LAB ASSIGNMENT 2

SUBJECT

Software Design and Architecture

TEACHER

Sir Mukhtair Zamin

SUBMITTED BY

NIMRA JADOON (FA22-BSE-011)

ZOYA KAYANI (FA22-BSE-042)

SAUD UR REHMAN (FA22-BSE-048)

TALHA REHMAN (FA22-BSE-159)

DEADLINE

01. 04. 2025

Department of Software Engineering

COMSATS University Islamabad

Abbottabad campus

Part 1: Five Major Architectural Problems and Their Solutions

---

1. Monolithic Architecture Limitations

- **Problem**:
  - A single, tightly coupled system becomes difficult to scale and maintain.
  - Adding a new feature requires modifying and redeploying the entire system.
  - Scaling the entire system is resource-intensive.
- **Solution**:
  - Transition to **Microservices Architecture**, where:
    - Components are independent and communicate via APIs.
    - Individual services can be developed, deployed, and scaled independently.
  - **Example**: Netflix successfully transitioned from a monolithic architecture to microservices.

---

2. Database Bottleneck

- **Problem**:
  - Centralized databases create performance bottlenecks in high-traffic applications.
  - Latency increases, and downtime becomes more likely as the load grows.
- **Solution**:
  - Implement a **Distributed Database System** or **Database Sharding** to spread the load across multiple nodes.
  - **Example**: Amazon moved to DynamoDB for a scalable and distributed database solution.

---

3. Single Point of Failure

- **Problem**:
  - Dependency on a single server or component can lead to system-wide outages.
  - Example: Early Twitter's "Fail Whale" incidents were caused by server overloads.

- **Solution**:
    - Introduce **Redundancy** and **Load Balancing** to distribute traffic across multiple servers.
    - **Example**: AWS Elastic Load Balancer ensures high availability by distributing workloads.

---

4. Legacy Code and Incompatibility

- **Problem**:
    - Legacy systems are difficult to integrate with modern software.
    - Incompatibility leads to delays, errors, and high maintenance costs.
- **Solution**:
    - Use **APIs** and **Middleware** to facilitate gradual migration.
    - Adopt **Service-Oriented Architecture (SOA)** for better modularity and integration.

---

5. Performance Issues in Real-Time Systems

- **Problem**:
    - Latency in real-time data processing leads to slow responses in IoT systems or high-speed applications.
- **Solution**:
    - Use **Event-Driven Architecture** and tools like **Apache Kafka** for real-time data streaming and processing.

Part 2: Replicating and Solving a Problem

Problem: Monolithic to Microservices Transition

Scenario:

- A monolithic e-commerce system has a tightly coupled "Order Management" module.
- Placing an order slows down unrelated features like browsing and searching.
- This creates scalability and performance issues.

Step 1: Initial Monolithic Architecture

- **Description**:
    - All features (order placement, browsing, searching) are handled in a single class.
    - Adding new features or scaling specific parts is difficult.

```
// Monolithic Architecture Example
import java.util.HashMap;
import java.util.Map;

public class EcommerceSystem {
    private Map<String, Integer> inventory = new HashMap<>();
    private Map<String, Integer> orders = new HashMap<>();

    public EcommerceSystem() {
        inventory.put("item1", 10);
        inventory.put("item2", 5);
    }

    public String placeOrder(String item, int quantity) {
        if (inventory.containsKey(item) && inventory.get(item) >= quantity) {
            inventory.put(item, inventory.get(item) - quantity);
            orders.put(item, orders.getOrDefault(item, 0) + quantity);
            return "Order placed successfully";
        }
        return "Order failed";
    }

    public Map<String, Integer> browseItems() {
        return inventory;
```

```
  }

  public String searchItem(String item) {
    return inventory.containsKey(item) ?
        "Available: " + inventory.get(item) : "Item not found";
  }

  public static void main(String[] args) {
    EcommerceSystem ecommerce = new EcommerceSystem();
    System.out.println(ecommerce.placeOrder("item1", 2));
    System.out.println(ecommerce.searchItem("item1"));
    System.out.println(ecommerce.browseItems());
  }
}
```

Step 2: Transition to Microservices

Goal:

- Split the system into three services:
  - **OrderService**: Handles orders.
  - **InventoryService**: Manages inventory.
  - **SearchService**: Handles product search.

Order Service

```
public class OrderService {
  private Map<String, Integer> orders = new HashMap<>();

  public String placeOrder(InventoryService inventoryService, String item, int quantity) {
    if (inventoryService.reduceStock(item, quantity)) {
            orders.put(item, orders.getOrDefault(item, 0) + quantity);
            return "Order placed successfully";
        }
        return "Order failed";
    }
}
```

Inventory Service

```
import java.util.HashMap;
import java.util.Map;

public class InventoryService {
    private Map<String, Integer> inventory = new HashMap<>();

    public InventoryService() {
        inventory.put("item1", 10);
        inventory.put("item2", 5);
    }

    public boolean reduceStock(String item, int quantity) {
        if (inventory.containsKey(item) && inventory.get(item) >= quantity) {
            inventory.put(item, inventory.get(item) - quantity);
            return true;
        }
        return false;
    }

    public Map<String, Integer> getInventory() {
        return inventory;
    }
}
```

Search Service

```
public class SearchService {
    private InventoryService inventoryService;

    public SearchService(InventoryService inventoryService) {
        this.inventoryService = inventoryService;
    }

    public String searchItem(String item) {
        return inventoryService.getInventory().containsKey(item) ?
              "Available: " + inventoryService.getInventory().get(item) : "Item not found";
    }
}
```

Step 3: Orchestrating the Microservices

```
public class MicroservicesExample {
    public static void main(String[] args) {
        // Create services
        InventoryService inventoryService = new InventoryService();
        OrderService orderService = new OrderService();
        SearchService searchService = new SearchService(inventoryService);

        // Place an order
        System.out.println(orderService.placeOrder(inventoryService, "item1", 2));

        // Search for an item
        System.out.println(searchService.searchItem("item1"));

        // View inventory
        System.out.println("Inventory: " + inventoryService.getInventory());
    }
}
```

Benefits of Microservices Transition

1. **Scalability**:
   - Scale each service independently based on demand.
2. **Maintainability**:
   - Update or debug individual services without affecting others.
3. **Fault Isolation**:
   - A failure in one service (e.g., inventory) does not crash the others.

This Java example demonstrates the transition from a monolithic architecture to microservices by splitting responsibilities into independent classes and coordinating them effectively.