Microservice Architecture

CSE Seminar

S Vigneshwaran
21075073
CSE (B.Tech) 4th year

What is a microservice ?

A microservices architecture structures an application as a collection of small, independently deployable services.

We all know that Monolithic architecture consist only of a single server which handles all of the functionalities.

Example :

Monolithic: A single large e-commerce application.
Microservices: Separate services for **Users**, **Orders**, **Payments**, **Notifications**.

Monolithic vs Microservice

Properties	Monolithic	Microservice
Scalability	Limited	High
Deployment	Entire app redeployed	Independent service redeployed
Tech-Stack	Single Tech-stack	Multiple Tech-stack
Fault Isolation	Low	High

Advantages of Microservices

1. Better Scalability

Microservices allow individual components to scale **independently** based on demand, improving performance and resource utilization.

Example:

- In an **e-commerce app**, the **payment service** can scale separately from the **product catalog**.
- If millions of users browse products but only thousands make purchases, the product service can have **10 instances**, while the payment service has **2 instances** to optimize costs.

2. Faster Development & Deployment

Each microservice is developed **independently**, enabling **parallel development** and **faster releases**.

Example:

- A **team** can work on the **authentication service**, while another team develops the **order management service**.
- Since each microservice is **separate**, it can be **deployed without affecting the entire application**.

How microservices communicate within each other ?

1.REST API

REST (Representational State Transfer) is a widely used architectural style for building web services. It relies on standard HTTP methods such as `GET`, `POST`, `PUT`, and `DELETE` to interact with resources, typically represented in **JSON or XML** format. Ideal for web applications, mobile apps, and microservices..

2.gRPC

gRPC (Google Remote Procedure Call) is a high-performance RPC (Remote Procedure Call) framework that enables **efficient communication between services** using **Protocol Buffers (Protobuf)**. It uses a **binary format**, making it much faster and more efficient, especially for high-throughput applications..gRPC is **not natively supported in browsers**, requiring gRPC-Web or a REST Gateway to communicate with frontend applications.

3.Message Queues

Message Queues (MQ) are used for **asynchronous communication** between services, where messages are sent to a queue and processed later by consumers. Popular MQ systems like **RabbitMQ, Apache Kafka, and ActiveMQ** handle large volumes of messages, making them ideal for **background processing, logging, real-time event streaming, and load balancing**.

REST APIs

Types : GET,POST,PUT,DELETE

Mainly used in client-server architecture - (frontend and backend communication and backend microservices communication)

Pros:

- Simple & widely adopted
- Language-agnostic (any client can call REST APIs)
- Human-readable (JSON responses)

Cons:

- Slower compared to gRPC (due to JSON text-based format)
- Lacks built-in real-time streaming

GET request : This is used from getting the resource (data) from one microservice.The data is received in form of a JSON/XML(Extensible Markup Language)

```
    @GetMapping(value = "/useridfromjwt")
    public Long userIdFromJwt(String jwt){
        return userService.userIdFromJwt(jwt);
    }
}
```

```
const userid = await axios.get(
  "http://localhost:8080/usermanagement/api/user/useridfromjwt" +
    "?jwt=" +
    localStorage.getItem("jwtToken")
);
```

POST request

This request is used to send data from one microservice to other

```java
@PostMapping(value = "/register")
private String userRegistration(RegisterRequest registerRequest){
    return authService.register(registerRequest);
}
```

```javascript
const response = await axios.post(
  "http://localhost:8080/usermanagement/api/auth/register" +
    "?userName=" +
    userName +
    "&userPhone=" +
    userPhone +
    "&userEmail=" +
    userEmail +
    "&userPassword=" +
    userPassword +
    "&userRoles=" +
    userRole
);
```

PUT request

PUT is used for updating the data

```
@PutMapping(value = "/edit/{user_id}")
public String editDetails(@PathVariable("user_id") Long user_id, User user){
    user.setUserId(user_id);
    return userService.editUserDetails(user);
}
```

DELETE request

DELETE is used for deleting the data

```
@DeleteMapping("/clearCart")
public void clearCart(){
    cartService.clearCart();
}
```

gRPC

What is RPC ?

RPC - remote procedure calling

Remote Procedure Call (RPC) is a protocol that allows a program to execute a function on another server **as if it were a local function call.**

In **March 2015, Google** invented a framework for modern communication within microservices - gRPC (google Remote Procedure Calling)

RPC is an old concept used for remote function calls.

gRPC is a modern, fast, and efficient version of RPC.

Microservices **can use RPC, but gRPC is preferred over traditional RPC** for high-performance communication.

What is gRPC ?

gRPC is a **high-performance, open-source RPC (Remote Procedure Call) framework** developed by **Google**. It enables efficient communication between microservices using **HTTP/2** and **Protocol Buffers (protobufs)** instead of JSON or XML.

gRPC Architecture Overview

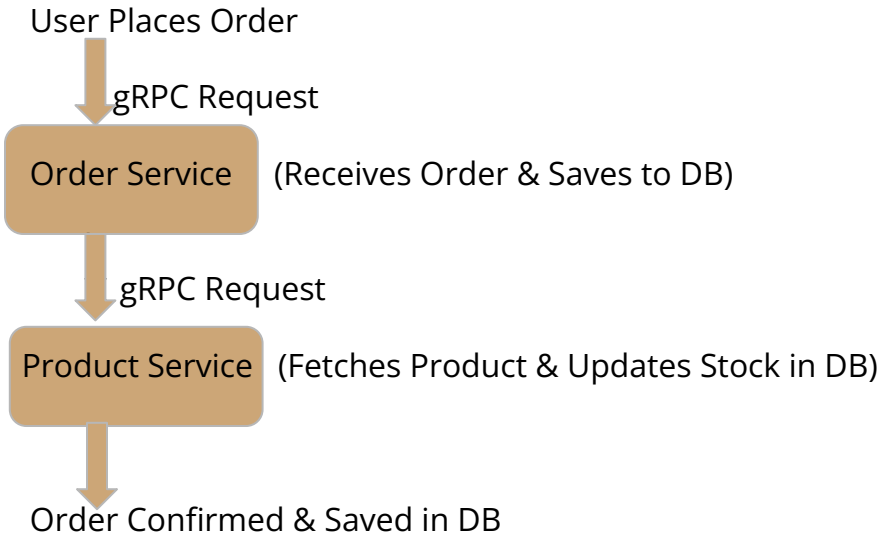
1. **Client** sends a request using a gRPC stub*.
2. **Server** receives the request, processes it, and responds.
3. Uses **Protocol Buffers (protobufs)** for message exchange (binary format, faster than JSON).

Protobuf (Protocol Buffers) :

Protocol Buffers (Protobuf) is a language-neutral, platform-independent, and extensible mechanism for serializing structured data. Think of Protobuf as a faster, more efficient alternative to JSON or XML.Protobuf uses a compact binary format, making it faster and smaller than JSON.Binary format Objects which act as an alternative to JSON .

*stub is a piece of code that converts parameters passed between the client and server during a remote procedure call (RPC). .

Example of communication using gRPC



The **client (Order Service)** will now get data **from the database** instead of hardcoded values.

```java
public class OrderServiceClient {
    public static void main(String[] args) {
        // Connect to ProductService
        ManagedChannel channel = ManagedChannelBuilder.forAddress("localhost", 9090)
                .usePlaintext()
                .build();

        ProductServiceGrpc.ProductServiceBlockingStub stub = ProductServiceGrpc.newBlockingStub(channel);

        // Request product details from database
        ProductRequest request = ProductRequest.newBuilder().setProductId(1).build();
        ProductResponse response = stub.getProduct(request);

        // Print product details
        System.out.println("Product ID: " + response.getProductId());
        System.out.println("Name: " + response.getName());
        System.out.println("Price: $" + response.getPrice());
        System.out.println("Description: " + response.getDescription());

        channel.shutdown();
    }
}
```

Sequence of action :

- ProductService fetches product details from MySQL.
- OrderService acts as a gRPC client, requesting product details.
- gRPC efficiently serializes & transmits data using Protobuf.

Let's consider an **eCommerce app** where we have **two microservices** communicating using **gRPC** and **Protobuf**.

1. **Product Service** → Manages product details
2. **Order Service** → Places orders and fetches product details from Product Service

```
no usages
@Override
public void getProduct(ProductRequest request, StreamObserver<ProductResponse> responseObserver) {
    int productId = request.getProductId();

    try (Connection conn = DriverManager.getConnection(JDBC_URL, JDBC_USER, JDBC_PASSWORD);
         PreparedStatement stmt = conn.prepareStatement( sql: "SELECT * FROM products WHERE id = ?")) {

        stmt.setInt( parameterIndex: 1, productId);
        ResultSet rs = stmt.executeQuery();

        if (rs.next()) {
            ProductResponse response = ProductResponse.newBuilder()
                    .setProductId(rs.getInt( columnLabel: "id"))
                    .setName(rs.getString( columnLabel: "name"))
                    .setPrice(rs.getDouble( columnLabel: "price"))
                    .setDescription(rs.getString( columnLabel: "description"))
                    .build();

            responseObserver.onNext(response);
        } else {
            responseObserver.onError(new RuntimeException("Product not found"));
        }
    } catch (SQLException e) {
        responseObserver.onError(e);
    } finally {
        responseObserver.onCompleted();
    }
}
```

Connected to MySQL database.

Fetched product details based on `product_id`.

Returned the data as a gRPC response.

gRPC vs REST API

Which one is better ? When to use which one ?

Features	**REST (JSON)**	**gRPC (ProtoBuf)**
Speed	REST **uses JSON**, which is **and larger in size**, **slower** .	gRPC uses ProtoBuf which is **binary format**, **faster** than JSON.
Communication Protocol	**REST opens & closes connections multiple times**, causing overhead.	gRPC can send multiple requests over a single connection
Streaming	No support	Live streaming supported
Use case	Web apps, external/public APIs, and simple CRUD apps.	Real-time apps, and high-performance systems.

Thank You

Please feel free to ask questions based on this seminar