Assignment: Modernization and Scalability Challenges in Real-World Applications

By: Syed Hadi Raza – Fa22-BSE-057

Scenario 1: Scaling Real-Time Messaging Systems (Example: WhatsApp Outage)

• **Project Goal:** Build a scalable real-time messaging platform to handle millions of concurrent messages across a global user base.
• **Problem Faced: Latency and Throughput Limitations**
o **Details:** WhatsApp faced performance degradation during peak usage, especially during major global events or emergencies, causing delayed message delivery.
o **Cause:** The initial server architecture could not handle high throughput during peak traffic, leading to delays.
o **Solution Attempted:** Migration to a distributed message queue and optimization of the messaging protocol for better performance. However, challenges in managing the distributed system arose.

Scenario 2: Designing an Online Food Delivery App (Example: Uber Eats Performance Issues)

• **Project Goal:** Develop a highly efficient mobile app for ordering food, offering real-time order tracking and seamless payment integration.
• **Problem Faced: Geolocation and Real-Time Data Synchronization**
o **Details:** Uber Eats faced delays and inaccuracies in order tracking during peak hours, affecting user satisfaction.
o **Cause:** The geolocation API failed to synchronize real-time tracking data accurately across all devices.
o **Impact:** Orders were delivered late or to the wrong location, impacting customer experience.
o **Solution Attempted:** Upgraded the geolocation system and synchronized order status with a more efficient backend, but scalability was still a challenge.

Scenario 3: Managing a Cloud Storage System (Example: Google Drive Scaling Challenges)

• **Project Goal:** Build a cloud storage system capable of handling billions of files and supporting millions of concurrent users.
• **Problem Faced: File Management and Retrieval Performance**
o **Details:** Google Drive experienced performance bottlenecks as the number of users and files increased, especially when users requested large files simultaneously.

o **Cause:** The underlying file storage architecture struggled with concurrency, leading to slower file retrieval times during peak periods.
o **Solution Attempted:** Introduced distributed storage systems and sharding to balance the load, but managing consistency across shards posed additional difficulties.

Scenario 4: Real-Time Collaborative Document Editing (Example: Google Docs)

• **Project Goal:** Build a system that allows multiple users to collaborate on a document in real-time with minimal latency.
• **Problem Faced: Synchronization and Conflict Resolution in Real-Time Collaboration**
o **Details:** Google Docs faced issues with real-time collaboration, where multiple users editing the same document could experience data conflicts.
o **Cause:** The backend synchronization mechanism struggled to handle high volumes of concurrent edits without conflicts.
o **Impact:** Users experienced lost changes or data overwriting during simultaneous edits.
o **Solution Attempted:** Implemented an operational transformation algorithm to resolve conflicts and synchronize changes, but the solution required continuous tuning to improve performance.

Scenario 5: AI-Based Customer Support Chatbot (Example: Facebook Messenger AI Integration)

• **Project Goal:** Develop an AI-powered chatbot capable of handling customer queries and providing instant support across multiple platforms.
• **Problem Faced: Natural Language Understanding and User Context**
o **Details:** The chatbot faced difficulties understanding complex queries or providing personalized responses, leading to user frustration.
o **Cause:** The initial AI model was trained on generic data, lacking the necessary domain-specific understanding for personalized conversations.
o **Impact:** Users often had to escalate their queries to human agents, reducing the efficiency of the system.
o **Solution Attempted:** Retrained the model with domain-specific data and fine-tuned user context recognition, improving response quality, but maintaining accuracy was still an ongoing challenge.

Question 1: Architectural Design and Solution for Legacy System Integration in Banking

Legacy Architecture Used: Monolithic to Middleware

• **Monolithic Architecture:** Legacy banking systems are built on a monolithic structure where all components (transaction handling, data storage, etc.) are tightly coupled, making it hard to integrate with modern apps.
• **Modern Architecture:** Middleware bridges the gap between the modern banking app and the legacy monolithic system by enabling data exchanges in real-time.

Problem Faced:

• **Legacy System Compatibility:** The banking system lacked modern APIs or the capability to interact with new applications. • **Data Format Mismatch:** The legacy system used proprietary data formats, while the modern app required JSON or XML for smooth interaction.

Solution Proposed: Middleware Integration

1. **Middleware Setup:** Develop a middleware layer to fetch legacy data, convert it to modern formats, and serve it to the banking app.
2. **Legacy Data Extraction:** Middleware pulls data from the legacy system.
3. **Data Conversion:** Converts the old data format into a JSON format suitable for modern apps.
4. **Modern Application Use:** The processed data is now compatible with the banking app, providing real-time access to customer information.

Code Example:

Legacy System Class:

```
java
Copy code
class LegacyBankSystem {
    public String getCustomerData(String customerId) {
        // Simulate returning old format data
        return "Customer ID: " + customerId + ", Account Balance: 2000 USD";
    }
}
```

Middleware Class:

```
java
Copy code
import org.json.JSONObject;

class BankingMiddleware {
    private LegacyBankSystem legacySystem;
```

```java
    public BankingMiddleware() {
        this.legacySystem = new LegacyBankSystem();
    }

    public JSONObject getCustomerDataInModernFormat(String customerId) {
        // Fetch data from legacy system
        String legacyData = legacySystem.getCustomerData(customerId);

        // Split and process legacy data
        String[] dataParts = legacyData.split(", ");
        String customerIdVal = dataParts[0].split(": ")[1];
        String balanceVal = dataParts[1].split(": ")[1];

        // Format into JSON
        JSONObject modernData = new JSONObject();
        modernData.put("customerId", customerIdVal);
        modernData.put("accountBalance", balanceVal);

        return modernData;
    }
}
```

Modern Application:

```java
java
Copy code
public class ModernBankApp {
    public static void main(String[] args) {
        BankingMiddleware middleware = new BankingMiddleware();

        // Fetch customer data through middleware
        String customerId = "123456789";
        JSONObject customerData =
middleware.getCustomerDataInModernFormat(customerId);

        // Display the modernized data
        System.out.println("Customer Data: " + customerData.toString(2));
    }
}
```

Execution and Expected Output:

Execution Steps:

1. The modern app sends a request with a customer ID.
2. Middleware fetches legacy data from the old banking system.
3. Converts the legacy format into JSON.
4. The app displays the modernized account details.

Output:

```json
Customer Data: {
  "customerId": "123456789",
  "accountBalance": "2000 USD"
}
```

Benefits of the Middleware Solution:

1. **Modularity:** Keeps legacy systems intact while providing an interface for modern applications.
2. **Scalability:** Middleware can scale independently to support additional systems and integrations.
3. **Maintaining Legacy Systems:** The legacy system remains unaffected, reducing risk to existing operations.

Key Takeaways:

This example shows how middleware allows legacy systems to be integrated into modern applications without full replacement, ensuring data continuity while fostering modernization.