Practical Assignment: Microservice Architecture Development

Title: Design, Develop, and Deploy a Microservice-based Application

---

Objective

This assignment aims to give students practical experience with microservice architecture and expose them to the **core concepts** of microservices, including **service decoupling**, **API communication (REST/gRPC)**, **inter-service communication**, **containerization (Docker)**, **orchestration (Kubernetes)**, **CI/CD integration**, **fault tolerance**, and **monitoring**.

Students will gain hands-on experience by **designing, building, deploying, and testing** a cloud-based microservice application with a real-world use case.

Project Overview

You will build a **microservice-based e-commerce application** (or similar domain, such as booking or inventory management). This system will consist of multiple independent services communicating with each other, including:

1. **User Service**: Manages user data and authentication (e.g., sign-up/login).
2. **Product Service**: Manages product catalog and product-related operations.
3. **Order Service**: Handles order creation, updates, and tracking.
4. **Payment Service**: Simulates payment processing (including error simulation for failed transactions).
5. **Notification Service**: Sends email or SMS notifications when orders are placed/processed.

Each service should be self-contained, maintain its own **database**, and communicate via **REST APIs** or **gRPC**.

Tasks and Deliverables

Part 1: Project Setup and Service Design (10%)

- **Architectural Design**: Design the architecture diagram for the microservice system.
- **Service Definitions**: Identify and define the **responsibilities** and **interfaces** (API endpoints) of each microservice.
- **Database Design**: Choose appropriate databases for each service (SQL/NoSQL).
    - Example: Use **PostgreSQL** for User Service and **MongoDB** for Product Service.
- **Tech Stack Selection**: Choose appropriate technologies (e.g., Django/Flask for API services, Node.js, etc.).
- **API Contracts**: Provide **OpenAPI/Swagger documentation** for each service API.

Deliverable: Submit the **architecture diagram, API contracts**, and database schema design.

Part 2: Service Implementation (40%)

Implement each of the following services:

1. **User Service**:
    - Register/login users using **JWT authentication**.
    - Provide a user profile management feature.
    - Store user data securely.
2. **Product Service**:
    - Manage CRUD operations for products (create, read, update, delete).
    - Provide product search functionality.
3. **Order Service**:
    - Manage order creation and updates.
    - Integrate with the **Payment Service** to simulate payments.
4. **Payment Service**:
    - Process payments and return a success/failure response.
    - Simulate payment errors for testing.
5. **Notification Service**:
    - Send notifications (using a **message queue** such as **RabbitMQ/Kafka**).

    - Notify users via email/SMS when an order is successfully placed.

Deliverable:

- Source code for each microservice.
- API documentation using Swagger/OpenAPI.

Part 3: Communication and Fault Tolerance (15%)

- Implement **synchronous communication** using REST/gRPC between services.
- Integrate a **message queue** (e.g., Kafka, RabbitMQ) for **asynchronous communication** between Order and Notification services.
- Implement **retry mechanisms** and **circuit breakers** to handle service failures using tools like **Resilience4j** or **Istio.**

Deliverable:

- Code demonstrating fault-tolerant communication.
- Brief report explaining inter-service communication strategies used.

Part 4: Containerization and Deployment (20%)

- **Containerize** all services using **Docker**.
- Create a **docker-compose** file to run the services locally.
- Deploy the services to a **Kubernetes cluster** (e.g., Minikube or cloud provider like GKE, AKS, or EKS).
- Expose APIs through a **Kubernetes ingress controller** or **API gateway** (like **Kong** or **Istio**).

Deliverable:

- Dockerfiles for each service.
- Kubernetes deployment configurations (YAML files).
- A brief demo video showing the services running on Kubernetes.

Part 5: CI/CD and Monitoring (10%)

- Set up a **CI/CD pipeline** using **GitHub Actions** or **Jenkins** to automate building, testing, and deploying your services.
- Integrate **monitoring tools** such as **Prometheus** and **Grafana** for service health monitoring.
- Configure **alerts** for service failures or high resource usage.

Deliverable:

- CI/CD pipeline configurations.
- A dashboard screenshot from Prometheus/Grafana monitoring your microservices.

Part 6: Testing and Reporting (5%)

- Write **unit tests** and **integration tests** for critical services (e.g., Payment Service).
- Perform **load testing** on the system using tools like **Apache JMeter** or **k6**.
- Submit a **final report** summarizing:
  - Challenges faced.
  - Design and implementation decisions.
  - Future improvements or enhancements.

Deliverable:

- Test scripts and results.
- Final report (max 2000 words).

Evaluation Criteria

Category	***Weight***	***Criteria***
Architecture and Design	10%	Clear architecture diagram and API contracts
Service Implementation	40%	Functional services with correct logic
Communication and Fault Tolerance	15%	Proper use of REST/gRPC, message queues, and fault-tolerant mechanisms

Containerization and Deployment	20%	Correct Docker and Kubernetes deployment
CI/CD and Monitoring	10%	Working CI/CD pipelines and functional monitoring
Testing and Reporting	5%	Comprehensive testing and well-structured final report

Submission Guidelines

1. Submit a **GitHub repository** with all code, configurations, and documentation.
2. Include instructions on how to run the services locally using **Docker Compose**.
3. Upload your **final report** and API documentation as PDFs.
4. Optional: Provide a link to your deployed application (if deployed on cloud).

Additional Instructions

- Work **individually** or in **teams of 2** students.
- Use **Git version control** throughout the project. Each commit must reflect meaningful progress.
- Document all services and configurations to ensure **reproducibility**.
- Follow best practices for **security and scalability**.

Academic Integrity

Adhere to the university's policies on **plagiarism** and **academic integrity**. Collaboration within teams is encouraged, but any external sources must be properly cited.

Conclusion

This assignment offers a practical introduction to **microservices** and simulates a real-world software development project. It requires careful planning, effective collaboration, and technical proficiency across multiple domains, including API development, containerization, deployment, and testing. This hands-on approach ensures students are well-prepared for future roles in software engineering and cloud computing environments.