```
################
Microservices
################


==============================
What is Monolith Architecture
==============================

-> Developing all functionalities in single application is called Monolithic Architecture.

                        - products
                        - cart
                        - checkout
                        - payment
                        - orders
                        - tracking
                        - cancellation
                        - reports
                        - admin

-> We will package our application as jar/war => fat jar or fat war

-----------
Advantages
-----------

-> Simple to develop

-> Everything is available at once place

-> Configurations required only once

---------------
Dis-Advantages
---------------

-> Difficult to maintain

-> Single Point of failure

-> Re-Deploy entire application

####### To overcome the problems of monolithic architecture, Microservices architecture came into
picture... #######

-> Microservices is not a programmaing language

-> Microservices is not a framework

-> Microservices is not an API

-> Microservices is an Architectural Design Pattern

-> Microservices architecture is used to develop application functionalities with loosely coupling.

-> Instead of developing all functionalities in single project, we will divide functionalities and we
will develop multiple apis.

        Note: Every API is called as REST API / Backend API / Service/ Microservice.

-> Microservices are independently deployable components.
```

Note: Microservices architecture is universal. Any programming language project also can be developed using this architecture.

Advantages
---

1) Loosely coupling

2) Easy To Maintain

3) Faster Development

4) Quick Release

5) Technology independence

6) Less Downtime

Challenges with Microservices
---

1) Bounded Context (deciding no.of services we need to develop in the application)

2) Lot of configurations

3) Visibility

4) Infrastructure cost

==========
Microservices Architecture
==========

-> There is no standard / fixed architecture for microservices development.

-> People are customizing microservices architecture based on their requirement.

-> Below are the Microservices architecture components

1) Service Registry / Service Discovery

2) Admin Server

3) Zipkin Server

4) Backend apis / rest api / services

5) FeignClient

6) API Gateway

==========
Service Registry
==========

-> Service Registry is used to maintain list of services available in the project.

-> It provides information about registered services like

Name of service, url of service, status of service

-> It provides no.of instances available for each service.

-> We can use Eureka Server as a service registry

-> Eureka server provided by Spring Cloud Netflix library

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

-> Actuators are used to monitor and manage our applications

-> Monitoring and managing all the apis seperatley is a challenging task

-> Admin Server Provides an user interface to monitor and manage all the apis at one place using actuator endpoints.

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

-> It is Used for Distributed tracing

-> Using zipkin server, we can monitor which api is taking more time to process request.

-> Using Zipkin we can understand how many apis involved in request processing.

==========
Backend apis
==========

-> Backend apis contains business logic

-> Backend apis are also called as REST APIs / services / microservices

Ex: payment-api, cart-api, flights-api, hotels-api

Note: Backend api can register as client for Service Registry, Admin server & Zipkin server (It is optional)

==========
FeignClient
==========

-> It is provided by spring cloud libraries

-> It is used for Inter Service Communication

-> Inter service communication means one api is accessing another api using Service Registry.

Note: External communication means accessing third party apis.

-> When we are using FeignClient we no need mention URL of the api to access. Using service name feign client will get service URL from service registry.

-> Feign Client uses Ribbon to perform Client side load balancing.

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

-> API Gateway is used to manage our project backend apis

-> API Gateway acts as mediator between user requests and backend apis

-> API Gateway acts as entrypoint for all backend apis

-> In API Gateway we will have 2 types of logics

1) Request Filter : To validate the request (go / no-go)

2) Request Router : forward request to particular backend-api based on URL Pattern

/hotels => hotels - api

/flights => flights - api

/trains => trains - api

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

-> Config Server is part of Spring Cloud Library

-> Config Server is used to externalize config properties of application

Note: In realtime we will keep app config properties outside of the project to simpliy application maintanence.

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

-> Kafka is a message broker

-> Kafka works based on Publisher - Subscriber model

-> To send msgs from one app to another app we will use Kafka as a mediator.

-> Using Kafka we can develop Event Driven Microservices based applications.

==========
Redis Cache
==========

-> In our application we will have 2 types of tables

1) Transaction tables (app will insert/update/delete records)

2) Non-Transactional tables (app will only retrieve records)

Note: It is not recommended to load non-transactional tables data from db everytime.

-> To reduce no.of round trips between Java app and Database we will use cache.

-> Redis is used for distributed cache implementation.

==========
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 start class

  3) Create RestController with required method

  4) Configure below properties in application.yml file

  ----------------------application.yml-----------------------------------------
  server:
    port: 8081

  spring:
    application:
      name: WELCOME-API

    boot:
      admin:
        client:
          url: http://localhost:1111/
  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
                - open-feign
```

2) Configure @EnableDiscoveryClient & @EnableFeignClients annotation at start class

3) Create Feign Client to access WELCOME-API

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

        @GetMapping("/welcome")
        public String invokeWelcomeApi();

}
```

4) Create RestController with required methods and inject feign-client to access welcome-api

5) Configure below properties in application.yml file

```
----------------------application.yml----------------------------------------
server:
  port: 8081

spring:
  application:
    name: GREET-API

  boot:
    admin:
      client:
        url: http://localhost:1111/
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka

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

--------------------------------------------------------------------------------
```

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

7) 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)

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

1) Create Spring boot application with below dependencies

```
                -> web-stater
                -> 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---------------------------------
spring:
  cloud:
    gateway:
      discovery.locator:
        enabled: true
        lowerCaseServiceId: true
      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
server:
  port: 3333
-------------------------------------------------------------------------------------

In API gateway we will have 3 types of logics

1) Route

2) Predicate

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.

-> 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 MyPreFilter implements GlobalFilter {

        private Logger logger = LoggerFactory.getLogger(MyPreFilter.class);

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

                logger.info("MyPreFilter :: filter () method executed...");

                // Accessing HTTP Request information
                ServerHttpRequest request = exchange.getRequest();

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

```
                keySet.forEach(key -> {
                        List<String> values = headers.get(key);
                        System.out.println(key +" :: "+values);
                });

                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, moniroing 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 seperate our project code and project confg properties files.

=> To externalize config properties from the application we can use 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


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

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: 9090
spring:
  config:
    import: optional:configserver:http://localhost:8080
  application:
    name: greet
```

4) Run the application and test it.

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

==========

==========
Redis Cache
==========

=> Cache : It is a memory to store data in key-value format

=> Cache is used to reduce no.of db calls in our application

=> DB call is a costly operation in terms of execution time and no.of DB calls reduces application performance.

=> To reduce DB calls in our application we will use Cache.

=> Cache is used to improve performance of the application.

==========
What is Redis ?
==========

The open source, in-memory data store used by millions of developers as a database, cache, streaming engine, and message broker.

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

=> Apache Kafka is a distributed streaming platform

=> Apache Kafka is called as Message Broker

=> Apache Kafka is used to process real time data feeds with high throughput and low latency

        Ex : flights data, sensors data, stocks data, news data, social media etc....

=> Kafka works based on Publisher and Subscriber model

```
==================
Kafka Terminology
==================
Zookeeper
Kafka Server
Kafka Topic
Message
Publisher
Subscriber

===========
Kafka APIs
===========
Connector API
Publisher API
Subscriber API
Streams API


======================================
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

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 ccd_topic

Step-7 : View created Topics using below command

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



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


============================
1) Add 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) Add 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 application
=========================

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


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


================================
Circuit Breaker Design Pattern
================================


=> Circuit Breaker => It is an electic concept

=> It is used to protect us from high voltage or low voltage power

=> It is used to divert traffic when some problem detected in normal execution flow.

=> We can use Circuit Break concept in our microservices to implement fault tolerence systems /
Resillence systems.
```

```
Note: When main logic is failing continuosly then we have to execute fallback logic for sometime.

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

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

		a) web-starter
		b) actuator
		c) aop
		d) resillence4J

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

#### 2) Create Model class to represent Bored API response

	@Data
	public class Activity {

		private String activity;
		private String type;
		private String link;
		private String key;
		private Integer participants;
		private Double price;
		private Double accessibility;

	}

#### 3) Create Rest Controller to consume Bored API

@RestController
public class ActivityRestController {

	private final String BORED_API = "https://www.boredapi.com/api/activity";

	@GetMapping
	@CircuitBreaker(name = "randomActivity", fallbackMethod = "fallbackRandomActivity")
	public String getRandomActivity() {

		RestTemplate rt = new RestTemplate();

		ResponseEntity<Activity> responseEntity = rt.getForEntity(BORED_API, Activity.class);
		Activity activity = responseEntity.getBody();
		System.out.println("Activity Recieved::" + activity.getActivity());

		int i = 10/0;

		return activity.getActivity();
	}

	public String fallbackRandomActivity(Throwable throwable) {
		return "Watch a video from Ashok IT...!!";
	}

}


#### 4) 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: 5s
      failureRateThreshold: 50
      eventConsumerBufferSize: 10


#### 5) Test The application and monitor actuator health endpoint
```