**Spring and Spring boot mock questions**

**Diff between Controller and RestController**

1) What is Spring?

It is a lightweight, loosely coupled and integrated framework for developing enterprise applications in java.

2) What are the advantages of spring framework?

1. Predefined Templates
2. Loose Coupling
3. Easy to test
4. Lightweight
5. Fast Development
6. Powerful Abstraction
7. Declarative support

**Q) Who creates and manages entire life cycle of objects?**

**🡪** spring IOC

### 4) What is IOC and DI?

### 🡪 Spring IOC (Inversion of control) is the core of spring framework. It creates the object, configures and assembles their dependencies, manage their entire life cycle.

### The container uses DI to manage the components that make up the application.

### Dependencies are nothing but predefined classes and interfaces.

### DI(Dependency injection) is a concept in which objects get other objects from outside.

IOC (Inversion of Control) and DI (Dependency Injection) is a design pattern to provide loose coupling. It removes the dependency from the program.

**public** **class** Employee{

Address address;

Employee(){

address=**new** Address();//creating instance

}

}

Now, there is dependency between Employee and Address because Employee is forced to use the same address instance.

Let's write the IOC or DI code.

**public** **class** Employee{

Address address;

Employee(Address address){

**this**.address=address;//not creating instance

}

}

Now, there is no dependency between Employee and Address because Employee is not forced to use the same address instance. It can use any address instance.

Q) Difference between Spring and Spring Boot.

|  |  |  |
| --- | --- | --- |
| **No.** | **Spring** | **Spring Boot** |
| 1) | Spring is an open-source, lightweight framework widely used to develop enterprise applications. | Spring boot is built on top of the conventional spring framework widely used to develop REST APIs. |
| 2) | Spring framework’s most important feature is dependency injection. | Spring boot’s most important feature is auto-configuration. |
| 3) | It helps to create loosely coupled applications. | It helps to create stand-alone applications. |
| 4) | To run an application, we need to set the server explicitly. | Spring boot provides embedded servers like Tomcat, Jetty, Undertow. |
| 5) | To run spring application, a deployment description is required | There is no requirement for deployment description. |
| 6) | To create spring application, developers write lot of code. | Spring boot reduces lines of code |
| 7) | It doesn’t support for the in-memory database | It provides support for the in-memory database such ||}s H2 |

Q) Explain SDLC(software development life cycle)

🡪 There should be clear understanding among team representative about when and what to do.

Systematic and proper planning should be there.

A software lifecycle model describes entry and exit criteria for each phase

Without life-cycle models, it becomes tough for software project managers to monitor the progress of the report.

Stages:-

1. Requirement analysis and planning
2. Defining requirements
3. Designing the software
4. Developing the project
5. Testing
6. Deployment
7. Maintenance

**Q) Spring Boot Annotations**

# 🡪 Spring Boot Annotations is a form of metadata that provides data about a program. In other words, annotations are used to provide **supplemental** information about a program. It is not a part of the application that we develop. It does not have a direct effect on the operation of the code they annotate. It does not change the action of the compiled program.

In this section, we are going to discuss some important **Spring Boot Annotation** that we will use later in this tutorial.

## **Core Spring Framework Annotations**

**@Required:** It applies to the **bean** setter method. It indicates that the annotated bean must be populated at configuration time with the required property, else it throws an exception **BeanInitilizationException**.

**Example**

1. **public** **class** Machine
2. {
3. **private** Integer cost;
4. @Required
5. **public** **void** setCost(Integer cost)
6. {
7. **this**.cost = cost;
8. }
9. **public** Integer getCost()
10. {
11. **return** cost;
12. }
13. }

**@Autowired:** Spring provides annotation-based auto-wiring by providing @Autowired annotation. It is used to autowire spring bean on setter methods, instance variable, and constructor. When we use @Autowired annotation, the spring container auto-wires the bean by matching data-type.

**Example**

1. @Component
2. **public** **class** Customer
3. {
4. **private** Person person;
5. @Autowired
6. **public** Customer(Person person)
7. {
8. **this**.person=person;
9. }
10. }

**@Configuration:** It is a class-level annotation. The class annotated with @Configuration used by Spring Containers as a source of bean definitions.

**Example**

1. @Configuration
2. **public** **class** Vehicle
3. {
4. @BeanVehicle engine()
5. {
6. **return** **new** Vehicle();
7. }
8. }

**@ComponentScan:** It is used when we want to scan a package for beans. It is used with the annotation @Configuration. We can also specify the base packages to scan for Spring Components.

**Example**

1. @ComponentScan(basePackages = "com.javatpoint")
2. @Configuration
3. **public** **class** ScanComponent
4. {
5. // ...
6. }

**@Bean:** It is a method-level annotation. It is an alternative of XML <bean> tag. It tells the method to produce a bean to be managed by Spring Container.

**Example**

1. @Bean
2. **public** BeanExample beanExample()
3. {
4. **return** **new** BeanExample ();
5. }

## **Spring Framework Stereotype Annotations**

**@Component:** It is a class-level annotation. It is used to mark a Java class as a bean. A Java class annotated with **@Component** is found during the classpath. The Spring Framework pick it up and configure it in the application context as a **Spring Bean**.

**Example**

1. @Component
2. **public** **class** Student
3. {
4. .......
5. }

**@Controller:** The @Controller is a class-level annotation. It is a specialization of **@Component**. It marks a class as a web request handler. It is often used to serve web pages. By default, it returns a string that indicates which route to redirect. It is mostly used with **@RequestMapping** annotation.

**Example**

1. @Controller
2. @RequestMapping("books")
3. **public** **class** BooksController
4. {
5. @RequestMapping(value = "/{name}", method = RequestMethod.GET)
6. **public** Employee getBooksByName()
7. {
8. **return** booksTemplate;
9. }
10. }

**@Service:** It is also used at class level. It tells the Spring that class contains the **business logic**.

**Example**

1. **package** com.javatpoint;
2. @Service
3. **public** **class** TestService
4. {
5. **public** **void** service1()
6. {
7. //business code
8. }
9. }

**@Repository:** It is a class-level annotation. The repository is a **DAOs** (Data Access Object) that access the database directly. The repository does all the operations related to the database.

1. **package** com.javatpoint;
2. @Repository
3. **public** **class** TestRepository
4. {
5. **public** **void** delete()
6. {
7. //persistence code
8. }
9. }

## **Spring Boot Annotations**

* **@EnableAutoConfiguration:** It auto-configures the bean that is present in the classpath and configures it to run the methods. The use of this annotation is reduced in Spring Boot 1.2.0 release because developers provided an alternative of the annotation, i.e. **@SpringBootApplication**.
* **@SpringBootApplication:** It is a combination of three annotations **@EnableAutoConfiguration, @ComponentScan,** and **@Configuration**.

### Spring MVC and REST Annotations

* **@RequestMapping:** It is used to map the **web requests**. It has many optional elements like **consumes, header, method, name, params, path, produces**, and **value**. We use it with the class as well as the method.

**Example**

1. @Controller
2. **public** **class** BooksController
3. {
4. @RequestMapping("/computer-science/books")
5. **public** String getAllBooks(Model model)
6. {
7. //application code
8. **return** "bookList";
9. }

* **@GetMapping:** It maps the **HTTP GET** requests on the specific handler method. It is used to create a web service endpoint that **fetches** It is used instead of using: **@RequestMapping(method = RequestMethod.GET)**
* **@PostMapping:** It maps the **HTTP POST**requests on the specific handler method. It is used to create a web service endpoint that **creates** It is used instead of using: **@RequestMapping(method = RequestMethod.POST)**
* **@PutMapping:** It maps the **HTTP PUT** requests on the specific handler method. It is used to create a web service endpoint that **creates** or **updates** It is used instead of using: **@RequestMapping(method = RequestMethod.PUT)**
* **@DeleteMapping:** It maps the **HTTP DELETE** requests on the specific handler method. It is used to create a web service endpoint that **deletes**a resource. It is used instead of using: **@RequestMapping(method = RequestMethod.DELETE)**
* **@PatchMapping:** It maps the **HTTP PATCH**requests on the specific handler method. It is used instead of using: **@RequestMapping(method = RequestMethod.PATCH)**
* **@RequestBody:** It is used to **bind** HTTP request with an object in a method parameter. Internally it uses **HTTP MessageConverters** to convert the body of the request. When we annotate a method parameter with **@RequestBody,** the Spring framework binds the incoming HTTP request body to that parameter.
* **@ResponseBody:** It binds the method return value to the response body. It tells the Spring Boot Framework to serialize a return an object into JSON and XML format.
* **@PathVariable:** It is used to extract the values from the URI. It is most suitable for the RESTful web service, where the URL contains a path variable. We can define multiple @PathVariable in a method.
* **@RequestParam:** It is used to extract the query parameters form the URL. It is also known as a **query parameter**. It is most suitable for web applications. It can specify default values if the query parameter is not present in the URL.
* **@RequestHeader:** It is used to get the details about the HTTP request headers. We use this annotation as a **method parameter**. The optional elements of the annotation are **name, required, value, defaultValue.**For each detail in the header, we should specify separate annotations. We can use it multiple time in a method
* **@RestController:** It can be considered as a combination of **@Controller** and **@ResponseBody**annotations**.** The @RestController annotation is itself annotated with the @ResponseBody annotation. It eliminates the need for annotating each method with @ResponseBody.
* **@RequestAttribute:** It binds a method parameter to request attribute. It provides convenient access to the request attributes from a controller method. With the help of @RequestAttribute annotation, we can access objects that are populated on the server-side.

6) What are the types of IOC container in spring?

There are two types of IOC containers in spring framework.

1. BeanFactory
2. ApplicationContext

BeanFactory is the **basic container** whereas ApplicationContext is the **advanced container**. ApplicationContext extends the BeanFactory interface. ApplicationContext provides more facilities than BeanFactory such as integration with spring AOP, message resource handling for i18n etc.

### Using BeanFactory

The XmlBeanFactory is the implementation class for the BeanFactory interface. To use the BeanFactory, we need to create the instance of XmlBeanFactory class as given below:

1. Resource resource=**new** ClassPathResource("applicationContext.xml");
2. BeanFactory factory=**new** XmlBeanFactory(resource);

The constructor of XmlBeanFactory class receives the Resource object so we need to pass the resource object to create the object of BeanFactory.

#### **Using ApplicationContext**

The ClassPathXmlApplicationContext class is the implementation class of ApplicationContext interface. We need to instantiate the ClassPathXmlApplicationContext class to use the ApplicationContext as given below:

1. ApplicationContext context =
2. **new** ClassPathXmlApplicationContext("applicationContext.xml");

The constructor of ClassPathXmlApplicationContext class receives string, so we can pass the name of the xml file to create the instance of ApplicationContext.

8) What is the difference between constructor injection and setter injection?

|  |  |  |
| --- | --- | --- |
| **No.** | **Constructor Injection** | **Setter Injection** |
| 1) | No Partial Injection | Partial Injection |
| 2) | Desn't override the setter property | Overrides the constructor property if both are defined. |
| 3) | Creates new instance if any modification occurs | Doesn't create new instance if you change the property value |
| 4) | Better for too many properties | Better for few properties. |

9) What is autowiring in spring? What are the autowiring modes?

Autowiring enables the programmer to inject the bean automatically. We don't need to write explicit injection logic.

Let's see the code to inject bean using dependency injection.

1. <bean id="emp" **class**="com.javatpoint.Employee" autowire="byName" />

The autowiring modes are given below:

|  |  |  |
| --- | --- | --- |
| **No.** | **Mode** | **Description** |
| 1) | no | this is the default mode, it means autowiring is not enabled. |
| 2) | byName | injects the bean based on the property name. It uses setter method. |
| 3) | byType | injects the bean based on the property type. It uses setter method. |
| 4) | constructor | It injects the bean using constructor |

The "autodetect" mode is deprecated since spring 3.

10) What are the different bean scopes in spring?

There are 5 bean scopes in spring framework.

|  |  |  |
| --- | --- | --- |
| **No.** | **Scope** | **Description** |
| 1) | singleton | The bean instance will be only once and same instance will be returned by the IOC container. It is the default scope. |
| 2) | prototype | The bean instance will be created each time when requested. |
| 3) | request | The bean instance will be created per HTTP request. |
| 4) | session | The bean instance will be created per HTTP session. |
| 5) | globalsession | The bean instance will be created per HTTP global session. It can be used in portlet context only. |

11) In which scenario, you will use singleton and prototype scope?

Singleton scope should be used with EJB **stateless session bean** and prototype scope with EJB **stateful session bean**.

12) What are the transaction management supports provided by spring?

Spring framework provides two type of transaction management supports:

1. **Programmatic Transaction Management**: should be used for few transaction operations.
2. **Declarative Transaction Management**: should be used for many transaction operations.
3. **@Configuration and @AutoConfiguration and @ComponentScan**

* @ComponentScan scans for Spring components while @EnableAutoConfiguration is used for auto-configuring beans present in the classpath in Spring Boot applications.

The IoC container is responsible to instantiate, configure and assemble the objects. The IoC container gets informations from the XML file and works accordingly. The main tasks performed by IoC container are:

* to instantiate the application class
* to configure the object
* to assemble the dependencies between the objects

1. **What is IOC(Inversion of controller) container? What is DI(Dependency injection)?**
2. **Different types of IOC container do we have?**
3. **Difference between @Controller and @RestController**

* A key difference between a traditional MVC @Controller and the RESTful web service @RestController is the**way that the HTTP response body is created**. Rather than relying on a view technology to perform server-side rendering of the data to HTML, rest controller simply populates and returns the domain object itself.

|  |  |  |
| --- | --- | --- |
| **No.** | **@Controller** | **@RestController** |
| 1) | @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 annotations. |
| 2) | It is the specialized version of @Component annotation. | It is a specialised version of @Controller annotation |
| 3) | In @Controller, we can return a view in Spring Web MVC | In @RestController, we can not return a view |
| 4) | @Controller annotation indicates that the class is a “controller” like a web controller | @RestController annotation indicates that class is a controller where @RequestMapping methods assume @ResponseBody semantics by default |
| 5) | In @Controller, we need to use @ResponseBody on every handler method | In @RestController, we don’t need to use @ResponseBody on every handler method |
| 6) | It was added to Spring 2.5 version | It was added to Spring 4.0 version |

## 1. Spring Controller

In Spring, incoming requests are always handled by some controller. Usually [dispatcher servlet](https://howtodoinjava.com/spring5/webmvc/spring-dispatcherservlet-tutorial/) is responsible for identifying the controller and appropriate request handler method inside controller by URL matching.

#### 1.1. @Controller

In typical [spring mvc](https://howtodoinjava.com/spring-mvc-tutorial/) application, controller is indicated by annotation @Controller. This annotation serves as a specialization of @Component, allowing for implementation classes to be auto-detected through classpath scanning. It is typically used in combination with annotated handler methods based on the [@RequestMapping](https://howtodoinjava.com/spring-mvc/spring-mvc-requestmapping-annotation-examples/) annotation.

|  |
| --- |
| @Target(value=TYPE)  @Retention(value=RUNTIME)  @Documented  @Component  **public** @interface Controller {    //...  } |
| Controller.java |

A spring mvc controller is used typically in UI based applications where response is generally HTML content. The handler method returns the response “view name” which is resolved to a view technology file (e.g. JSP or FTL) by [view resolver](https://howtodoinjava.com/spring-boot/spring-boot-jsp-view-example/). And then parsed view content is sent back to browser client.

Imagine is the request is sent from [AJAX](https://howtodoinjava.com/ajax/complete-ajax-tutorial/) technology and client is actually looking for response in JSON format to that it can parse the result itself in browser and display as needed. Here, we must use @ResponseBody annotation along with @Controller.

@ResponseBody annotation indicates a method return value should be bound to the web response body i.e. no view resolver is needed.

#### 1.2. @RestController

As name suggest, it shall be used in case of REST style controllers i.e. handler methods shall return the JSON/XML response directly to client rather using view resolvers. It is a convenience annotation that is itself annotated with @Controller and @ResponseBody.

|  |
| --- |
| @Target(value=TYPE)  @Retention(value=RUNTIME)  @Documented  @Controller  @ResponseBody  **public** @interface RestController {    //...  } |
| RestController.java |

## 2. Difference between @Controller and @RestController

Clearly from above section, @RestController is a convenience annotation that does nothing more than **adds the @Controller and**[**@ResponseBody**](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/ResponseBody.html)**annotations in single statement**.

A key difference between a traditional MVC @Controller and the RESTful web service @RestController is the way that the HTTP response body is created. Rather than relying on a view technology to perform server-side rendering of the data to HTML, rest controller simply populates and returns the domain object itself.

The object data is be written directly to the HTTP response as JSON or XML and parsed by client to further process it either for modifying the existing view or for any other purpose.

#### 2.1. Using @Controller in spring mvc application

|  |
| --- |
| @Controller  @RequestMapping("employees")  **public** **class** EmployeeController  {    @RequestMapping(value = "/{name}", method = RequestMethod.GET)  **public** Employee getEmployeeByName(@PathVariable String name, Model model) {        //pull data and set in model    **return** employeeTemplate;    }  } |
| @Controller example without @ResponseBody |

#### 2.2. Using @Controller with @ResponseBody in spring

|  |
| --- |
| @Controller  @ResponseBody  @RequestMapping("employees")  **public** **class** EmployeeController  {    @RequestMapping(value = "/{name}", method = RequestMethod.GET, produces = "application/json")  **public** Employee getEmployeeByName(@PathVariable String name) {        //pull date    **return** employee;    }  } |
| @Controller example with @ResponseBody |

#### 2.3. Using @RestController in spring

|  |
| --- |
| @RestController  @RequestMapping("employees")  **public** **class** EmployeeController  {    @RequestMapping(value = "/{name}", method = RequestMethod.GET, produces = "application/json")  **public** Employee getEmployeeByName(@PathVariable String name) {        //pull date    **return** employee;    }  } |
| @RestController example |

[**Top 20 Spring Boot Interview Questions (2022) - javatpoint**](https://www.javatpoint.com/spring-boot-interview-questions)

[Top 20 Spring Boot Interview Questions (2022) - javatpoint](https://www.javatpoint.com/spring-boot-interview-questions)