Pre-Work Submission – Full Stack Developer

Project: Wells Fargo – Enterprise Data Processing Platform

Candidate: Mounika Yarramasu

SECTION 1: CONTEXT

Brief Description:

At Wells Fargo, I worked as a full-stack developer on an internal data-processing platform used by different business teams. The system handled large amounts of customer and financial data, and our goal was to make the application faster, more reliable, and easier for teams to use. I mainly worked on building React.js screens and integrating them with Spring Boot and Node.js microservices. I also helped improve backend APIs, database queries, and deployment pipelines. The project taught me how to build stable systems that support heavy workloads while still being easy to use.

Primary Technical Constraints:

- The app needed to work with **high-volume data** and respond quickly.
- All API calls had to follow **strict security and compliance rules**.
- The system had to run on **Docker + Kubernetes**, so each service needed to be lightweight.
- We had to support both **PostgreSQL and MongoDB,** depending on the feature.
- CI/CD pipelines had to be stable because deployments were frequent.

SECTION 2: TECHNICAL IMPLEMENTATION

Architecture Diagram (ASCII):

```
                      [ React.js UI ]
                            |
                            v
                    ---------------------
                    |   API Gateway |
                    ---------------------
                      /                 \
                     v                   v
          [Spring Boot Service]      [Node.js Service]
                     |                          |
                     v                          v
        [PostgreSQL Database]        [MongoDB Database]

                       |
                       v
                [Docker Containers]
                       |
                       v
               [Kubernetes Cluster]
```

Two-sentence explanation:

The frontend (React.js) communicates with our microservices through a secure API gateway. Each service runs in its own Docker container and is managed inside a Kubernetes cluster for scaling and reliability.

Code Walkthrough – One Critical Function

Function: User Login Validation (Spring Boot)

```
@PostMapping("/login")
public ResponseEntity<?> loginUser(@RequestBody LoginRequest request) {
   // 1. Check if user exists
   User user = userRepository.findByEmail(request.getEmail());
   if (user == null) {
      return ResponseEntity.status(404).body("User not found");
   }

   // 2. Match password
   boolean match = passwordEncoder.matches(request.getPassword(), user.getPassword());
   if (!match) {
      return ResponseEntity.status(401).body("Invalid password");
   }

   // 3. Generate token
   String token = jwtTokenUtil.generateToken(user.getEmail());

   return ResponseEntity.ok(new LoginResponse("Login successful", token));
}
```

Explanation:

- First, it checks if the user exists in the database.
- Then it verifies the password.
- If everything is correct, it creates a JWT token and sends it back to the frontend.
- The token is used for all future secure API calls.

Data Flow for Key Operation (Fetching Dashboard Data):

1. User enters email + password in React.js UI.
2. React sends the data to /login endpoint in Spring Boot.
3. Spring Boot checks the email in PostgreSQL.
4. If the user exists, the password is verified.
5. A JWT token is created.
6. Token is returned to the UI.
7. UI stores the token and uses it for further API calls.

SECTION 3: TECHNICAL DECISIONS

Decision 1: Using React.js + TypeScript for Frontend
Why we chose it:
React made UI development faster because we could reuse components. TypeScript helped reduce bugs by catching mistakes early.
Trade-offs:
- Good: Strong typing, reusable components, easy team collaboration.
- Bad: Slight learning curve for new developers.

Decision 2: Using Spring Boot Microservices Instead of a Monolithic App
Why:
We needed better performance and the ability to scale different services independently.
Trade-offs:
- Good: Faster deployments, isolated issues, easier scaling.
- Bad: Required more DevOps effort and more service communication.

Scaling Bottleneck + Mitigation
Bottleneck:
One service became slow when it handled large data sets, especially when multiple teams used it at the same time.
Solution:
We added caching using Redis and optimized SQL queries to reduce unnecessary joins.
After this change, the API became almost **40% faster** and reduced server load.

SECTION 4: LEARNING & ITERATION

Technical mistake & what I learned
At first, I wrote a few database queries that returned too much data, which increased load time.
I learned the importance of writing **lean queries** and using pagination to control data size.

One thing I would do differently today
I would introduce **Open API / Swagger documentation** from the start.
This would help frontend, backend, and QA teams understand each API without confusion.