Frontend polling + backend caching work together efficiently.

Outcome:
Reduced external API calls, improved response time, and avoided blocking risks.

Challenge 4: Partial Failures & Data Reliability

Problem:
If one stock or one external source fails, the entire portfolio should not break.

Solution:

- Implemented **per-stock error isolation** using safe wrappers.
- If Yahoo or Google fails for a single stock, fallback values are used.
- Added **last-known-good cache fallback** when the entire fetch fails.

Outcome:
The dashboard remains usable even during partial or complete data failures.

Challenge 5: Mixing Business Logic with UI Logic

Problem:
Initially, sector-wise aggregation and portfolio calculations were happening on the frontend, increasing complexity and re-renders.

Solution:

- Moved **all business logic** (portfolio calculations, sector aggregation) to the backend.
- Backend now returns:
    - Stock-level data
    - Pre-aggregated sector summaries
- Frontend components are now **purely presentational**.

Outcome:
Cleaner frontend, reusable API, and better separation of concerns.

Challenge 6: Real-Time Updates Without Over-Engineering

Problem:
Real-time updates were required, but WebSockets would add unnecessary complexity.

Solution:

- Implemented **client-side polling (15 seconds)** using `setInterval`.
- Combined with backend caching for efficiency.
- Proper cleanup using `clearInterval` to avoid memory leaks.

Outcome:
Achieved real-time behavior with a simple, production-friendly approach.

3. Architecture Summary

```
Frontend (Next.js / React)
        ↓ fetch("/api/portfolio")
Backend (Next.js API Routes)
        ↓
Caching Layer (TTL)
        ↓
Yahoo Finance / Google Finance
```

4. Final Outcome

- Fully functional **dynamic portfolio dashboard**
- Secure and scalable architecture
- Graceful error handling and fallbacks
- Clean separation of frontend and backend responsibilities
- Interview-ready explanations for all major design decisions

5. Conclusion

This assignment helped me apply real-world engineering practices such as handling unreliable external APIs, designing fault-tolerant systems, and balancing performance with simplicity. The final solution prioritizes **stability, maintainability, and user experience**, which are critical for financial applications.