lab Task

Name Zain Zahoor

Reg No Fa21-Bse-166

Software Design and Architecture

Software architecture is backbone of software systems which refers to the high level structure or blueprint of a software system. It defines the organization components their interactions and the principles and guidelines use to build maintain and scale system.

Following is the Software which faces some architecture issue's in past

AMAZON:

Amazon's early software architecture faced several significant problems that hindered its ability to scale and meet the growing demands of its rapidly expanding business.

1. Monolithic architecture:

Amazon used a **Monolithic architecture**, meaning all the various components of the system such as product search, order processing, payment handling, and user management were tightly coupled and ran as a single, unified system.

Problems with monolithic architecture:

- **Scalability issues**:

As Amazon expanded its product catalog and attracted more users, the system struggled to scale effectively. The monolithic structure made it difficult to scale individual components independently.

- **Tight coupling**: All the components of the system were interdependent. This meant a failure in one component (like payment processing) could affect the entire application.
- **Deployment complexity**: As the application grew, deploying new features or updates became more difficult.
- **Slower innovation**: Because of the complexity of the monolithic system, teams working on different features or components of the system had to coordinate tightly. This slowed down innovation and development, as changes in one part of the application could have far-reaching consequences across the system.

2. Inability to Handle Increased Load:

As Amazon grew, the volume of traffic, user interactions, and data exponentially increased. The monolithic system couldn't handle this surge effectively. Specific workloads, such as processing customer orders, could become bottlenecks that reduces the entire system's performance.

Problems in handling increased load:

- **Performance degradation**: With more users and a broader product offering, certain services such as product search or order processing became slower, impacting the user experience.
- **Single points of failure**: A failure in any part of the monolithic system could cause a cascading failure, leading to outages that affected the entire platform.

  e.g: if the inventory system had a bug, the entire platform might experience issues.
- **Resource inefficiency**: Scaling the monolithic system to handle more users and data required adding more hardware or virtual machines but this was often inefficient.

3. **Difficulty in Adding New Features and Enhancements**:

Amazon's product offering rapidly expanded, and so did the features required to support the business. However, the monolithic architecture made it difficult to add new features quickly or improve existing ones without risking other parts of the system.

Problems with adding new features:

- **Long development cycles**: Since different parts of the application were tightly coupled, adding a new feature or changing an existing one required testing the entire system to ensure that no other parts were broken
- **Risk of introducing bugs**: When adding new functionality, there was a high risk of introducing bugs.
- **Poor fault isolation**: In a monolithic system, isolating faults or bugs in a specific feature was difficult.

4. Inability to Scale Development Teams:

With a monolithic architecture, development teams worked on a single large codebase, and there was a heavy reliance on centralized coordination. This became problematic as Amazon's engineering team grew, as it was harder to distribute development tasks efficiently across multiple teams.

Problems with scaling development teams:

- **Coordination overhead**: As the company grew, coordinating between teams responsible for different features became increasingly difficult.
- **Lack of autonomy**: Engineers working on different parts of the system often had little autonomy to make changes independently. This centralized approach slowed down progress and innovation.
-

5. Limited Flexibility and Agility:

Amazon was also growing its business into new markets, such as cloud computing and media streaming. The monolithic system lacked the flexibility to quickly adapt to these changes, as it was not modular enough to support new, separate business units.

Problems with flexibility:

- **Limited ability to innovate**: The rigid structure of the monolithic application made it difficult to rapidly introduce and scale new products or features.
- **Lack of technology flexibility**: In a monolithic system, updating or replacing technology stacks could be difficult because the entire system had to be modified.

Solution:

Transition to Service-Oriented Architecture (SOA) and Microservices:

Recognizing these issues, Amazon began transitioning away from the monolithic architecture to a **service-oriented architecture (SOA)** and later pioneered **microservices**. This shift fundamentally changed the way the platform was built and scaled.

Service-Oriented Architecture (SOA):

Service-Oriented Architecture (SOA) is a design pattern where software components, called **services**, are independently developed, deployed, and maintained. These services communicate with each other over a network to provide functionality. Each service is designed to perform a specific business function or process, and they interact using well-defined protocols (e.g., HTTP, SOAP, REST).

When Amazon transitioned to **SOA**, it marked a fundamental shift from their earlier **monolithic** architecture, where all components of the system were tightly coupled and existed in a single codebase. Moving to SOA allowed Amazon to break down their system into smaller, more manageable services, each focusing on a specific function, making the platform more scalable, flexible, and resilient.

Key Concepts of SOA:

1. **Services**: In SOA, applications are composed of discrete services. Each service performs a specific function or set of functions (e.g., user authentication, order processing, inventory management). These services are **loosely coupled** and communicate with each other through standard protocols.
2. **Interoperability**: Services in SOA are designed to communicate with one another over a network using standardized communication protocols such as HTTP, SOAP (Simple Object Access Protocol), and REST (Representational State Transfer). This allows services to run on different platforms or technologies but still interact seamlessly.
3. **Reusability**: Once a service is created, it can be reused across different applications or parts of the system. This reduces duplication and accelerates development time.
4. **Loose Coupling**: Each service in SOA operates independently, meaning changes to one service do not directly affect others. This makes it easier to develop, test, and maintain different parts of the system without worrying about breaking other areas.

5. **Scalability**: Because each service can be scaled independently, SOA enables more efficient resource allocation. If a particular service (e.g., payment processing) experiences high traffic, it can be scaled up without affecting other services like product search or user management.
6. **Standardized Communication**: SOA relies on a standardized way for services to interact. This could involve using web service protocols such as **SOAP** (which was more commonly used in early SOA implementations) or **RESTful APIs**, which allow communication over HTTP in a more lightweight and flexible manner.

Why Amazon Transitioned to SOA:

Before transitioning to SOA, Amazon's system faced major problems due to its **monolithic** design:

- **Single codebase**: All components of the system were tightly coupled, meaning a change in one part of the application affected the entire system.
- **Limited scalability**: Scaling the system required increasing resources across the entire application, even if only a specific component (like search or checkout) needed more power.
- **Slow development and updates**: Developers had to work on the same codebase, making it difficult to deploy new features without risking introducing bugs in other parts of the system.

The Process of Transitioning to SOA:

The transition to SOA involved breaking down Amazon's monolithic codebase into independent services that could communicate over a network. Here's how the transition unfolded:

1. **Breaking Down the Monolith**: Amazon's engineering team started by identifying the various functional areas within the platform (e.g., product search, customer reviews, order processing, payments). They then began developing individual services to handle each of these functions.
2. **Service Interfaces**: For services to communicate with each other, Amazon developed standardized service interfaces. These interfaces defined how each service would expose its functionality and how other services could request and receive information.

3. **Decoupling the Components**: Rather than having tightly coupled components, Amazon built each service to operate independently. This allowed the team to make changes to one service without affecting others.
4. **API Communication**: Services began communicating with each other using **APIs** (Application Programming Interfaces). For example, the payment processing service might call the inventory service via an API to check stock levels before completing a transaction. These APIs typically used REST or SOAP protocols for communication.
5. **Introducing Flexibility**: With SOA, Amazon's team could choose the best technology for each service. For example, the search service might use a different database technology compared to the order processing service, enabling more efficient and tailored performance.
6. **Scalability and Fault Isolation**: Each service could now be scaled independently. For instance, if traffic on the search service increased, it could be scaled without needing to scale other services like user management. Similarly, failures in one service could be contained without affecting the entire platform.

Benefits of SOA for Amazon:

1. **Scalability**: Amazon could scale different parts of the platform independently. If the payment service experienced high demand, it could be scaled without affecting the product search or customer review services.
2. **Flexibility**: Amazon could add new features or services to the platform without overhauling the entire system. For example, the introduction of **Amazon Web Services (AWS)** was made easier by the SOA architecture, as the existing system could integrate with AWS without disrupting Amazon's main retail platform.
3. **Faster Deployment**: By decoupling the system into smaller services, Amazon was able to speed up the deployment of new features. Different teams could work on different services simultaneously, leading to faster innovation and updates.
4. **Improved Fault Tolerance**: With independent services, if one service failed (e.g., order processing), it would not bring down the entire system. This increased the reliability of Amazon's platform.
5. **Easier Maintenance**: Each service could be maintained independently, meaning developers could focus on specific functionality without worrying about the impact on the rest of the platform.

6. **Better Resource Allocation**: Amazon could allocate computing resources more efficiently by scaling only the services that needed more resources (e.g., scaling the search service when demand was high) rather than scaling the entire system.

FOR SOLVING ADDING NEW FEATURES AND ENHANCEMENTS AT A SMALL STAGE :

To solve the problem of adding new features and enhancements at a small stage, especially in the context of **scalability** and **maintainability**, adopting a **modular** approach is essential. A modular design ensures that new features can be developed and deployed independently of the core system. One of the best ways to implement this is by using **Microservices** or **Modular Programming** within a single application.

Modular Programming (Single Application):

1. **Modular Programming (Single Application):**

- If you're dealing with a monolithic application, divide the application into smaller, manageable modules. Each module should have a well-defined responsibility and interface.
- Use Java packages or Java modules (Java 9 introduced the module system) to keep your codebase modular and easier to scale.
- This modular approach allows teams to focus on specific modules without interfering with others.

2. **Microservices (Distributed System):**

    - If you are building or maintaining a more complex system, consider using a microservices architecture, where each feature or enhancement is its own microservice.
    - Each microservice can be deployed and scaled independently.

- Use REST APIs or gRPC to enable communication between microservices.

Example: Modular Programming with Java:

In the example below, we simulate a small feature addition to a Library Management System. We'll modularize it into separate classes with clear responsibilities, which can be easily extended with additional features later.

```
// Book.java (Module for managing books)
public class Book {
  private String title;
  private String author;
  private String isbn;

  public Book(String title, String author, String isbn) {
    this.title = title;
    this.author = author;
    this.isbn = isbn;
  }

  public String getTitle() {
    return title;
  }

  public String getAuthor() {
    return author;
  }
```

```
    public String getIsbn() {
        return isbn;
    }

    public void displayBookDetails() {
        System.out.println("Title: " + title);
        System.out.println("Author: " + author);
        System.out.println("ISBN: " + isbn);
    }
}
```

Create a Service Class for Adding and Managing Books

```
// LibraryService.java (Module for managing book operations)
import java.util.ArrayList;
import java.util.List;

public class LibraryService {
    private List<Book> books;

    public LibraryService() {
        books = new ArrayList<>();
    }

    public void addBook(Book book) {
        books.add(book);
        System.out.println("Book added: " + book.getTitle());
```

```
    }

    public void displayAllBooks() {
        for (Book book : books) {
            book.displayBookDetails();
        }
    }
}
```

Add New Feature - Search for Books by Title

```
// SearchService.java (Module for searching books)
public class SearchService {
    public static void searchBookByTitle(LibraryService libraryService, String title) {
        boolean found = false;
        for (Book book : libraryService.getBooks()) {
            if (book.getTitle().equalsIgnoreCase(title)) {
                book.displayBookDetails();
                found = true;
                break;
            }
        }
        if (!found) {
            System.out.println("Book with title '" + title + "' not found.");
        }
    }
}
```

Testing the System

```
public class Main {
    public static void main(String[] args) {
        // Create LibraryService instance
        LibraryService libraryService = new LibraryService();

        // Add books
        libraryService.addBook(new Book("The Catcher in the Rye", "J.D. Salinger", "9780316769488"));
        libraryService.addBook(new Book("1984", "George Orwell", "9780451524935"));

        // Display all books
        libraryService.displayAllBooks();

        // Search for a book by title
        System.out.println("\nSearching for '1984':");
        SearchService.searchBookByTitle(libraryService, "1984");

        // Searching for a non-existent book
        System.out.println("\nSearching for 'Non-existent Book':");
        SearchService.searchBookByTitle(libraryService, "Non-existent Book");
    }
}
```