**Case Study: Building and Deploying "Alstrom" with REST APIs, Docker, and Robust Testing**

**Problem Statement**

E-commerce platforms often struggle with maintaining consistent performance and scalability while managing complex deployments. There is a need for a robust architecture that supports seamless integration, scalability, and thorough testing to ensure a reliable user experience.

**Objectives**

1. **RESTful API Development:** Create RESTful APIs to handle user and product data.
2. **Containerization with Docker:** Use Docker to containerize the application for consistency across development, testing, and production environments.
3. **Comprehensive Testing Strategy:** Implement unit, integration, and end-to-end testing to ensure reliability and performance.
4. **CI/CD Integration:** Set up Continuous Integration and Continuous Deployment (CI/CD) pipelines for automated testing and deployment.

**Solution**

We developed " **Alstrom**," an e-commerce mobile application with a backend powered by RESTful APIs. Docker/Rancher was used for containerization, ensuring consistent environments across all stages of development. A thorough testing strategy was implemented to guarantee the app's reliability.

**Implementation**

**1. API Development:**

* **Tech Stack:**
  + **Backend:** SpringBoot for the RESTful API.
  + **Database:** Oracle DB for storing user and product information.
* **Endpoints:** Developed endpoints for user authentication, product listing, cart management, and order processing.

**2. Containerization with Docker:**

* **Dockerfiles:** Created Dockerfiles for both the backend and database services.
* **Docker Compose:** Used Docker Compose to define and run multi-container Docker applications, ensuring seamless integration between services.
* **Environment Configuration:** Managed different environment configurations (development, staging, production) using environment variables.

**3. Testing Strategy:**

* **Unit Testing:** Used Jest for unit testing individual components and functions.
* **Integration Testing:** Employed Supertest to test the integration of various API endpoints.
* **End-to-End Testing:** Utilized Cypress for end-to-end testing to simulate real user interactions.
* **Continuous Testing:** Integrated tests into the CI/CD pipeline using GitHub Actions for automated testing on every push.

});

**4. CI/CD Integration:**

* **Jenkins :**  Set up Jenkins for automated testing and deployment.
* **Pipeline Configuration:** Defined workflows for running tests, building Docker images.

**Results**

1. **Performance and Scalability:** Achieved consistent performance and scalability due to Docker containerization and Kubernetes orchestration.
2. **Reduced Bugs:** Comprehensive testing strategy significantly reduced bugs and improved the app's reliability.
3. **Faster Deployment:** CI/CD pipeline streamlined the deployment process, reducing deployment times by 50%.
4. **User Satisfaction:** Positive user feedback on the app's performance and reliability, contributing to a 20% increase in active users within the first three months.

**Lessons Learned**

1. **Docker Benefits:** Containerization with Docker ensured consistent environments, making development, testing, and deployment smoother.
2. **Importance of Testing:** A robust testing strategy is crucial for maintaining app reliability and performance.
3. **CI/CD Efficiency:** Integrating CI/CD pipelines accelerated the development cycle and improved deployment efficiency.
4. **Scalability Planning:** Planning for scalability from the outset helped in handling increased user load without performance degradation.

**Conclusion**

The "Alstrom" app successfully leveraged RESTful APIs, Docker, and a comprehensive testing strategy to provide a reliable and scalable e-commerce platform. This approach ensured consistency across development stages and contributed to a seamless user experience, demonstrating the importance of modern development and deployment practices in building high-quality applications.

This case study can be customized further based on specific requirements or additional details you'd like to include.

Problem Statement 1 :

Hands on :

1) Create a spring boot project named : messaging-api-app

2) Create a method named printGreetings to return Hi Oracle !! with the rest url :: http://localhost:9091/oracle

3) Create a test method to fail . Any method in src/test/ package

4) Create a git repo named messaging-api-app and upload the above project in this git repo

5) Create a jenkins job named message-job to build.

6) The above jenkins job should automatically helps automate build stage.

### 7) Find a way to build successfully even if tests are getting failed.

8) Run the build (jar file) and access http://localhost:9091/oracle

9) Verify the output !!