========================
Monolith Architecture
========================

=> Develop all functionalities in single app

=> Application will be packaged as one fat jar / fat war

=> App will be deployed in single server

==========
Drawbacks
==========

1) Single Point of failure
2) Re-Deploy entire app
3) Maintenence of the app
4) Burden on server

===============
Microservices
===============

=> Microservices is a not a technology

=> Microservices is not a framework

=> Microservices is not an API

=> Microservices is an architectural design pattern

=> Microservices design pattern is universal

=> The main aim of microservices is to develop app with loosely coupling

=> Microservices based application means collection of rest apis.

=> Microservices means independetly deployable and executable services.

=========
Benefits
=========

1) Loosely Coupled
2) Easy Maintenence`
3) Load will be distributed
4) Technology Independency
5) High Availability

===========
Challenges
===========

1) Bounded Context (deciding no.of rest apis to develop)
2) Duplicate Configuration
3) Visibility

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

-> There is no standard architecture for Microservices development

-> People are customizing microservices project architecture according to their requirement.

1) Service Registry
2) Admin Server
3) Zipkin Server
4) Backend Services (REST APIs)
5) API Gateway
6) Feign Client
7) Config Server
8) Apache Kafka
9) Redis Cache
10) Docker

==========
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 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 yml 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.

==========
What is Auto Scaling ?
==========

=> It is used to scale up or scale down servers to run our application based on incoming traffic.

1) Fault Tolerance

2) High Availability

3) Cost Management

##########
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 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 config 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

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:9090
  application:
```

```
    name: welcome
```

4) Run the application and test it.

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

==========

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

1) Transactional Tables (app will perform DML operations)

2) Non-Transactional tables (app will perform DQL operations)

=> When table is static there is no use of retrieving data from that table again and again.

=> For static tables data we should use Cache.

What is Cache : temporary storage

=> Get static table data only once and store it in a variable and re-use that variable for future requests.

---------- countries data example ----------

```
@Controller
public class UserController {

      private List<String> countries = null;

      @GetMapping("/register")
      public String loadRegisterPage(Model model){
                  if(countries == null)
                        countries = service.getCountries();
                  }
                  model.addAttribute("countries", countries);
                  return "regiterPage";
      }
}
```

=> The advantage with above logic is countries we will fetch only once from database.

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

=> It is an open source data store

=> We can use Redis as

a) database
b) cache
c) message broker

=> Redis supporting for 50+ programming languages

=> We can setup Redis in 2 ways

1) On Prem Setup (Windows / Linux)

2) Redis Cloud

==========
Spring Boot with Redis Cloud DB
==========

Git Repo URL : https://github.com/ashokitschool/SpringBoot_Redis_Cloud_DB_App.git

1) Setup Redis Cloud Database

2) Create Spring boot app with below dependencies

a) web-starter
b) devtools
c) data-redis
d) redis.client

3) Configure redis db server details in application.properties file

4) Create RedisConfig class to build JedisConnectionFactory

5) Create Binding class for data representation

6) Create Repository for crud operations (CrudRepository)

7) Create Rest Controller with required methods

8) Run the app and test it using postman.

==========
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 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: 5s
      failureRateThreshold: 50
      eventConsumerBufferSize: 10
```

4) Test The application and monitor actuator health endpoint