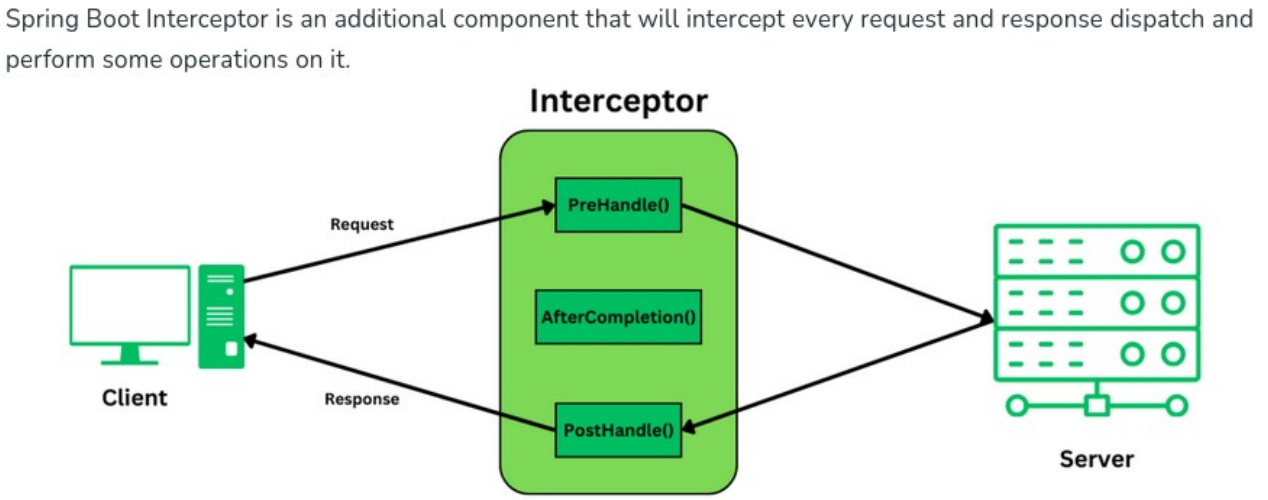
What is use of interceptor in Spring?

Spring Interceptor are used to intercept client requests and process them. Sometimes we want to intercept the HTTP Request and do some processing before handing it over to the controller handler methods.



Implementation of Interceptor

```
@Configuration
public class WebConfig implements WebMvcConfigurer {
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new RequestInterceptor());
    }
}
```

```
@Component
public class RequestInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request,
                             HttpServletResponse response, Object object) {
        return true;
    }
    @Override
    public void postHandle(HttpServletRequest request,
                           HttpServletResponse response, Object object, ModelAndView model){
    }
    @Override
    public void afterCompletion(HttpServletRequest request,
                                HttpServletResponse response, Object object, Exception exception){
    }
}
```

What are conditional beans?

A Spring application context contains an object graph that makes up all the beans that our application needs at runtime. Spring's **@Conditional annotation** allows us to define conditions under which a certain bean is included into that object graph.

ConditionalOnBean :-- Conditional that only matches when the given Bean name or

classes are already present in the BeanFactory.

```
@Configuration
public class ConditionalOnBeanConfig {

    @Bean
    public A beanA(){
        return new A();
    }

    @Bean
    @ConditionalOnBean(name="beanA")
    public B beanB(){
        return new B(); // it will initialize as beanA is present
                        in the beanFactory.
    }
}
```

ConditionalOnResource :-- Conditional that only matches when the specified resource is available on the classpath.

```
@Configuration
public class ConditionalOnResourceConfig {

    @Bean
    @ConditionalOnResource(resources={"classpath:application.properties"})
    public A beanA(){
        return new A(); // will initiate as application.properties is in
                        // classpath.
    }

}
```

ConditionalOnProperty :-- Conditional on property checks if the specified property has the specific value. It checks by default if the property is in the Environment and not equals to false.

```
@Configuration
public class ConditionalBeanPropertyConfig {

    @Bean
    @ConditionalOnProperty(name="app.feature.new", havingValue="true")
    public A beanA(){
        return new A();
    }

    @Bean
    @ConditionalOnProperty(name="app.feature.new", havingValue="false")
    public B beanB(){
        return new B();
    }
}
```

application.properties

```
app.feature.new=true
```

What is difference between @controller and @RestController?

@Controller is used to mark classes as **Spring MVC Controller.** @RestController annotation is a special controller used **in RESTful Web services, and it's the combination of @Controller and @ResponseBody annotation.** It is a specialized version of @Component annotation.

What is Spring boot profile?

Spring Boot profiles, A profile is a set of configuration settings. Spring Boot allows to define profile specific property files in the form of application-{profile}. Properties . It automatically loads the properties in an application.

application.properties **application-local.properties**

```
spring.profiles.default=local
```

Can we activate multiple profiles in spring boot?

Profiles are a core feature of the framework — allowing us to map our beans to different profiles — for example, **dev, test, and prod**. We can then activate different profiles in different environments to bootstrap only the beans we need.

What is the difference between @component ,@Repository and @service?

@Component is a generic stereotype for any Spring-managed component. @Service annotates classes at **the service layer. @Repository annotates classes at the persistence layer**, which will act as a database repository.

How do you add filters to a Spring boot application?
There are three ways to add your filter,

Annotate your filter with one of the Spring stereotypes such as @Component.
Register a @Bean with Filter type in Spring @Configuration.
Register a @Bean with FilterRegistrationBean type in Spring @Configuration.

What is difference between filter and interceptor?

Interceptors share a common API for the server and the client side. Whereas **filters are primarily intended to manipulate request and response parameters like HTTP headers, URIs and/or HTTP methods**, interceptors are intended to manipulate entities, via manipulating entity input/output streams.

What is the use of @SpringBootApplication annotation?

Spring Boot @SpringBootApplication annotation is used to mark a configuration class that declares one or more @Bean methods and also triggers auto-configuration and component scanning. **It's same as declaring a class with @Configuration, @EnableAutoConfiguration and @ComponentScan annotations.**

what are bean scopes in spring?

The scope of a bean defines the life cycle and visibility of that bean in the contexts we use it.

singleton	(Default) Scopes a single bean definition to a single object instance for each Spring IoC container. The container creates a single instance of that bean; all requests for that bean name will return the same object, which is cached. Any modifications to the object will be reflected in all references to the bean. Scopes a single bean definition to any number of object instances.
prototype	A bean with the *prototype* scope will return a different instance every time it is requested from the container. It is defined by setting the value *prototype* to the *@Scope* annotation in the bean definition
request	Scopes a single bean definition to the lifecycle of a single HTTP request. That is, each HTTP request has its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring ApplicationContext.
session	Scopes a single bean definition to the lifecycle of an HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext.
application	Scopes a single bean definition to the lifecycle of a ServletContext. Only valid in the context of a web-aware Spring ApplicationContext.
websocket	Scopes a single bean definition to the lifecycle of a WebSocket. Only valid in the context of a web-aware Spring ApplicationContext.

What is AOP?

AOP breaks the program logic into distinct parts (called concerns). It is used to increase modularity by **cross-cutting concerns.**
A cross-cutting concern is a concern that can affect the whole application and should be centralized in one location in code as possible, such as transaction management, authentication, logging, security etc.

We can use below annotation for AOP

@Aspect :-- This annotation is used to enable AOP

@Pointcut :-- This annotation is used to create cross-cutting concern for spring boot components **(Controller,RestController,service,Repository).**

Example :--
```
@Pointcut("within(@org.springframework.web.bind.annotation.RestController *) || "
+ " within(@org.springframework.stereotype.Service *) || "
+ " within(@org.springframework.stereotype.Repository *)")
private void appPointCut() {
}
```

@Around :-- This annotation is used call created cross-cutting pointCut.

Example :--
```
@Around("appPointCut()")
public Object applicationLogger(ProceedingJoinPoint proceedingJoinPoint) throws Throwable
{
}
```

@AfterThrowing :-- This annotation is used to catch exceptions , which occurs in **(RestController,service,Repository).**

Example :--
```
@AfterThrowing(pointcut = "within(@org.springframework.web.bind.annotation.RestController
*) || "
+ " within(@org.springframework.stereotype.Service *) || "
+ " within(@org.springframework.stereotype.Repository *)", throwing = "exception")
public void logError(JoinPoint joinPoint, Exception exception) throws
JsonProcessingException {
}
```

What is filter ?

A filter dynamically intercepts requests and responses to transform or use the information contained in the requests or responses. The filter API is defined by the Filter. **FilterChain, and FilterConfig** are interfaces in the javax.servlet package.

You define a filter by implementing the Filter interface. A filter chain, passed to a filter by the container, provides a mechanism for invoking a series of filters. A filter config contains initialization data. The most important method in the Filter interface is the **doFilter method**, which **is the heart of the filter.**

Filters can perform many different types of functions.

- Authentication-Blocking requests based on user identity.
- Logging and auditing-Tracking users of a web application.
- Image conversion-Scaling maps, and so on.
- Data compression-Making downloads smaller.
- Localization-Targeting the request and response to a particular locale.

What is Spring IoC (Inversion of Control) Container?

Spring IoC (Inversion of Control) Container is the core of Spring Framework. It creates the objects, configures and assembles their dependencies, manages their entire life cycle. The Container uses **Dependency Injection(DI)** to manage the components that make up the application.

What is Spring
Spring is dependency Injection framework to make java application loosely coupled.Spring provides (IOC) Inversion of Control and using (IOC) we can do dependency Injection.

dependency Injection is design pattern and Spring dynamically creates the object.This process is called (IOC) Inversion of Control, Spring will create the object of class and will inject to in another class is called (IOC) Inversion of Control.

What is Bean life cycle in Java Spring?

A bean is created, used, and finally destroyed when its purpose is over. These are the different stages of a spring life cycle.
The lifecycle of a bean is managed by the Spring container (BeanFactory) which is responsible for **creating, configuring, and destroying beans** according to the defined bean scope (singleton, prototype, etc.).

There are two Bean Lifecycle Callback Methods
Post-initialization callback methods :-- **@PostConstruct** annotation
Pre-destruction callback methods :-- **@PreDestroy** annotation

What is Dependency Injection?

Through Dependency Injection, **the Spring container "injects" objects into other objects or "dependencies". Dependency injection is used to make a class independen**t of its dependencies or **to create a loosely coupled program.** Dependency injection is useful for **improving the reusability of code.**

what is bean in spring ?
A bean is nothing but an instance of a class.

What are Types of dependency injection ?

There are **three main ways** in which a client can receive injected services:

1. Constructor injection:-- where dependencies are provided through a client's class constructor.
2. Setter injection:-- where the client exposes a setter method which accepts the dependency.
3. Property Injection: In the property injection , the injector supplies the dependency through a public property of the client class.

Note :-- With **Setter Injection**, we **automatically cover the Property Injection.**

what is componentscan in spring ?

@ComponentScan is an annotation used for auto-detecting and registering Spring-managed components.

What is jOOQ is (Java Object Orientated Query).

jOOQ is (Java Object Orientated Query). and This library generates Java classes based on the database tables and lets us create type-safe SQL queries through its fluent API.

JOOQ gives you full control because you can read and write the actual query in your code but with type safety. JPA embraces the OO model and this simply does not match the way a database works in all cases, which could result in unexpected queries such as N+1 because you put the wrong annotation on a field.

jOOQ is (Java Object Orientated Query).
jOOQ is a popular database library written in Java, which models the SQL language as a type-safe, internal DSL.
jOOQ is a type safe way to access your database from java programs.

Note :-- In order to use jOOQ type-safe queries, you need to generate Java classes from your database schema.

using DSL (domain specific language) , we can execute type safe sql

Example :--
```
 UsersRecord usersRecord = dsl.selectFrom(USERS)
                .where(USERS.ID.eq(id))
                .fetchOptional().orElseThrow();
```

We can use the DSLContext bean to execute native SQL queries.

Example :--
```
     Record record =
                dsl.resultQuery("select * from users where id = ?", id)
                   .fetchOptional().orElseThrow();
```

Important :-- We can use the jOOQ Code Generation tool to generate the jOOQ classes from our database schema. But to use the jOOQ Code Generation tool, we need to have an existing database up and running.

Examples

With the jOOQ DSL, SQL looks almost as if it were natively supported by Java.

SQL

```
SELECT TITLE
FROM BOOK
WHERE BOOK.PUBLISHED_IN = 2011
ORDER BY BOOK.TITLE
```

JOOQ DSL

```
create.select(BOOK.TITLE)
      .from(BOOK)
      .where(BOOK.PUBLISHED_IN.eq(2011))
      .orderBy(BOOK.TITLE)
```

jOOQ also supports more complex SQL statements.

SQL

```
SELECT AUTHOR.FIRST_NAME,
AUTHOR.LAST_NAME, COUNT(*)
FROM AUTHOR
JOIN BOOK ON AUTHOR.ID = BOOK.AUTHOR_ID
WHERE BOOK.LANGUAGE = 'DE'
AND BOOK.PUBLISHED > DATE '2008-01-01'
GROUP       BY       AUTHOR.FIRST_NAME,
AUTHOR.LAST_NAME
HAVING COUNT(*) > 5
ORDER  BY  AUTHOR.LAST_NAME  ASC  NULLSAUTHOR.LAST_NAME)
FIRST
LIMIT 2
OFFSET 1
```

JOOQ DSL

```
create.select(AUTHOR.FIRST_NAME,
AUTHOR.LAST_NAME, count())
      .from(AUTHOR)
      .join(BOOK).on(AUTHOR.ID.equal(BOO
K.AUTHOR_ID))
      .where(BOOK.LANGUAGE.eq("DE"))
      .and(BOOK.PUBLISHED.gt(date("2008-
01-01")))
            .groupBy(AUTHOR.FIRST_NAME,
      .having(count().gt(5))
      .orderBy(AUTHOR.LAST_NAME.asc().nu
llsFirst())
      .limit(2)
      .offset(1)
```

how to configure multiple database in spring boot?

Reference Link :-- https://javainfinite.com/spring-boot/configuring-multiple-data-sources-with-spring-boot-with-example/

Set multiple database configuration in application.properties.

```
first.datasource.jdbc-url=jdbc:mysql://localhost:3306/employee
first.datasource.username=username
first.datasource.password=password

second.datasource.jdbc-url=jdbc:mysql://localhost:3306/manager
second.datasource.username=username
second.datasource.password=password

spring.jpa.database=default
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=update
```

Database Configurations for primary database

```
package com.multidb;
import jakarta.persistence.EntityManagerFactory;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.jdbc.DataSourceBuilder;
import org.springframework.boot.orm.jpa.EntityManagerFactoryBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.transaction.PlatformTransactionManager;
import javax.sql.DataSource;

@Configuration
@EnableJpaRepositories(
        entityManagerFactoryRef = "employeeEntityManager",
        transactionManagerRef = "employeeTransactionManager",
        basePackages = {"com.multidb.employee.repository"}
)
public class FirstDatabaseConfiguration {

    @Primary
    @Bean(name = "employeedb")
    @ConfigurationProperties(prefix = "first.datasource")
    public DataSource dataSource() {
        return DataSourceBuilder.create().build();
    }
    @Primary
    @Bean(name = "employeeEntityManager")
  public LocalContainerEntityManagerFactoryBean
  employeeManager(EntityManagerFactoryBuilder builder,
        @Qualifier("employeedb") DataSource dataSource) {
        return builder
                .dataSource(dataSource)
                .packages("com.multidb.employee")
```

```
                .build();
    }
    @Primary
    @Bean(name = "employeeTransactionManager")
    public PlatformTransactionManager primaryTransactionManager(
            @Qualifier("employeeEntityManager")
            EntityManagerFactory employeeEntityManagerFactory) {

        return new JpaTransactionManager(employeeEntityManagerFactory);
    }

}
```

Database Configurations for Secondary database

```
package com.multidb;

import jakarta.persistence.EntityManagerFactory;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.jdbc.DataSourceBuilder;
import org.springframework.boot.orm.jpa.EntityManagerFactoryBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.transaction.PlatformTransactionManager;
import javax.sql.DataSource;

@Configuration
@EnableJpaRepositories(
        entityManagerFactoryRef = "employeeEntityManager",
        transactionManagerRef = "employeeTransactionManager",
        basePackages = {"com.multidb.employee.repository"}
)
public class SecondDatabaseConfiguration {

    @Bean(name = "managerdb")
    @ConfigurationProperties(prefix = "second.datasource")
    public DataSource dataSource() {
        return DataSourceBuilder.create().build();
    }

    @Bean(name = "employeeEntityManager")
    public LocalContainerEntityManagerFactoryBean
    employeeManager(EntityManagerFactoryBuilder builder,
       @Qualifier("managerdb") DataSource dataSource) {
        return builder
                .dataSource(dataSource)
                .packages("com.multidb.manager")
                .build();
    }

    @Bean(name = "managerTransactionManager")
    public PlatformTransactionManager primaryTransactionManager(
            @Qualifier("managerTransactionManager")
            EntityManagerFactory managerEntityManagerFactory) {

        return new JpaTransactionManager(managerEntityManagerFactory);
    }

}
```

Create Entitty

Employee.java (Entity)

```
import jakarta.persistence.*;
import lombok.Data;
import lombok.NoArgsConstructor;

@Entity
@Data
@NoArgsConstructor
@Table(name="employee")
public class Employee {

    @Id
    @GeneratedValue (strategy = GenerationType.AUTO)
    private int id;
    private String eName;
    private String eDept;
}
```

Manager.java (Entity)

```
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import lombok.Data;
import lombok.NoArgsConstructor;

@Entity
@Data
@NoArgsConstructor
public class Manager {

    @Id
    @GeneratedValue (strategy = GenerationType.AUTO)
    private int id;
    private String mName;
    private String mDept;
}
```

Create Repository

EmployeeRepository.java (Repository)

```
import com.multidb.employee.model.Employee;
import
org.springframework.data.jpa.repository.JpaRepository;

public interface EmployeeRepository
extends JpaRepository<Employee, Integer> {

}
```

ManagerRepository.java (Repository)

```
import com.multidb.manager.model.Manager;
import
org.springframework.data.jpa.repository.JpaRepository;

public interface ManagerRepository
extends JpaRepository<Manager, Integer> {

}
```

Create Service

EmployeeService.java (Service)

```
import com.multidb.employee.model.Employee;
import com.multidb.employee.repository.EmployeeRepository;
import
org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Service;

import javax.sql.DataSource;

@Service
public class EmployeeService {

    private EmployeeRepository repository;

    private DataSource dataSource;

    public EmployeeService(EmployeeRepository repository,
    @Qualifier("employeedb") DataSource dataSource) {
        this.repository = repository;
        this.dataSource = dataSource;
    }

    public Employee saveEmployee(Employee employee) {
        return repository.save(employee);
    }
}
```

ManagerService.java (Service)

```
import com.multidb.manager.model.Manager;
import com.multidb.manager.repository.ManagerRepository;
import
org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Service;

import javax.sql.DataSource;

@Service
public class ManagerService {

    private DataSource dataSource;

    private ManagerRepository repository;

    public ManagerService(ManagerRepository repository,
    @Qualifier("managerdb") DataSource dataSource ) {
        this.repository = repository;
        this.dataSource = dataSource;
    }

    public Manager saveManager(Manager manager) {
        return repository.save(manager);
    }
}
```

What is ambiguous exception and what is solution

spring scanning same type bean with different names returning ambiguous exception. **when multiple beans of the same type are present then ambiguity happen** so You need to use the @Qualifier annotation together with @Annotated to resolve ambiguity between different beans with the same type.

SPRING Annotations :-

@Qualifier : - The @Qualifier annotation is used along with **@Autowired annotation.** It provides more control over the dependency injection process. It is useful for reducing the duplicity while creating more bean files of the same type.

@Autowired :- The @Autowired annotation is a great way of making the need to inject a dependency in Spring. **The @Autowired annotation is applied on fields, instance variables, setter methods, and constructors**. It provides auto wiring to bean properties

@Component :- The @Component annotation indicates that a java class is a bean. The @Component annotation is used at the class level to mark a Java class as a bean.

@Controller:- A @Controller / RestController stereotype used at class level in spring MVC! It marks a class as a spring web controller, responsible to handle HTTP request! it is used with @RequestMapping annotation.

@Bean :- A @Bean annotation is used at method level to indicate that the annotated method will **simply return a new instance that should be registered as a spring managed bean.** It is a method level annotation, which tells the method to produce a bean to be managed by the Spring Container.

@Configuration :- @Configuration annotation is used at class level to indicate that the class will be used as a source of bean definitions. @Configuration used to mark a class as a source of the bean definitions. Beans are the components of the system that you want to wire together.
A method marked with the @Bean annotation is a bean producer. Spring will handle the life cycle of the beans for you, and it will use these methods to create the beans.

The @Configuration annotation is used on classes which defines bean as the source of the bean file.

@Valid :-- @Valid annotation is used validated Request object (@RequestBody) using JSR 380 Hibernate specification. Which validate all attribute of RequestBody

@ComponentScan :- @ComponentScan use to make sure that Spring knows about your configuration classes and can initialize the beans correctly. It makes Spring scan the packages configured with it for the @Configuration classes. T**he @ComponentScan annotation is used to scan a package for beans.** It is used with @Configuration annotation. We can also define the base package for scanning

@Scope - used to define the scope of a @Component class or a @Bean definition and can be either singleton, prototype, request, session, globalSession, or custom scope.

@Profile - adds beans to the application only when that profile is active.

@Primary - gives higher preference to a bean when there are multiple beans of the same type.

@SpringBootApplication :- One of the most basic and helpful annotations, is @SpringBootApplication. @SpringBootApplication is @Configuration, @EnableAutoConfiguration and @ComponentScan annotations combined, configured with their default attributes.

@ResponseBody :- - The @ResponseBody is a utility annotation that makes Spring bind a method's return value to the HTTP response body and response body can be JSON or XML format.

@RequestMapping :-- The @RequestMapping annotation is used to map the web requests. It can hold several optional components such as consumes, header, method, name, params, path, value, etc.

@RequestParam :-- The @RequestParam annotation also called query parameter is used to extract the query parameters from the URL.

@PathVariable :-- @PathVariable("placeholderName") annotation to bring the values from the URL to the method arguments.

@Value - used to assign values into fields in Spring-managed beans. It's compatible with the constructor, setter, and field injection.

@Lazy - makes beans to initialize lazily. @Lazy annotation may be used on any class. The bean will be created and initialized when it is requested. When the @Lazy annotation is used on a @Configuration class; it will initialize all the @Bean methods lazily.

@Required :- The @Required annotation is applied on the setter method of the bean file. Use of this annotation indicates that the annotated bean must be populated at configuration time with the required property. Otherwise, it will throw a BeanInitializationException.

@Service :- The @Service annotation is also used at the class level to mark a service implementation including business logic, calculations, call external APIs, etc. Generally, it holds the business logic of the application.

@RestController :- The @RestController annotation is a combination of @Controller and @ResponseBody annotations. It is itself annotated with the @ResponseBody annotation. If the @RestController is used, then there is no need for annotating each method with @ResponseBody.

@CrossOrigin :-- The @CrossOrigin annotation is used both at the class & method level. It is used to enable cross-origin requests. It useful for the cases, if a **host serving JavaScript is different from the data serving host.** In such cases, the CORS (Cross-Origin Resource Sharing) enables cross-domain communication.

SPRING BOOT MICRO SERVICES :-

Microservices Architecture: - In a microservices architecture, the application is divided into small, independently deployable services. Each service focuses on a specific business capability and communicates with other services through APIs.

Microservice has almost 6 **major components,** which are required for building a scalable microservices environment

- Service Discovery & Service Registry
- Load Balancer
- API Gateway
- Circuit Distributed Tracing Breaker
- Service Monitoring Distributed Tracing Breaker (Sleuth)
- Configuration Management Config Server

Service Discovery & Service Registry (Eureka): - Spring Cloud Eureka is a service registry and discovery server. It allows microservices to register themselves and discover other services in the ecosystem. This enables dynamic scaling and routing.

Load Balancer :- Netflix's Ribbon is a load-balancing library that provides client-side load balancing for RESTful applications.
Using Ribbon with Spring Boot microservices can improve service availability, scalability, and fault tolerance.

Note :- Resilience4j is a lightweight fault tolerance library that provides a variety of fault tolerance and stability patterns to a web application.

API Gateway :- Spring Cloud Zuul acts as an API gateway that manages the routing, filtering, and load balancing of requests to different microservices. It helps simplify the client-side communication with multiple services. It acts as gatekeeper to the outside world, not allowing any unauthorized external requests.

Circuit Breaker (Hystrix):- Spring Cloud Hystrix helps prevent system failures due to service outages. It provides a circuit breaker pattern to handle and manage failures gracefully.

Service Monitoring Circuit Distributed Tracing Breaker (Sleuth):- Spring Cloud Sleuth provides distributed tracing to track requests across various microservices. It helps diagnose and troubleshoot performance issues.

Configuration Management (Config Server):- Spring Cloud Config Server manages externalized configuration for microservices. It allows you to centralize configuration properties for different environments and services.