**Microservice Architecture**

Microservice architecture is a software development approach where a single application is composed of small, loosely coupled, independently deployable services, each responsible for a specific business function. These services are organized around business capabilities and communicate with each other through well-defined APIs, typically over HTTP/REST or messaging protocols.

Here are some key characteristics and principles of microservice architecture:

* Decomposition: Decompose the application into smaller services based on business capabilities or bounded contexts. Each service should be focused on doing one thing well.
* Independence: Microservices are independent entities, each with their own codebase, data storage, and development/deployment lifecycle. This independence allows teams to work autonomously and release updates without affecting other services.
* Technology Diversity: Each microservice can be implemented using the most suitable technology stack for its requirements. This enables teams to choose the best tools and frameworks for their specific needs.
* Resilience: Microservices are designed to be resilient to failures. They should be able to handle failures gracefully, recover quickly, and degrade functionality when necessary. Implementing techniques like circuit breakers, retries, and timeouts can enhance resilience.
* Scalability: Microservices can be independently scaled based on demand. This allows you to allocate resources efficiently and scale only the services that require more capacity, rather than scaling the entire application.
* Data Management: Each microservice should have its own database or data store, ideally following the principles of bounded context from Domain-Driven Design (DDD). This avoids tight coupling between services and minimizes the risk of data inconsistency.
* Communication: Services communicate with each other through well-defined APIs. RESTful APIs over HTTP or messaging protocols like AMQP or Kafka are commonly used for inter-service communication.
* Deployment Automation: Microservices are typically deployed using containerization technologies like Docker and orchestrated using platforms like Kubernetes. This enables automated deployment, scaling, and management of services.
* Continuous Delivery: Embrace DevOps practices and implement continuous integration and continuous delivery (CI/CD) pipelines to automate the testing, building, and deployment of microservices.
* Monitoring and Observability: Implement comprehensive monitoring and logging solutions to gain visibility into the health and performance of microservices. Tools like Prometheus, Grafana, and ELK stack are commonly used for monitoring microservices architectures.
* Evolutionary Design: Microservices architectures are designed to be evolutionary. Services can be added, modified, or replaced over time without disrupting the entire system. This flexibility allows the architecture to adapt to changing business requirements and technological advancements.

A demo project was done to illustrate the Microservices.

**UserService:**

**Dependencies used for UserService:**

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework. boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>${spring-cloud.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

This project consists of three services UserService, ContactService and Api Gateway. The Eureka Server is used as Service discovery agent in the project.

**User class (Entity):**

The parameterized constructor containing userId,name and phone is created here. The variable List<Contacts> is created as we have the Contact entity class in the entity package.

package com.user.entity;

import java.util.ArrayList;

import java.util.List;

public class User {

private Long userId;

private String name;

private String phone;

List<Contact> contacts=new ArrayList<>();

public User(Long userId, String name, String phone, List<Contact> contacts) {

this.userId = userId;

this.name = name;

this.phone = phone;

this.contacts = contacts;

}

public User(Long userId, String name, String phone) {

this.userId = userId;

this.name = name;

this.phone = phone;

}

public User() {

}

public Long getUserId() {

return userId;

}

public void setUserId(Long userId) {

this.userId = userId;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getPhone() {

return phone;

}

public void setPhone(String phone) {

this.phone = phone;

}

public List<Contact> getContacts() {

return contacts;

}

public void setContacts(List<Contact> contacts) {

this.contacts = contacts;

}

}

**Contact class (Entity):**

public class Contact {

private Long cId;

private String email;

private String contactName;

private Long userId;

public Contact (Long cId, String email, String contactName, Long userId) {

this.cId = cId;

this.email = email;

this.contactName = contactName;

this.userId = userId;

}

public Contact () {

}

public Long getcId() {

return cId;

}

public void setcId(Long cId) {

this.cId = cId;

}

public String getEmail() {

return email;

}

public void setEmail(String email) {

this.email = email;

}

public String getContactName() {

return contactName;

}

public void setContactName(String contactName) {

this.contactName = contactName;

}

public Long getUserId() {

return userId;

}

public void setUserId(Long userId) {

this.userId = userId;

}

}

**UserService interface:**

The userService interface is created with the method getUser(Long id) in it.

public interface UserService {

public User getUser(Long id);

}

**UserServiceImpl class:**

Here we created the List of the users, and we are finding the particular user with the id.

@Service

public class UserServiceImpl implements UserService {

List<User> list = List.of(

new User (1311L, "Suraj", "23525625"),

new User (1312L, "Vishwanath", "99999"),

new User (1314L, "Prajwal", "888");

@Override

public User getUser(Long id) {

return list.stream().filter(user -> user.getUserId().equals(id)).findAny().orElse(null);

}

}

**User Controller class:**

The user is fetched through the id in the below method with <http://localhost:9001/user/1311> api

@GetMapping("/{userId}")

public User getUser(@PathVariable("userId") Long userId) {

User user = userService.getUser(userId);

return user;

}

**Contact Service:**

**Dependencies used for the Contact Service:**

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>${spring-cloud.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

**Contact Class (Entity):**

public class Contact {

private Long cId;

private String email;

private String contactName;

private Long userId;

public Contact(Long cId, String email, String contactName, Long userId) {

this.cId = cId;

this.email = email;

this.contactName = contactName;

this.userId = userId;

}

public Contact() {

}

public Long getcId() {

return cId;

}

public void setcId(Long cId) {

this.cId = cId;

}

public String getEmail() {

return email;

}

public void setEmail(String email) {

this.email = email;

}

public String getContactName() {

return contactName;

}

public void setContactName(String contactName) {

this.contactName = contactName;

}

public Long getUserId() {

return userId;

}

public void setUserId(Long userId) {

this.userId = userId;

}

**ContactService Interface:**

ContactService interface is created with List<Contact> getContactsOfUser(Long userId) method in it.

public interface ContactService {

public List<Contact> getContactsOfUser(Long userId);

}

**ContactServiceImpl class:**

The List<Contact> is implemented here.

@Service

public class ContactServiceImpl implements ContactService {

List<Contact> list = List.of(

new Contact(1L, "amit@gmail.com", "Amit", 1311L),

new Contact(2L, "anil@gmail.com", "Anil", 1311L),

new Contact(3L, "rohan@gmail.com", "Rohan", 1312L),

new Contact(4L, "virat@gmail.com", "Virat", 1314L)

);

@Override

public List<Contact> getContactsOfUser(Long userId) {

return list.stream().filter(contact -> contact.getUserId().equals(userId)).collect(Collectors.toList());

}

}

**ContactController Class:**

Here the list of contacts is fetched through the <http://localhost:9002/contact/user/1311> api.

@RestController

@RequestMapping("/contact")

public class ContactController {

@Autowired

private ContactService contactService;

@RequestMapping("/user/{userId}")

public List<Contact> getContacts(@PathVariable("userId") Long userId) {

return this.contactService.getContactsOfUser(userId);

}

}

**RestTemplate**

In Spring Boot, RestTemplate is a powerful class provided by the Spring Framework that simplifies consuming RESTful web services. It provides a straightforward way to make HTTP requests to remote servers and process the responses.

We can use different methods provided by RestTemplate to handle the response. For instance, getForObject() returns the response body as an object of the specified type, while getForEntity() returns a ResponseEntity object that provides access to the response body, headers, and status code.

Here we are accessing the endpoint of contactservice concatenated with userId with the help of RestTemplate which will give the contacts associated with the particular userId.

Before this, The RestTemplate bean must be written in the main application.

In Spring Boot, @LoadBalanced is an annotation used in conjunction with RestTemplate or WebClient to enable client-side load balancing. It's particularly useful in microservices architectures where multiple instances of a service are running, and requests need to be distributed among them.

When you annotate a RestTemplate or WebClient bean with @LoadBalanced, Spring Boot automatically integrates it with client-side load balancing provided by Spring Cloud.

In our application, we have only one instance of each service.

@Bean

@LoadBalanced

public RestTemplate restTemplate() {

return new RestTemplate();

}

}

@Autowired

private UserService userService;

@Autowired

private RestTemplate restTemplate;

@GetMapping("/{userId}")

public User getUser(@PathVariable("userId") Long userId) {

User user = userService.getUser(userId);

//http://localhost:9002/contact/user/1311

List contacts = restTemplate.getForObject("http://contact-service/contact/user/" + user.getUserId(), List.class);

user.setContacts(contacts);

return user;

}

**Eureka Server:**

Eureka Server is a component of Netflix OSS (Open Source Software) that provides service discovery in a microservices architecture. It allows services to register themselves and discover other services in the system. Eureka follows the server-client architecture, where the Eureka Server acts as the service registry, and the microservices act as clients that register with and discover other services through the server.

Here's an overview of how Eureka Server works:

* Service Registration: Microservices register themselves with the Eureka Server during startup. Each microservice instance (node) provides metadata about itself, such as its hostname, port, and health status.
* Heartbeats and Health Checks: After registration, microservices send heartbeats to the Eureka Server at regular intervals to indicate that they are still alive. The Eureka Server keeps track of the health status of registered services based on these heartbeats. Additionally, microservices can implement health checks to provide more detailed health information.
* Service Discovery: Microservices use Eureka Client to discover other services in the system. They query the Eureka Server to obtain the list of available instances of a particular service. Eureka provides client-side load balancing, enabling microservices to distribute requests among multiple instances of a service.
* Self-Preservation Mode: Eureka Server has a self-preservation mechanism to protect against network partitions and prevent the accidental removal of healthy instances due to temporary network issues. If the Eureka Server detects that the number of renewals falls below a certain threshold within a given time window, it switches to self-preservation mode and stops expiring instances.
* High Availability: To ensure high availability, you can run multiple instances of Eureka Server in a cluster configuration. Each instance replicates its state to other instances in the cluster, providing redundancy and fault tolerance. Clients can be configured with multiple Eureka Server URLs to handle failover and load distribution.

Setting up Eureka Server in a Spring Boot application involves adding the spring-cloud-starter-netflix-eureka-server dependency and annotating a configuration class with

@EnableEurekaServer

**Dependencies used:**

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>${spring-cloud.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

@SpringBootApplication

@EnableEurekaServer

public class EServerApplication {

public static void main (String[] args) {

SpringApplication.run(EServerApplication.class, args);

}

}

A screenshot of a computer

Description automatically generated

Each service’s application.yml file should be configured with eureka server details as per below.

eureka:

instance:

hostname: localhost

**Api Gateway:**

In a microservices architecture, an API Gateway is a critical component that acts as a single entry point for clients to access various microservices. It serves as a reverse proxy, routing requests from clients to the appropriate microservices, aggregating responses, and handling cross-cutting concerns such as authentication, authorization, rate limiting, logging, and monitoring. Here are some key aspects of API Gateways in microservices:

* Unified Entry Point: API Gateway provides a unified entry point for clients to interact with the system. Instead of directly calling individual microservices, clients make requests to the API Gateway, which then forwards those requests to the appropriate services.
* Routing and Aggregation: API Gateway routes incoming requests to the appropriate microservices based on predefined routing rules. It can also aggregate data from multiple microservices to compose a single response for the client, reducing the number of round trips.
* Protocol Translation: API Gateway can handle protocol translation, allowing clients to communicate using different protocols (e.g., HTTP, WebSocket) while internally interacting with microservices using a standardized protocol.
* Authentication and Authorization: API Gateway centralizes authentication and authorization logic, allowing it to authenticate clients, enforce access control policies, and validate JWT tokens or OAuth tokens before forwarding requests to microservices.
* Rate Limiting and Throttling: API Gateway can enforce rate limiting and throttling policies to prevent abuse and ensure fair usage of resources. It can limit the number of requests per second per client or API key and throttle requests when necessary.
* Load Balancing: API Gateway can perform client-side or server-side load balancing to distribute incoming requests across multiple instances of microservices, improving scalability and fault tolerance.
* Caching: API Gateway can cache responses from microservices to reduce latency and improve performance. It can cache both static and dynamic content based on caching rules and expiration policies.
* Monitoring and Logging: API Gateway provides centralized monitoring and logging capabilities, allowing administrators to monitor request/response metrics, track errors, and troubleshoot issues effectively.
* Security: API Gateway can enforce security measures such as SSL termination, request/response encryption, and input validation to protect against common security threats like injection attacks and XSS attacks.
* Service Discovery: API Gateway integrates with service discovery mechanisms like Eureka or Consul to dynamically discover and route requests to available instances of microservices. This enables seamless integration with dynamic environments where microservices are frequently deployed and scaled.

It acts as first level security for us where without knowing the actual port of the service, we can access them with port number of Api Gateway. The @LoadBalanced for the RestTemplate should be done for RestTemplate bean as we will pass the service name in the “uri: lb://user-service” in the application.yml file

**Application.yml file**

server:

port: 8999

eureka:

instance:

hostname: localhost

spring:

application:

name: api-gateway

cloud:

gateway:

routes:

- id: user-service

uri: lb://user-service

predicates:

- Path=/user/\*\*

- id: contact-service

uri: lb://contact-service

predicates:

- Path=/contact/\*\*