**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**

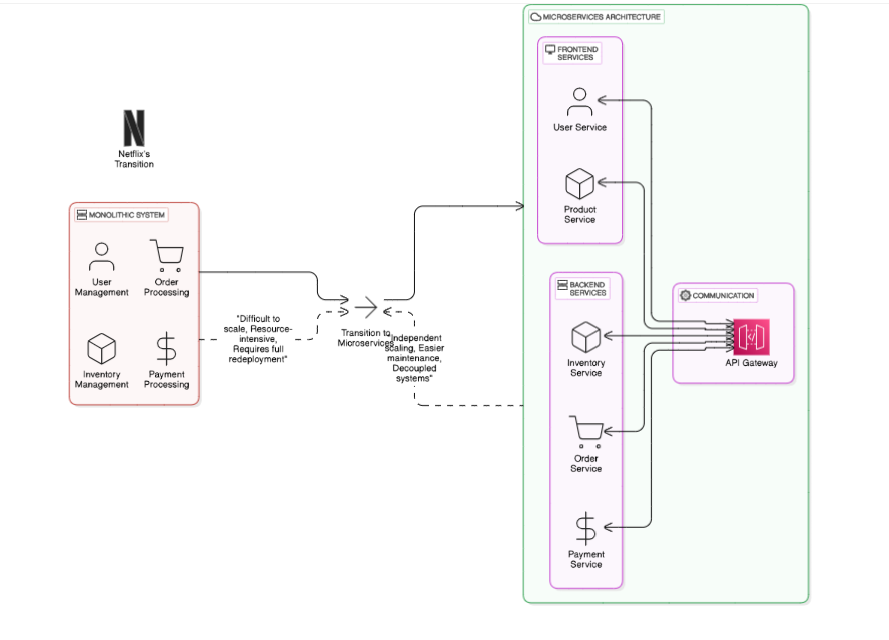
**Abbottabad campus**

**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.

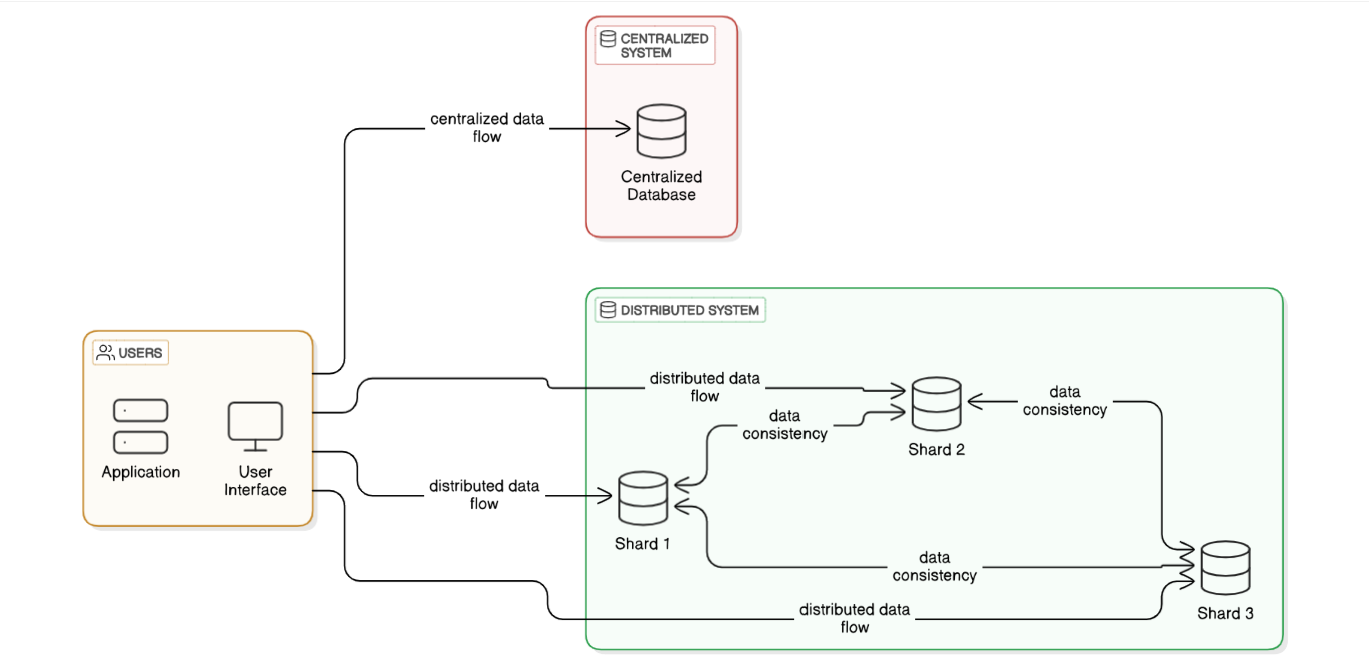
**Diagram**:



**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.

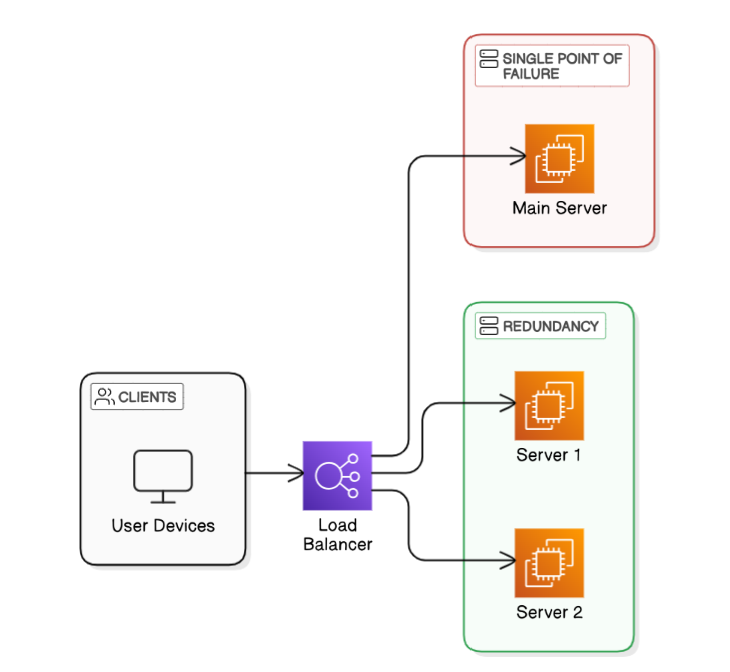
**Diagram**:



**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.

**Diagram**:



**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.

**Diagram:**

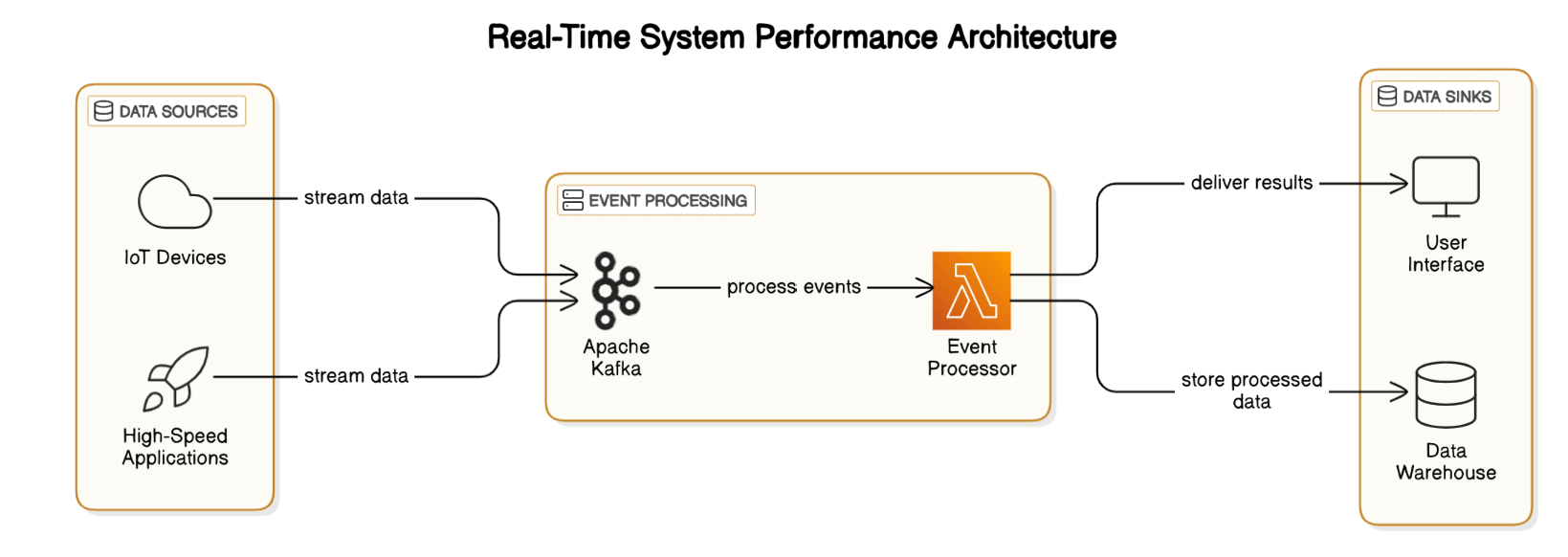
A diagram of a diagram

Description automatically generated

**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.

**Diagram:**



**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.

**1. Pipe and Filter Pattern**

**Step 1: Initial Monolithic Architecture**

* In the **Monolithic Architecture**, everything (order placement, browsing, searching) is handled in a single class.

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 Pipe and Filter Architecture**

* In **Pipe and Filter**, we separate the logic into distinct filters that handle different aspects of the process (inventory, order, search).

**Filter: InventoryFilter**

import java.util.HashMap;

import java.util.Map;

public class InventoryFilter {

private Map<String, Integer> inventory = new HashMap<>();

public InventoryFilter() {

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;

}

}

**Filter: OrderFilter**

public class OrderFilter {

public String placeOrder(InventoryFilter inventoryFilter, String item, int quantity) {

if (inventoryFilter.reduceStock(item, quantity)) {

return "Order placed successfully";

}

return "Order failed";

}

}

**Filter: SearchFilter**

java

Copy code

public class SearchFilter {

private InventoryFilter inventoryFilter;

public SearchFilter(InventoryFilter inventoryFilter) {

this.inventoryFilter = inventoryFilter;

}

public String searchItem(String item) {

return inventoryFilter.getInventory().containsKey(item) ?

"Available: " + inventoryFilter.getInventory().get(item) : "Item not found";

}

}

**Main Pipe and Filter Orchestration**

public class PipeAndFilterExample {

public static void main(String[] args) {

// Create filters

InventoryFilter inventoryFilter = new InventoryFilter();

OrderFilter orderFilter = new OrderFilter();

SearchFilter searchFilter = new SearchFilter(inventoryFilter);

// Pipe data through filters

System.out.println(orderFilter.placeOrder(inventoryFilter, "item1", 2)); // Order Filter

System.out.println(searchFilter.searchItem("item1")); // Search Filter

System.out.println("Inventory: " + inventoryFilter.getInventory()); // Inventory Filter

}

}

**2. Observer Pattern**

**Step 1: Initial Monolithic Architecture**

* In the **Monolithic Architecture**, all features (order placement, browsing, searching) are handled in a single class.

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 Observer Pattern**

* In **Observer Pattern**, we introduce the **Subject** (inventory) and **Observers** (order and search) that react to changes in the inventory.

**Subject: InventorySubject**

import java.util.HashMap;

import java.util.Map;

import java.util.ArrayList;

import java.util.List;

public class InventorySubject {

private Map<String, Integer> inventory = new HashMap<>();

private List<Observer> observers = new ArrayList<>();

public InventorySubject() {

inventory.put("item1", 10);

inventory.put("item2", 5);

}

public void addObserver(Observer observer) {

observers.add(observer);

}

public void removeObserver(Observer observer) {

observers.remove(observer);

}

public void notifyObservers() {

for (Observer observer : observers) {

observer.update(inventory);

}

}

public boolean reduceStock(String item, int quantity) {

if (inventory.containsKey(item) && inventory.get(item) >= quantity) {

inventory.put(item, inventory.get(item) - quantity);

notifyObservers(); // Notify observers of the change

return true;

}

return false;

}

public Map<String, Integer> getInventory() {

return inventory;

}

}

**Observer: OrderObserver**

public class OrderObserver implements Observer {

private String orderStatus = "Order not placed";

public String getOrderStatus() {

return orderStatus;

}

@Override

public void update(Map<String, Integer> inventory) {

// Handle inventory change

orderStatus = "Order processed with updated inventory";

}

public String placeOrder(InventorySubject inventorySubject, String item, int quantity) {

if (inventorySubject.reduceStock(item, quantity)) {

return "Order placed successfully";

}

return "Order failed";

}

}

**Observer: SearchObserver**

public class SearchObserver implements Observer {

private String searchStatus = "Searching...";

public String getSearchStatus() {

return searchStatus;

}

@Override

public void update(Map<String, Integer> inventory) {

// Handle inventory change

searchStatus = "Inventory updated";

}

public String searchItem(InventorySubject inventorySubject, String item) {

return inventorySubject.getInventory().containsKey(item) ?

"Available: " + inventorySubject.getInventory().get(item) : "Item not found";

}

}

**Observer Interface**

interface Observer {

void update(Map<String, Integer> inventory);

}

**Main Observer Pattern Orchestration**

public class ObserverPatternExample {

public static void main(String[] args) {

// Create the subject (Inventory)

InventorySubject inventorySubject = new InventorySubject();

// Create observers

OrderObserver orderObserver = new OrderObserver();

SearchObserver searchObserver = new SearchObserver();

// Register observers

inventorySubject.addObserver(orderObserver);

inventorySubject.addObserver(searchObserver);

// Place an order (Observer will be notified of changes)

System.out.println(orderObserver.placeOrder(inventorySubject, "item1", 2));

System.out.println(searchObserver.searchItem(inventorySubject, "item1"));

// View inventory and observe changes

System.out.println("Inventory: " + inventorySubject.getInventory());

System.out.println("Order Status: " + orderObserver.getOrderStatus());

System.out.println("Search Status: " + searchObserver.getSearchStatus());

}

}

**Summary of Transitions:**

1. **Pipe and Filter**:
   * We separated the different responsibilities (inventory management, order placement, and search) into independent filters.
   * Data flows through these filters to achieve the desired functionality.
2. **Observer**:
   * The **InventorySubject** acts as the subject that notifies **Observers** (OrderObserver and SearchObserver) of changes in the inventory.
   * Observers react to changes and update their status accordingly.

Both designs break the monolithic structure into modular components, either through sequential data processing (Pipe and Filter) or event-driven updates (Observer).

**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.