

## =====

### Microservices

## =====

- 1) Monolith Architecture
- 2) Drawbacks of Monolithic
- 3) Microservices Architecture
- 4) Pros & Cons with Microservices
- 5) Service Registry (Eureka)
- 6) Admin Server
- 7) Zipkin Server
- 8) FeignClient (Interservice Comm)
- 9) Load Balancer (Ribbon)
- 10) API Gateway (Filters + Routing)
- 11) Config Server
- 12) Circuit Breaker (Resilience4J)
- 13) Spring Boot with Kafka
- 14) Spring Boot with Redis
- 15) Docker
- 16) Kubernetes
- 17) Jenkins
- 18) Spring Security
- 19) Spring Batch
- 20) Unit Testing (JUnit)
- 21) Angular Integration

## =====

### Monolith Architecture

## =====

=> Developing all functionalities in single application.

- 1) Presentation Layer
- 2) Business Layer
- 3) Data Access Layer

=> Drawbacks with Monolith Architecture

- 1) Burden on Server
- 2) Response Delay
- 3) Server can crash
- 4) Single Point of failure
- 5) Technology Dependent
- 6) Re-Deploy entire app

=> To overcome problems of Monolith Architecture, people are using Microservices Architecture.

## =====

### Microservices

## =====

=> It is not a technology

=> It is not a framework

=> It is not an API

=> It is an architectural design pattern

=> It is universal and anyone can use this architecture to develop applications.

#### =====

#### Advantages with Microservices

#### =====

- 1) Loosely Coupled
- 2) Burden Reduced on Servers
- 3) Easy Maintenance
- 4) No Single point of failure
- 5) Technology Independent
- 6) Quick deliveries

#### =====

#### Challenges with Microservices

#### =====

- 1) Bounded Context
- 2) Repeated configurations
- 3) Visibility

=> Bounded context means identifying how many microservices we need to develop for one application and deciding which functionality we need to add in which microservice.

=> In Several microserices we need to write same configurations like data source, smtp, kafka, redis etc....

=> In microservice architecture we might not get chance to work with all apis in the application.

#### =====

#### Microservices Architecture

#### =====

- 1) Service Registry (Eureka)
- 2) Admin Server
- 3) Zipkin Server
- 4) Config Server
- 5) Kafka Server
- 6) Redis Server
- 7) API Gateway
- 8) Interservice communication

#### =====

#### Service Registry

#### =====

=> Service Registry is used to maintain all apis information like name, status, url and health at once place.

=> It is also called as Service Discovery.

=> We can use Eureka Server as service registry.

=> It will provide user interface to get apis info.

=====  
Admin Server  
=====

=> It is used to monitor and manage all the apis at one place

=> It provides beautiful user interface to access all apis actuator endpoints at one place.

=====  
Zipkin Server  
=====

=> It is used for distributed tracing of our requests

=> It provides beautiful user interface to access apis execution details.

=====  
Config Server  
=====

=> It is used to separate application code and application properties.

=> It is used to externalize config props of our application.

=> It makes our application loosely coupled with properties file or yaml file.

=====  
FeignClient  
=====

=> It is used for interservice communication

=> If one api communicate with another api with in the same application then it is called as Inter service communication.

=====  
Kafka Server  
=====

=> It is used as message broker

=> Distributed streaming platform

=> It works based on pub-sub model

=====  
Redis Server  
=====

=> Redis is a cache server

=> Redis represents data in key-value format

=> Redis is used to reduce no.of db calls

=====  
API Gateway  
=====

=> It acts as Entry point for all backend apis

=> It acts mediator between frontend app and backend apis.

=> In API Gateway we will write filters + Routings

Filter : We can perform pre-process & post-process

Routings : To forward request to particular backend-api.

=====

Steps to develop Service Registry Application (Eureka Server)

=====

1) Create Service Registry application with below dependency

- EurekaServer (spring-cloud-starter-netflix-eureka-server)

2) Configure @EnableEurekaServer annotation in boot start class

3) Configure below properties in application.yml file

```
server:
  port: 8761
```

```
eureka:
  client:
    register-with-eureka: false
```

Note: If Service-Registry project port is 8761 then clients can discover service-registry and will register automatically with service-registry. If service-registry project running on any other port number then we have to register clients with service-registry manually.

4) Once application started we can access Eureka Dashboard using below URL

URL : <http://localhost:8761/>

=====

Steps to develop Spring Admin-Server

=====

1) Create Boot application with admin-server dependency  
(select it while creating the project)

2) Configure @EnableAdminServer annotation at start class

3) Change Port Number (Optional)

4) Run the boot application

5) Access application URL in browser (We can see Admin Server UI)

=====

Steps to work with Zipkin Server

=====

1) Download Zipin Jar file

URL : <https://zipkin.io/pages/quickstart.html>

2) Run zipkin jar file

\$ java -jar <jar-name>

3) Zipkin Server Runs on Port Number 9411

4) Access zipkin server dashboard

URL : <http://localhost:9411/>

```
#####  
Steps to develop WELCOME-API  
#####
```

1) Create Spring Boot application with below dependencies

- eureka-discovery-client
- starter-web
- devtools
- actuator
- zipkin
- admin-client

2) Configure @EnableDiscoveryClient annotation at boot start class

3) Create RestController with required method

4) Configure below properties in application.yml file

```
-----application.yml-----  
server:  
  port: 1111  
  
spring:  
  application:  
    name: WELCOME-API  
  
  boot:  
    admin:  
      client:  
        url: http://localhost:9090/  
eureka:  
  client:  
    serviceUrl:  
      defaultZone: http://localhost:8761/eureka  
  
management:  
  endpoints:  
    web:  
      exposure:  
        include: '*'
```

5) Run the application and check in Eureka Dashboard (It should display in eureka dashboard)

6) Check Admin Server Dashboard (It should display) (we can access application details from here)

Ex: Beans, loggers, heap dump, thred dump, metrics, mappings etc...

7) Send Request to REST API method

8) Check Zipkin Server UI and click on Run Query button  
(it will display trace-id with details)

```
#####
Steps to develop GREET-API
#####
```

1) Create Spring Boot application with below dependencies

- eureka-discovery-client
- starter-web
- devtools
- actuator
- zipkin
- admin-client
- openfeign

2) Configure @EnableDiscoveryClient annotation at boot start class

3) Create RestController with required method

4) Configure below properties in application.yml file

```
-----application.yml-----
server:
  port: 2222

spring:
  application:
    name: GREET-API

  boot:
    admin:
      client:
        url: http://localhost:9090/

management:
  endpoints:
    web:
      exposure:
        include: '*'

-----
```

5) Run the application and check in Eureka Dashboard (It should display in eureka dashboard)

6) Check Admin Server Dashboard (It should display) (we can access application details from here)

Ex: Beans, loggers, heap dump, thred dump, metrics, mappings etc...

7) Send Request to REST API method

8) Check Zipkin Server UI and click on Run Query button  
(it will display trace-id with details)

```
=====
Interservice communication
=====
```

=> Add @EnableFeignClients dependency in GREET-API boot start class

=> Create FeignClient interface like below

```
@FeignClient(name = "WELCOME-API")
public interface WelcomeApiClient {
```

```
@GetMapping("/welcome")
public String invokeWelcomeMsg();
```

```
}
```

=> Inject feign client into GreetRestController like below

```
@RestController
public class GreetRestController {

    @Autowired
    private WelcomeApiClient welcomeClient;

    @GetMapping("/greet")
    public String getGreetMsg() {

        String welcomeMsg = welcomeClient.invokeWelcomeMsg();

        String greetMsg = "Good Morning, ";

        return greetMsg.concat(welcomeMsg);
    }
}
```

=> Run the applications and access greet-api method

(It should give combined response)

```
=====
Load Balancing
=====
```

=> Distribute requests to multiple servers

=> Run welcome-api in multiple instances.

1) Remove port number configuration welcome api yaml file

2) Make changes in rest controller to display port number in response.

3) Right click => Run as => run configuration => select welcome-api => VM Arguments => -Dserver.port=8081 and apply and run it.

4) Right click => Run as => run configuration => select welcome-api => VM Arguments => -Dserver.port=8082 and apply and run it.

```
#####
Working with Spring Cloud API Gateway
#####
```

1) Create Spring boot application with below dependencies

```
-> eureka-client
-> cloud-gateway
-> devtools
```

2) Configure @EnableDiscoveryClient annotation at boot start class

3) Configure API Gateway Routings in application.yml file like below

```
-----application.yml file-----
server:
  port: 3333
```

```

spring:
  cloud:
    gateway:
      routes:
        - id: welcome-api
          uri: lb://WELCOME-API
          predicates:
            - Path=/welcome
        - id: greet-api
          uri: lb://GREET-API
          predicates:
            - Path=/greet

    application:
      name: CLOUD-API-GATEWAY

```

---

welcome-api ==> 2 instances ==> 8081 & 8082 ==> /welcome

greet-api ==> 1 instance ==> 2222 ==> /greet

api-gateway ==> 1 instance ==> 3333

http://localhost:3333/welcome

http://localhost:3333/greet

---

In API gateway we will have 3 types of logics

1) Routes

2) Predicates

3) Filters

-> Routing is used to defined which request should be processed by which REST API in backend. Routes will be configured using Predicate.

-> Predicate : This is a Java 8 Function Predicate. The input type is a Spring Framework ServerWebExchange. This lets you match on anything from the HTTP request, such as headers or parameters or url-patterns.

-> Filters are used to manipulate incoming request and outgoing response of our application.

Note: Using Filters we can implement security also for our application.

---

```

@Component
public class MyFilter implements GlobalFilter {

    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {

        System.out.println("filter ( ) - executed.....");

        ServerHttpRequest request = exchange.getRequest();

        HttpHeaders headers = request.getHeaders();
        Set<String> keySet = headers.keySet();

```



```

        // validate request
        if(!keySet.contains("secret")) {
            throw new RuntimeException("Invalid Request");
        }

        List<String> list = headers.get("secret");
        if(!list.get(0).equals("ashokit@123")) {
            throw new RuntimeException("Invalid Token");
        }

        return chain.filter(exchange);
    }
}

```

- 
- > We can validate client given token in the request using Filter for security purpose
  - > We can write request and response tracking logic in Filter
  - > Filters are used to manipulate request & response of our application
  - > Any cross-cutting logics like security, logging, monitoring can be implemented using Filters

=====

What is Cloud Config Server

=====

=> We are configuring our application config properties in application.properties or application.yml file

Ex: DB Props, SMTP props, Kafka Props, App Messages etc...

=> application.properties or application.yml file will be packaged along with our application (it will be part of our app jar file)

=> If we want to make any changes to properties then we have to re-package our application and we have to re-deploy our application.

Note: If any changes required in config properties then We have to repeat the complete project build & deployment which is time consuming process.

=> To avoid this problem, we have to separate our project source code and project config properties files.

=> To externalize config properties from the application we can use Spring Cloud Config Server.

=> Cloud Config Server is part of Spring Cloud Library.

Note: Application config properties files we will maintain in git hub repo and config server will load them and will give to our application based on our application-name.

=> Our microservices will get config properties from Config server and config server will load them from git hub repo.

=====

Developing Config Server App

=====

1) Create Git Repository and keep ymls files required for projects

Note: We should keep file name as application name

app name : greet then file name : greet.yml

app name : welcome then file name : welcome.yml

### Git Repo : [https://github.com/ashokitschool/configuration\\_properties](https://github.com/ashokitschool/configuration_properties)

2) Create Spring Starter application with below dependency

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

3) Write @EnableConfigServer annotation at boot start class

```
@SpringBootApplication
@EnableConfigServer
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

}
```

4) Configure below properties in application.yml file

```
server:
  port: 9090

spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/ashokitschool/configuration_properties
          clone-on-start: true
management:
  security:
    enabled: false
```

5) Run Config Server application

```
=====
Config Server Client Development
=====
```

1) Create Spring Boot application with below dependencies

- a) web-starter
- b) config-client
- c) dev-tools

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

2) Create Rest Controller with Required methods

```

@RestController
@RefreshScope
public class WelcomeRestController {

    @Value("${msg}")
    private String msg;

    @GetMapping("/")
    public String getWelcomeMsg() {
        return msg;
    }
}

```

3) Configure ConfigServer url in application.yml file like below

```

server:
  port: 9091
spring:
  config:
    import: optional:configserver:http://localhost:7071
  application:
    name: welcome

```

4) Run the application and test it.

5) Change app-name to 'welcome' and test it.

```

=====
Circuit Breaker
=====

```

=> It is one of the most famous design pattern in microservices.

=> It is used to implement fault tolerant systems.

=> Fault Tolerant systems also called as Resilience systems

Note: If main logic is failed to execute then we have to execute fallback logic.

=> In springboot, we can implement circuit breaker in 2 ways

- 1) hystrix (outdated)
- 2) Resilience4J (trending)

=> Circuit Breaker works based on 3 states

- 1) CLOSED
- 2) OPEN
- 3) HALF\_OPEN

```

=====
Circuit Breaker Implementation
=====

```

#### 1) Create Spring Boot project with below dependencies

- a) web-starter
- b) actuator
- c) aop
- d) resilience4J

```

<dependency>
  <groupId>io.github.resilience4j</groupId>
  <artifactId>resilience4j-spring-boot3</artifactId>
  <version>2.0.2</version>
</dependency>

```

#### #### 2) Create Rest Controller

```

@RestController
public class DataRestController {

    @GetMapping("/data")
    @CircuitBreaker(fallbackMethod = "getDataFromDB", name = "ashokit")
    public String getData() {
        System.out.println("redis method called..");

        int i = 10 / 0;

        return "Redis Data sent to u r email";
    }

    public String getDataFromDB(Throwable t) {
        System.out.println("db method called..");
        return "DB Data sent to u r email";
    }
}

```

#### #### 3) Configure Circuit Breaker Properties

```

spring:
  application.name: resilience4j-demo

management:
  endpoints.web.exposure.include:
    - '*'
  endpoint.health.show-details: always
  health.circuitbreakers.enabled: true

resilience4j.circuitbreaker:
  configs:
    default:
      registerHealthIndicator: true
      slidingWindowSize: 10
      minimumNumberOfCalls: 5
      permittedNumberOfCallsInHalfOpenState: 3
      automaticTransitionFromOpenToHalfOpenEnabled: true
      waitDurationInOpenState: 100s
      failureRateThreshold: 50
      eventConsumerBufferSize: 10

```

#### #### 4) Test The application and monitor actuator health endpoint

```

=====
Apache Kafka
=====

```

=> Kafka is a Message broker

=> Kafka is used as Streaming platform

=> Kafka is used for realtime data processing

=> Kafka works based on pub & sub model

publisher : App which is producing msgs

subscriber : App which is consuming msgs

=====  
Kafka Architecture  
=====

1) Zookeeper

2) Kafka Server

3) Kafka Topic

4) Publisher App

5) Subscriber (Consumer/Listener)

=====  
Apache Kafka Setup In Windows  
=====

## Step-1 : Download Zookeeper from below URL

URL : <http://mirrors.estointernet.in/apache/zookeeper/>

## Step-2 : Download Apache Kafka from below URL

URL : <http://mirrors.estointernet.in/apache/kafka/>

## Step-3 : Set Path to ZOOKEEPER in Environment variables upto bin folder

### Note: Copy zookeeper.properties and server.properties files from kafka/config folder to kafka/bin/windows folder. ###

## Step-4 : Start Zookeeper server using below command from kafka/bin/windows folder

Command : zookeeper-server-start.bat zookeeper.properties

## Step-5: Start Kafka Server using below command from kafka/bin/windows folder

Command : kafka-server-start.bat server.properties

Note: If kafka server is getting stopped, delete kafka logs from c:/tmp/ folder.

## Step-6 : Create Kakfa Topic using below command from kafka/bin/windows folder

Command : kafka-topics.bat --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic ashokit\_topic

## Step-7 : View created Topics using below command

Command : kafka-topics.bat --list --bootstrap-server localhost:9092

#####  
Kafka Producer App Development  
#####

=====  
1) Create Spring Boot application with below dependencies

```

=====
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-streams</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.kafka</groupId>
        <artifactId>spring-kafka</artifactId>
    </dependency>

    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.kafka</groupId>
        <artifactId>spring-kafka-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>

```

## 2) Create Kafka Constants class

```

public class AppConstants {

    public static final String TOPIC = "ashokit_order_topic";
    public static final String HOST = "localhost:9092";

}

```

## 3) Create Model class to represent data

```

@Data
public class Order {

    private String id;
    private Double price;
    private String email;

}

```

## 4) Create Kafka Producer Config class

```

@Configuration
public class KafkaProduceConfig {

```

```

@Bean
public ProducerFactory<String, Order> producerFactory() {

    Map<String, Object> configProps = new HashMap<>();

    configProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, AppConstants.HOST);
    configProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
    configProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, JsonSerializer.class);

    return new DefaultKafkaProducerFactory<>(configProps);
}

@Bean
public KafkaTemplate<String, Order> kafkaTemplate() {
    return new KafkaTemplate<>(producerFactory());
}
}

```

```

=====
4) Create Service Class
=====

```

```

@Service
public class OrderService {

    @Autowired
    private KafkaTemplate<String, Order> kafkaTemplate;

    public String addMsg(Order order) {

        // publish msg to kafka topic
        kafkaTemplate.send(AppConstants.TOPIC, order);

        return "Msg Published To Kafka Topic";
    }
}

```

```

=====
5) Create RestController classs
=====

```

```

@RestController
public class OrderRestController {

    @Autowired
    private OrderService service;

    @PostMapping("/order")
    public String createOrder(@RequestBody Order order) {
        String msg = service.addMsg(order);
        return msg;
    }
}

```

```

#####
Kafka Subscriber App Development
#####

```

```

=====
1) Develop spring boot app with below dependencies

```

```

=====
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-streams</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.kafka</groupId>
        <artifactId>spring-kafka</artifactId>
    </dependency>

    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.kafka</groupId>
        <artifactId>spring-kafka-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>

```

## 2) Create Constants class

```

public class KafkaConstants {

    public static final String TOPIC = "ashokit_order_topic";
    public static final String HOST = "localhost:9092";

}

```

## 3) Create Model class

```

@Data
public class Order {

    private String id;
    private Double price;
    private String email;

}

```

## 4) Create Consumer Config

```

@Configuration

```



```

public class KafkaConsumerConfig {

    @Bean
    public ConsumerFactory<String, Order> consumerFactory() {

        Map<String, Object> configProps = new HashMap<String, Object>();

        configProps.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, AppConstants.HOST);
        configProps.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        configProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
JsonDeserializer.class);

        return new DefaultKafkaConsumerFactory<>(configProps, new StringDeserializer(), new
JsonDeserializer<>());

    }

    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Order> kafkaListnerFactory() {

        ConcurrentKafkaListenerContainerFactory<String, Order> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        factory.setConsumerFactory(consumerFactory());

        return factory;

    }

}

```

```

=====
5) Add below method in boot app start class
=====

```

```

@KafkaListener(topics = AppConstants.TOPIC, groupId="group_ashokit_order")
public void subscribeMsg(String order) {
    System.out.print("*** Msg Recieved From Kafka *** :: ");
    System.out.println(order);

    //logic
}

```

```

=====
6) Run the producer application & Consumer application
=====

```

##### Send Request to Producer app and observe Subscriber app console #####

```

{
    "id" : "OD101",
    "price" : 200.00,
    "email" : "smith@gmail.com"
}

```

```

=====
Git Hub Repo
=====

```

Producer App : [https://github.com/ashokitschool/spring\\_boot\\_kafka\\_producer.git](https://github.com/ashokitschool/spring_boot_kafka_producer.git)

Subscriber App: [https://github.com/ashokitschool/spring\\_boot\\_kafka\\_consumer.git](https://github.com/ashokitschool/spring_boot_kafka_consumer.git)

```

=====

```

## Redis Cache

=====

=> 2 types of tables

1) Transactional tables : CRUD operations

2) Non-Transactional Tables : SELECT

Git Hub repo : [https://github.com/ashokitschool/SpringBoot\\_Redis\\_Cloud\\_DB\\_App.git](https://github.com/ashokitschool/SpringBoot_Redis_Cloud_DB_App.git)