1. **What are the bean scopes in Spring**

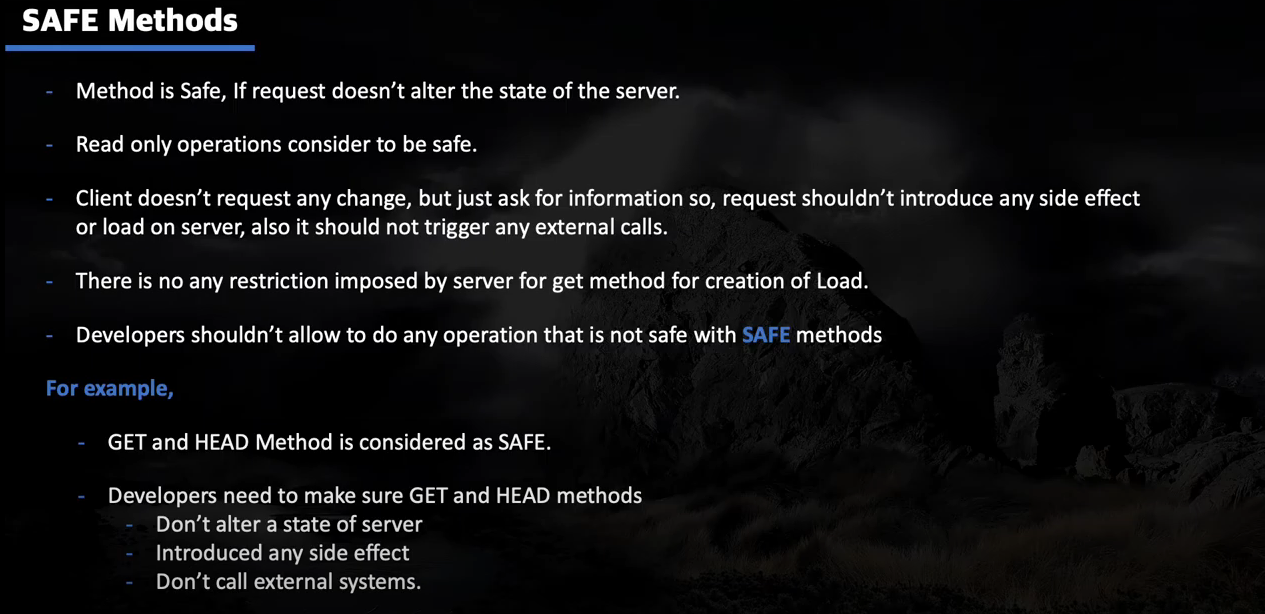
* **Singleton (default)** : This scopes the bean definition to a single instance per Spring IoC container
* **Prototype** : This scopes a single bean definition to have any number of object instances.
* **Request** : This scopes a bean definition to an HTTP request. Only valid in the context of a web-aware Spring ApplicationContext.
* **Session** : This scopes a bean definition to an HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.
* **global-session** : This scopes a bean definition to a global HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.

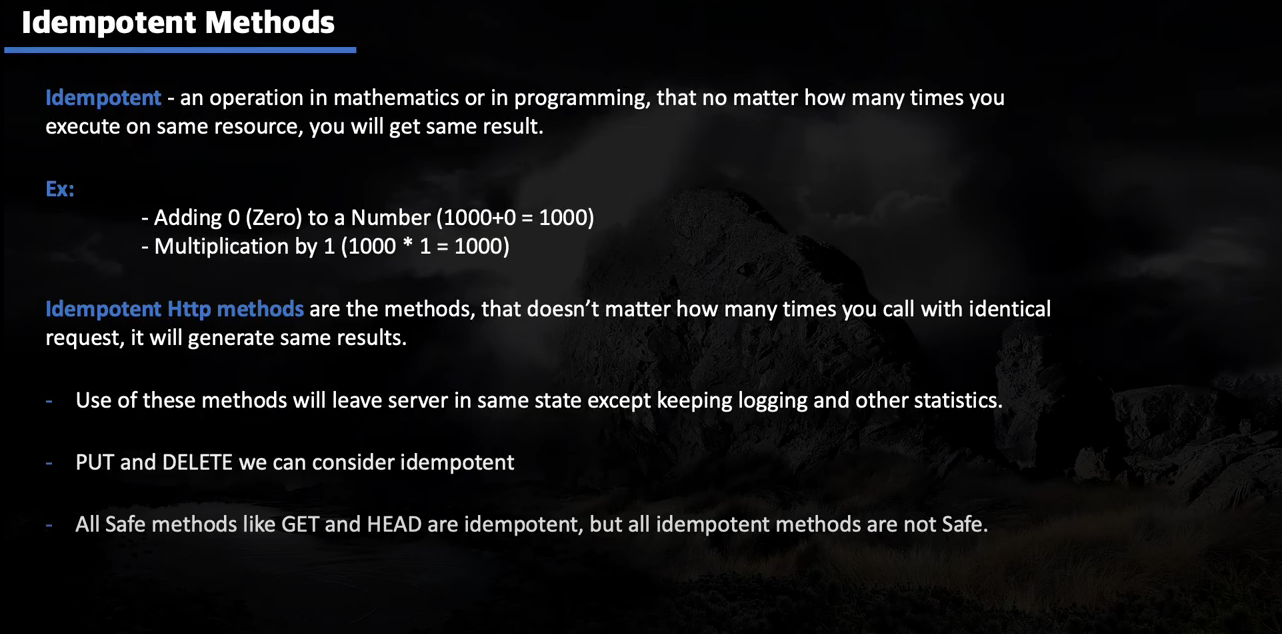
1. **Why stream API introduced in Java 8**

Streams provide the most convenient and natural way to apply functions to sequences of objects. Streams encourage less mutability

1. **HTTP methods:**

* **GET** : to fetch data from DB
* **POST**: to create entry in DB. It is not idempotent method
* **DELETE** : to delete data from DB. It is idempotent method
* **PUT** : update the existing data in DB. It is idempotent method
* **PATCH** : update only data which has sent in request. It is not idempotent method
* **OPTIONS** : to get the access to specific URL or use astric (\*) to get access to entire resource or server. It is idempotent method
* **TRACE** : this method used to echo the content of the HTTP request back to the requester for connection debugging purpose.
* **HEAD** : it is same as get method and faster than get method. But it is used to retrieve the information about document not actual data from document.





1. **HTTP Response codes:**

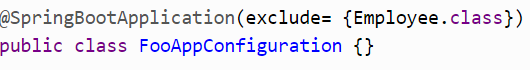
* 200 OK
* 404 Not Found
* 201 created
* 409 conflict if resource is already present
* 405 Method not allowed
* 202 Accepted
* 203 non Authorized information
* 400 Bad request
* 401 Unauthorized
* 403 Forbidden
* 500 internal server error

1. **What does the @SpringBootApplication annotation do internally**

As per the Spring Boot doc, the [@SpringBootApplication](https://docs.spring.io/spring-boot/docs/current/reference/html/using-boot-using-springbootapplication-annotation.html) annotation is equivalent to using [@Configuration](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/context/annotation/Configuration.html), [@EnableAutoConfiguration](https://docs.spring.io/spring-boot/docs/current/reference/html/using-boot-auto-configuration.html), and [@ComponentScan](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/context/annotation/ComponentScan.html) with their default attributes. Spring Boot enables the developer to use a single annotation instead of using multiple. But, as we know, Spring provided loosely coupled features that we can use for each individual annotation as per our project needs.

1. **How to exclude any package without using the basePackages filter?**

There are different ways you can filter any package. But Spring Boot provides a trickier option for achieving this without touching the component scan. You can use the exclude attribute while using the annotation  @SpringBootApplication. See the following code snippet



1. **How to disable a specific auto-configuration class?**

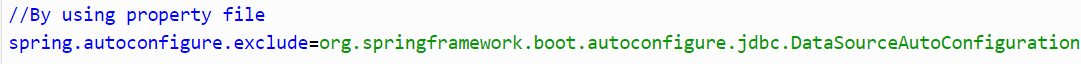
* You can use the exclude attribute of@EnableAutoConfiguration, if you find any specific auto-configuration classes that you do not want are being applied.



* On the other foot, if the class is not on the classpath, you can use the excludeName attribute of the annotation and specify the fully qualified name instead.



* Also, Spring Boot provides the facility to control the list of auto-configuration classes to exclude by using the spring.autoconfigure.exclude property. You can add into the application.properties. And you can add multiple classes with comma separated.



1. **How to enable/disable the Actuator?**

Enabling/disabling the actuator is easy; the simplest way is to enable features to add the dependency (Maven/Gradle) to the spring-boot-starter-actuator, i.e. Starter. If you don't want the actuator to be enabled, then don't add the dependency.

1. **Actuator endpoints**

* Beans: get all beans
* Sessions : retrieve and delete user created sessions
* Shutdown : to shutdown gracefully
* Health : shows application health information
* Conditions : shows condition added on configuration and auto-configuration.

1. **What is the Spring Initializer?**

Spring initializer menace the project creation i.e <https://start.spring.io/>

1. **What is a shutdown in the actuator?**

[Shutdown](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-endpoints.html) is an endpoint that allows the application to be gracefully shutdown. This feature is not enabled by default. You can enable this by using management.endpoint.shutdown.enabled=true in your application.properties file. But be careful about this if you are using this.

1. **Is this possible to change the port of Embedded Tomcat server in Spring boot?**

server.port=8081 in application.property file

1. **Can we disable the default web server in the Spring Boot application**

Yes, we can use the application.properties to configure the web application type, i.e.  spring.main.web-application-type=none.

1. **Can we override or replace the Embedded Tomcat server in Spring Boot?**

Yes, we can replace the Embedded Tomcat with any other servers by using the Starter dependencies. You can use spring-boot-starter-jetty  or spring-boot-starter-undertow as a dependency for each project as you need.

1. [**Spring**](https://www.edureka.co/blog/spring-tutorial/)**vs Spring Boot**

|  |  |
| --- | --- |
| Spring | Spring Boot |
| A web application framework based on Java | A module of Spring |
| Provides tools and libraries to create customized web applications | Used to create a Spring application project which can just run/ execute |
| Spring is more complex than Spring Boot | Spring Boot is less complex than the Spring framework |
| Takes an unopinionated view | Takes an opinionated view of a platform |

## What is Spring Boot and mention the need for it?

Spring Boot is a Spring module which aims to simplify the use of the Spring framework for Java development. It is used to a create stand-alone Spring-based applications which you can just run. So, it basically removes a lot of configurations and dependencies. Aiming at the Rapid Application Development, Spring Boot framework comes with the auto-dependency resolution, embedded HTTP servers, auto-configuration, management endpoints, and [Spring Boot CLI](https://www.edureka.co/blog/spring-boot-setup-helloworld-microservices-example/).

So, if you ask me why should anybody use Spring Boot, then I would say, Spring Boot not only improves productivity but also provides a lot of conveniences to write your own business logic.

## Mention the advantages of Spring Boot

* Provides auto-configuration to load a set of default configuration for a quick start of the application
* Creates stand-alone applications with a range of non-functional features that are common to large classes of projects
* It comes with embedded tomcat
* Spring Boot provides an opinionated view to reduce the developer effort and simplify maven configurations
* Consists of a wide range of APIs for monitoring and managing applications in dev and prod(actuator).

## Mention a few features of Spring Boot

## Starter Dependency

## Spring Initializer

## Auto-Configuration

## Spring Actuator

## Logging and Security

## Mention the possible sources of external configuration

* Application Properties
* Command-line properties
* YAML file
* Profile-specific properties

## Can you explain what happens in the background when a Spring Boot Application is “Run as Java Application”?

## When a Spring Boot application is executed as “Run as Java application”, then it automatically launches up the tomcat server as soon as it sees, that you are developing a web application.

## What are the Spring Boot starters and what are available the starters?

## set of convenient dependency management providers which can be used in the application to enable dependencies. These starters, make development easy and rapid and comes under the org.springframework.boot.

* spring-boot-starter: – This is the core starter and includes logging, auto-configuration support, and YAML.
* spring-boot-starter-jdbc – This starter is used for HikariCP connection pool with JDBC
* spring-boot-starter-web – Is the starter for building web applications, including RESTful, applications using Spring MVC
* spring-boot-starter-data-jpa – Is the starter to use Spring Data JPA with Hibernate
* spring-boot-starter-security – Is the starter used for Spring Security
* spring-boot-starter-aop: This starter is used for aspect-oriented programming with AspectJ and  Spring AOP
* spring-boot-starter-test: Is the starter for testing Spring Boot applications

## Mention the minimum requirements for a Spring boot System

Spring Boot 2.1.7.RELEASE requires

* Java 8 +
* Spring Framework 5.1.9 +

Explicit build support

* Maven 3.3+
* Gradle 4.4+

Servlet Container Support

* Tomcat 9.0 – Servlet Version 4.0
* Jetty 9.4 –  Servlet Version 3.1
* Undertow 2.0 – Servlet Version 4.0

## What are the @RequestMapping  and @RestController annotation in Spring Boot used for?

|  |  |
| --- | --- |
| @RequestMapping | @RestController |
| This annotation is used to provide the routing information and tells to Spring that any HTTP request must be mapped to the respective method. | This annotation is used to add the @ResponseBody and @Controller annotation to the class |
| To use this annotation, you have to import org.springframework.web.  bind.annotation.RequestMapping; | To use this annotation, you have to import org.springframework.web.  bind.annotation.RestController; |

## Mention the differences between JPA and [Hibernate](https://www.edureka.co/blog/what-is-hibernate-in-java/)

|  |  |
| --- | --- |
| JPA | Hibernate |
| JPA is a Data Access Abstraction used to reduce the amount of boilerplate code | Hibernate is an implementation of Java Persistence API and offers benefits of loose coupling |

## What are the differences between @SpringBootApplication and @EnableAutoConfiguration annotation?

|  |  |
| --- | --- |
| @SpringBootApplication | @EnableAutoConfiguration |
| Used in the main class or bootstrap class | Used to enable auto-configuration  and component scanning in your project |
| It is a combination of @Configuration, @ComponentScan and @EnableAutoConfiguration annotations. | It is a combination of @Configuration and @ComponentScan annotations |

## What is the way to use profiles to configure the environment-specific configuration with Spring Boot?

Since it is a known fact that a Profile is nothing but a key to identify an environment lets consider the following two profiles in the example:

* dev
* prod
* Consider the following properties present in the application properties file:

example.number: 100  
example.value: true  
example.message: Dynamic Message

Now, say you want to customize the application.properties for dev profile, then you need to create a file with name application-dev.properties and override the properties that you want to customize. You can mention the following code:

example.message: Dynamic Message in Dev

Similarly, if you want to customize the application.properties for prod profile, then you can mention the following code snippet:

example.message: Dynamic Message in Prod

Once you are done with the profile-specific configuration, you have to set the active profile in an environment. To do that, either you can

* Use -Dspring.profiles.active=prod in  arguments
* Use spring.profiles.active=prod in application.properties file

## What do you understand by Spring Boot supports relaxed binding?

Relaxed binding, is a way in which, the property name does not need to match the key of the environment property. In Spring Boot, relaxed binding is applicable to the type-safe binding of the configuration properties. For example, if a property in a bean class with the @ConfigurationPropertie annotation is used sampleProp, then it can be bounded to any of the following environment properties:

* sampleProp
* sample-Prop
* sample\_Prop
* SAMPLE\_PROP

## Where is the database connection information specified and how does it automatically connect to H2?

## Well, the answer to this question is very simple. It is because of the Spring Boot auto-configuration that, configures the dependencies of the application. So, the database connection information, and automatically connecting the database to H2 is done by the auto-configuration property.

### **What are the features of Spring Boot?**

* Web Development
* SpringApplication
* Application events and listeners
* Admin features

### **@Required**

This annotation is applied on bean setter methods. Consider a scenario where you need to enforce a required property. The @Required annotation indicates that the affected bean must be populated at configuration time with the required property. Otherwise an exception of type BeanInitializationException is thrown.

### **@Autowired**

This annotation is applied on fields, setter methods, and constructors. The @Autowired annotation injects object dependency implicitly.

When you use @Autowired on fields and pass the values for the fields using the property name, Spring will automatically assign the fields with the passed values.

You can even use @Autowired on private properties, as shown below. (This is a very poor practice though!)

1. public class Customer {
2. @Autowired
3. private Person person;
4. private int type;
5. }

When you use @Autowired on setter methods, Spring tries to perform the by Type autowiring on the method. You are instructing Spring that it should initiate this property using setter method where you can add your custom code, like initializing any other property with this property.

1. public class Customer {
2. private Person person;
3. @Autowired
4. public void setPerson (Person person) {
5. this.person=person;
6. }
7. }

Consider a scenario where you need instance of class A, but you do not store A in the field of the class. You just use A to obtain instance of B, and you are storing B in this field. In this case setter method autowiring will better suite you. You will not have class level unused fields.

When you use @Autowired on a constructor, constructor injection happens at the time of object creation. It indicates the constructor to autowire when used as a bean. One thing to note here is that only one constructor of any bean class can carry the @Autowired annotation.

1. @Component
2. public class Customer {
3. private Person person;
4. @Autowired
5. public Customer (Person person) {
6. this.person=person;
7. }
8. }

NOTE: As of Spring 4.3, @Autowired became optional on classes with a single constructor. In the above example, Spring would still inject an instance of the Person class if you omitted the @Autowired annotation.

### **@Qualifier**

This annotation is used along with @Autowired annotation. When you need more control of the dependency injection process, @Qualifier can be used. @Qualifier can be specified on individual constructor arguments or method parameters. This annotation is used to avoid confusion which occurs when you create more than one bean of the same type and want to wire only one of them with a property.

Consider an example where an interface BeanInterface is implemented by two beans BeanB1 and BeanB2.

1. @Component
2. public class BeanB1 implements BeanInterface {
3. //
4. }
5. @Component
6. public class BeanB2 implements BeanInterface {
7. //
8. }

Now if BeanA autowires this interface, Spring will not know which one of the two implementations to inject.  
One solution to this problem is the use of the @Qualifier annotation.

1. @Component
2. public class BeanA {
3. @Autowired
4. @Qualifier("beanB2")
5. private BeanInterface dependency;
6. ...
7. }

With the @Qualifier annotation added, Spring will now know which bean to autowire where beanB2 is the name of BeanB2.

### **@Configuration**

This annotation is used on classes which define beans. @Configuration is an analog for XML configuration file – it is configuration using Java class. Java class annotated with @Configuration is a configuration by itself and will have methods to instantiate and configure the dependencies.

Here is an example:

1. @Configuration
2. public class DataConfig{
3. @Bean
4. public DataSource source(){
5. DataSource source = new OracleDataSource();
6. source.setURL();
7. source.setUser();
8. return source;
9. }
10. @Bean
11. public PlatformTransactionManager manager(){
12. PlatformTransactionManager manager = new BasicDataSourceTransactionManager();
13. manager.setDataSource(source());
14. return manager;
15. }
16. }

### **@ComponentScan**

This annotation is used with @Configuration annotation to allow Spring to know the packages to scan for annotated components. @ComponentScan is also used to specify base packages using basePackageClasses or basePackage attributes to scan. If specific packages are not defined, scanning will occur from the package of the class that declares this annotation.

### **@Bean**

This annotation is used at the method level. @Bean annotation works with @Configuration to create Spring beans. As mentioned earlier, @Configuration will have methods to instantiate and configure dependencies. Such methods will be annotated with @Bean. The method annotated with this annotation works as bean ID and it creates and returns the actual bean.

Here is an example:

1. @Configuration
2. public class AppConfig{
3. @Bean
4. public Person person(){
5. return new Person(address());
6. }
7. @Bean
8. public Address address(){
9. return new Address();
10. }
11. }

### **@Lazy**

This annotation is used on component classes. By default all autowired dependencies are created and configured at startup. But if you want to initialize a bean lazily, you can use @Lazy annotation over the class. This means that the bean will be created and initialized only when it is first requested for. You can also use this annotation on @Configuration classes. This indicates that all @Bean methods within that @Configuration should be lazily initialized.

### **@Value**

This annotation is used at the field, constructor parameter, and method parameter level. The @Value annotation indicates a default value expression for the field or parameter to initialize the property with. As the @Autowired annotation tells Spring to inject object into another when it loads your application context, you can also use @Value annotation to inject values from a property file into a bean’s attribute. It supports both #{...} and ${...} placeholders.

### **@Component**

This annotation is used on classes to indicate a Spring component. The @Component annotation marks the Java class as a bean or say component so that the component-scanning mechanism of Spring can add into the application context.

### **@Controller**

The @Controller annotation is used to indicate the class is a Spring controller. This annotation can be used to identify controllers for Spring MVC or Spring WebFlux.

### **@Service**

This annotation is used on a class. The @Service marks a Java class that performs some service, such as execute business logic, perform calculations and call external APIs. This annotation is a specialized form of the @Component annotation intended to be used in the service layer.

### **@Repository**

This annotation is used on Java classes which directly access the database. The @Repository annotation works as marker for any class that fulfills the role of repository or Data Access Object.

This annotation has a automatic translation feature. For example, when an exception occurs in the @Repository there is a handler for that exception and there is no need to add a try catch block.

## Spring Boot Annotations

### **@EnableAutoConfiguration**

This annotation is usually placed on the main application class. The @EnableAutoConfiguration annotation implicitly defines a base “search package”. This annotation tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings.

### **@SpringBootApplication**

This annotation is used on the application class while setting up a Spring Boot project. The class that is annotated with the @SpringBootApplication must be kept in the base package. The one thing that the @SpringBootApplication does is a component scan. But it will scan only its sub-packages. As an example, if you put the class annotated with @SpringBootApplication in com.example then @SpringBootApplication will scan all its sub-packages, such as com.example.a, com.example.b, and com.example.a.x.

The @SpringBootApplication is a convenient annotation that adds all the following:

* @Configuration
* @EnableAutoConfiguration
* @ComponentScan

## Spring MVC and REST Annotations

### **@RequestMapping**

This annotation is used both at class and method level. The @RequestMapping annotation is used to map web requests onto specific handler classes and handler methods. When @RequestMapping is used on class level it creates a base URI for which the controller will be used. When this annotation is used on methods it will give you the URI on which the handler methods will be executed. From this you can infer that the class level request mapping will remain the same whereas each handler method will have their own request mapping.

Sometimes you may want to perform different operations based on the HTTP method used, even though the request URI may remain the same. In such situations, you can use the method attribute of @RequestMapping with an HTTP method value to narrow down the HTTP methods in order to invoke the methods of your class.

Here is a basic example on how a controller along with request mappings work:

1. @Controller
2. @RequestMapping("/welcome")
3. public class WelcomeController{
4. @RequestMapping(method = RequestMethod.GET)
5. public String welcomeAll(){
6. return "welcome all";
7. }
8. }

In this example only GET requests to /welcome is handled by the welcomeAll() method.

This annotation also can be used with Spring MVC and Spring WebFlux.

### **@CookieValue**

This annotation is used at method parameter level. @CookieValue is used as argument of request mapping method. The HTTP cookie is bound to the @CookieValue parameter for a given cookie name. This annotation is used in the method annotated with @RequestMapping.  
Let us consider that the following cookie value is received with a http request:

JSESSIONID=418AB76CD83EF94U85YD34W

To get the value of the cookie, use @CookieValue like this:

1. @RequestMapping("/cookieValue")
2. public void getCookieValue(@CookieValue "JSESSIONID" String cookie){
3. }

### **@CrossOrigin**

This annotation is used both at class and method level to enable cross origin requests. In many cases the host that serves JavaScript will be different from the host that serves the data. In such a case Cross Origin Resource Sharing (CORS) enables cross-domain communication. To enable this communication you just need to add the @CrossOrigin annotation.

By default the @CrossOrigin annotation allows all origin, all headers, the HTTP methods specified in the @RequestMapping annotation and maxAge of 30 min. You can customize the behavior by specifying the corresponding attribute values.

An example to use @CrossOrigin at both controller and handler method levels is this.

1. @CrossOrigin(maxAge = 3600)
2. @RestController
3. @RequestMapping("/account")
4. public class AccountController {
5. @CrossOrigin(origins = "http://example.com")
6. @RequestMapping("/message")
7. public Message getMessage() {
8. // ...
9. }
10. @RequestMapping("/note")
11. public Note getNote() {
12. // ...
13. }
14. }

In this example, both getExample() and getNote() methods will have a maxAge of 3600 seconds. Also, getExample() will only allow cross-origin requests from http://example.com, while getNote() will allow cross-origin requests from all hosts.

**Composed @RequestMapping Variants**

### **@GetMapping**

### **@PostMapping**

### **@PutMapping**

### **@PatchMapping**

### **@DeleteMapping**

1. **View resolvers**

