**Spring Interview Questions**

**1. What are the major features in different versions of Spring Framework?**

|  |  |  |
| --- | --- | --- |
| **Features of Spring Framework** | | |
| **Version** | **Logo** | **Feature** | |
| **Spring 2.5** | spring 2.5 logo - Spring Interview Questions - Edureka! | This version was released in 2007. It was the first version which supported annotations. | |
| **Spring 3.0** | spring 3.0 logo - Spring Interview Questions - Edureka! | This version was released in 2009. Itmade full-fledged use of improvements in Java5 and also provided support to JEE6. | |
| **Spring 4.0** | Spring 4.0 logo - Spring Interview Questions - Edureka! | This version was released in 2013. This was the first version to provide full support to Java 8. | |

**2. What is a Spring Framework?**

* Spring is a powerful open source, application framework created to reduce the complexity of enterprise application development.
* It is light-weighted and loosely coupled.
* It has layered architecture, which allows you to select the components to use, while also providing a cohesive framework for J2EE application development.
* Spring framework is also called the framework of frameworks as it provides support to various other frameworks such as Struts, Hibernate, Tapestry, EJB, JSF etc.

**3. List the advantages of Spring Framework.**

* Because of Spring Frameworks layered architecture, you can use what you need and leave which you don’t.
* Spring Framework enables POJO (Plain Old Java Object) Programming which in turn enables continuous integration and testability.
* JDBC is simplified due to Dependency Injection and Inversion of Control.
* It is open-source and has no vendor lock-in.

**4. What are the different features of Spring Framework?**

Following are some of the major features of Spring Framework :

* **Lightweight:** Spring is lightweight when it comes to size and transparency.
* **Inversion of control (IOC):** The objects give their dependencies instead of creating or looking for dependent objects. This is called Inversion Of Control.
* **Aspect oriented Programming (AOP):** Aspect oriented programming in Spring supports cohesive development by separating application business logic from system services.
* **Container:**Spring Framework creates and manages the life cycle and configuration of the application objects.
* **MVC Framework:** Spring Framework’s MVC web application framework is highly configurable. Other frameworks can also be used easily instead of Spring MVC Framework.
* **Transaction Management:** Generic abstraction layer for transaction management is provided by the Spring Framework. Spring’s transaction support can be also used in container less environments.
* **JDBC Exception Handling:** The JDBC abstraction layer of the Spring offers an exception hierarchy, which simplifies the error handling strategy.

**5. How many modules are there in Spring Framework and what are they?**

There are around 20 modules which are generalized into Core Container, Data Access/Integration, Web, AOP (Aspect Oriented Programming), Instrumentation and Test.

* **Spring Core Container –**This layer is basically the core of Spring Framework.It contains the following modules :

1. Spring Core
2. Spring Bean
3. SpEL (Spring Expression Language)
4. Spring Context

* **Data Access/Integration –**This layer provides support to interact with the database. It contains the following modules :

1. JDBC (Java DataBase Connectivity)
2. ORM (Object Relational Mapping)
3. OXM (Object XML Mappers)
4. JMS (Java Messaging Service)
5. Transaction

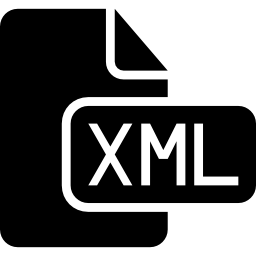
* **Web –**This layer provides support to create web application. It contains the following modules :

1. Web
2. Web – MVC
3. Web – Socket
4. Web – Portlet

* **Aspect Oriented Programming (AOP) –** In this layer you can use Advices, Pointcuts etc., to decouplethe code.
* **Instrumentation –**This layer provides support to class instrumentation and classloader implementations.
* **Test –**This layer provides support to testing with JUnit and TestNG.

Few Miscellaneous modules are given below:

* **Messaging –**This module provides support for STOMP. It also supports an annotation programming model that is used for routing and processing STOMP messages from WebSocket clients.
* **Aspects –**This module provides support to integration with AspectJ.

**6. What is a Spring configuration file?**

A Spring configuration file is an XML file. This file mainly contains the classes information. It describes how those classes are configured as well as introduced to each other. The XML configuration files, however, are verbose and more clean. If it’s not planned and written correctly, it becomes very difficult to manage in big projects.

**7. What are the different components of a Spring application?**

A Spring application, generally consists of following components:

* Interface: It defines the functions.
* Bean class: It contains properties, its setter and getter methods, functions etc.
* Spring Aspect Oriented Programming (AOP): Provides the functionality of cross-cutting concerns.
* Bean Configuration File: Contains the information of classes and how to configure them.
* User program: It uses the function.

**8. What are the various ways of using Spring Framework?**

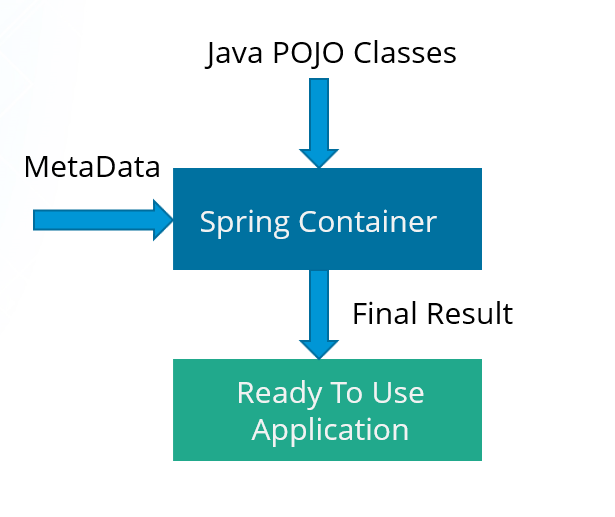
Spring Framework can be used in various ways. They are listed as follows:

1. As a Full-fledged Spring web application.
2. As a third-party web framework, using Spring Frameworks middle-tier.
3. For remote usage.
4. As Enterprise Java Bean which can wrap existing POJOs (Plain Old Java Objects).

The next section of Spring Interview Questions is on *Dependency Injection and IoC container*.

**Dependency Injection/ IoC Container – Spring Interview Questions**

**9. What is Spring IOC Container?**

****

At the core of the Spring Framework, lies the Spring container. The container creates the object, wires them together, configures them and manages their complete life cycle. The Spring container makes use of Dependency Injection to manage the components that make up an application. The container receives instructions for which objects to instantiate, configure, and assemble by reading the configuration metadata provided. This metadata can be provided either by XML, Java annotations or Java code.

**10. What do you mean by Dependency Injection?**

In Dependency Injection, you do not have to create your objects but have to describe how they should be created. You don’t connect your components and services together in the code directly, but describe which services are needed by which components in the configuration file. The IoC container will wire them up together.

**11. In how many ways can Dependency Injection be done?**

In general, dependency injection can be done in three ways, namely :

* Constructor Injection
* Setter Injection
* Interface Injection

In Spring Framework, only constructor and setter injections are used.

**12. Differentiate between constructor injection and setter injection.**

**Constructor Injection vs Setter Injection**

|  |  |
| --- | --- |
| **Constructor Injection** | **Setter Injection** |
| There is no partial injection. | There can be partial injection. |
| It doesn’t override the setter property. | It overrides the constructor property. |
| It will create a new instance if any modification is done. | It will not create new instance if any modification is done. |
| It works better for many properties. | It works better for few properties. |

**13. How many types of IOC containers are there in spring?**

1. **BeanFactory**: BeanFactory is like a factory class that contains a collection of beans. It instantiates the bean whenever asked for by clients.
2. **ApplicationContext**: The ApplicationContext interface is built on top of the BeanFactory interface. It provides some extra functionality on top BeanFactory.

**14. Differentiate between BeanFactory and ApplicationContext.**

**BeanFactory vs ApplicationContext**

|  |  |
| --- | --- |
| **BeanFactory** | **ApplicationContext** |
| It is an interface defined in org.springframework.beans.factory.**BeanFactory** | It is an interface defined in org.springframework.context.**ApplicationContext** |
| It uses Lazy initialization | It uses Eager/ Aggressive initialization |
| It explicitly provides a resource object using the syntax | It creates and manages resource objects on its own |
| It doesn’t supports internationalization | It supports internationalization |
| It doesn’t supports annotation based dependency | It supports annotation based dependency |

**15.  List some of the benefits of IoC.**

Some of the benefits of IoC are:

* It will minimize the amount of code in your application.
* It will make your application easy to test because it doesn’t require any singletons or JNDI lookup mechanisms in your unit test cases.
* It promotes loose coupling with minimal effort and least intrusive mechanism.
* It supports eager instantiation and lazy loading of the services.

Let’s move on to the next section of Spring Interview Questions, that is *Spring Beans Interview Questions*.

**Spring Beans – Spring Interview Questions**

**16. Explain Spring Beans?**

* They are the objects that form the backbone of the user’s application.
* Beans are managed by the Spring IoC container.
* They are instantiated, configured, wired and managed by a Spring IoC container
* Beans are created with the configuration metadata that the users supply to the container.

**17. How configuration metadata is provided to the Spring container?**

Configuration metadata can be provided to Spring container in following ways:

* **XML-Based configuration:**In Spring Framework, the dependencies and the services needed by beans are specified in configuration files which are in XML format. These configuration files usually contain a lot of bean definitions and application specific configuration options. They generally start with a bean tag. For example:

|  |  |
| --- | --- |
| 1  2  3 | <bean id="studentbean" class="org.edureka.firstSpring.StudentBean">   <property name="name" value="Edureka"></property>  </bean> |

* **Annotation-Based configuration**: Instead of using XML to describe a bean wiring, you can configure the bean into the component class itself by using annotations on the relevant class, method, or field declaration. By default, annotation wiring is not turned on in the Spring container. So, you need to enable it in your Spring configuration file before using it. For example:

|  |  |
| --- | --- |
| 1  2  3  4 | <beans>  <context:annotation-config/>  <!-- bean definitions go here -->  </beans> |

* **Java-based configuration:**The key features in Spring Framework’s new Java-configuration support are @Configuration annotated classes and @Bean annotated methods.

1. @Bean annotation plays the same role as the <bean/> element.

2.@Configuration classes allows to define inter-bean dependencies by simply calling other @Bean methods in the same class.

For example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @Configuration  public class StudentConfig  {  @Bean  public StudentBean myStudent()  { return new StudentBean(); }  } |

**18. How many bean scopes are supported by Spring?**

The Spring Framework supports five scopes. They are:

* **Singleton:**This provides scope for the bean definition to single instance per Spring IoC container.
* **Prototype:**This provides scope for a single bean definition to have any number of object instances.
* **Request:**This provides scope for a bean definition to an HTTP-request.
* **Session:**This provides scope for a bean definition to an HTTP-session.
* **Global-session:**This provides scope for a bean definition to an Global HTTP-session.

The last three are available only if the users use a web-aware ApplicationContext.

**19. What is the Bean life cycle in Spring Bean Factory Container?**

Bean life cycle in Spring Bean Factory Container is as follows:

1. The Spring container instantiates the bean from the bean’s definition in the XML file.
2. Spring populates all of the properties using the dependency injection, as specified in the bean definition.
3. The factory calls setBeanName() by passing the bean’s ID, if the bean implements the BeanNameAware interface.
4. The factory calls setBeanFactory() by passing an instance of itself, if the bean implements the BeanFactoryAware interface.
5. preProcessBeforeInitialization() methods are called if there are any BeanPostProcessors associated with the bean.
6. If an init-method is specified for the bean, then it will be called.
7. Finally, postProcessAfterInitialization() methods will be called if there are any BeanPostProcessors associated with the bean.

To understand it in better way check the below diagram:



**20. Explain inner beans in Spring.**

A bean can be declared as an inner bean only when it is used as a property of another bean. For defining a bean, the Spring’s XML based configuration metadata provides the use of <bean> element inside the <property> or <constructor-arg>. Inner beans are always anonymous and they are always scoped as prototypes. For example, let’s say we have one Student class having reference of Person class. Here we will be creating only one instance of Person class and use it inside Student.

Here’s a Student class followed by bean configuration file:

Student.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class Student  {  private Person person;  //Setters and Getters  }  public class Person  {  private String name;  private String address;  //Setters and Getters  } |

studentbean.xml

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <bean id=“StudentBean" class="com.edureka.Student">  <property name="person">  <!--This is inner bean -->  <bean class="com.edureka.Person">  <property name="name" value=“Scott"></property>  <property name="address" value=“Bangalore"></property>  </bean>  </property>  </bean> |

**21. Define Bean Wiring.**

When beans are combined together within the Spring container, it’s called wiring or bean wiring. The Spring container needs to know what beans are needed and how the container should use dependency injection to tie the beans together, while wiring beans.



** Trending Courses in this category**

**[Spring Framework Certification Training](https://www.edureka.co/spring-framework" \t "_blank)**

**[5 (2350)](https://www.edureka.co/spring-framework" \t "_blank)**

**[6k Learners Enrolled Live Class](https://www.edureka.co/spring-framework" \t "_blank)**

*[Best Price](https://www.edureka.co/spring-framework" \t "_blank)***[288  339](https://www.edureka.co/spring-framework" \t "_blank)**

**Similar Courses**[Java, J2EE & SOA Certification Training](https://www.edureka.co/java-j2ee-soa-training)[Python Scripting Certification Training](https://www.edureka.co/python-scripting)[Python Django Training and Certification](https://www.edureka.co/python-django)

**22. What do you understand by auto wiring and name the different modes of it?**

The Spring container is able to autowire relationships between the collaborating beans. That is, it is possible to let Spring resolve collaborators for your bean automatically by inspecting the contents of the BeanFactory.  
Different modes of bean auto-wiring are:

1. **no:** This is default setting which means no autowiring. Explicit bean reference should be used for wiring.
2. **byName:** It injects the object dependency according to name of the bean. It matches and wires its properties with the beans defined by the same names in the XML file.
3. **byType:**It injects the object dependency according to type. It matches and wires a property if its type matches with exactly one of the beans name in XML file.
4. **constructor:**It injects the dependency by calling the constructor of the class. It has a large number of parameters.
5. **autodetect:**First the container tries to wire using autowire by *constructor*, if it can’t then it tries to autowire by *byType*.

**23. What are the limitations with auto wiring?**

Following are some of the limitations you might face with auto wiring:

* **Overriding possibility:**You can always specify dependencies using <constructor-arg> and <property> settings which will override autowiring.
* **Primitive data type:**Simple properties such as primitives, Strings and Classes can’t be autowired.
* **Confusing nature:**Always prefer using explicit wiring because autowiring is less precise.

In the next section, we will discuss on *Spring Annotations Interview Questions*.

**Spring Annotations – Spring Interview Questions**

**24. What do you mean by  Annotation-based container configuration?**

Instead of using XML to describe a bean wiring, the developer moves the configuration into the component class itself by using annotations on the relevant class, method, or field declaration. It acts as an alternative to XML setups. For example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @Configuration  public class AnnotationConfig  {  @Bean  public MyDemo myDemo()   { return new MyDemoImpll(); }  } |

**25. How annotation wiring can be turned on in Spring?**

By default, Annotation wiring is not turned on in the Spring container. Thus, to use annotation based wiring we must enable it in our Spring configuration file by configuring **<context:annotation-config/>** element. For example:

|  |  |
| --- | --- |
| 1  2  3  4 | <beans xmlns="<http://www.springframework.org/schema/beans>" xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>" xmlns:context="<http://www.springframework.org/schema/context>">  <context:annotation-config/>  <beans ………… />  </beans> |

**26. What’s the difference between @Component, @Controller, @Repository & @Service annotations in Spring?**

**@Component:** This marks a java class as a bean. It is a generic stereotype for any Spring-managed component. The component-scanning mechanism of spring now can pick it up and pull it into the application context.

**@Controller:** This marks a class as a Spring Web MVC controller. Beans marked with it are automatically imported into the Dependency Injection container.

**@Service:** This annotation is a specialization of the component annotation. It doesn’t provide any additional behavior over the @Component annotation. You can use @Service over @Component in service-layer classes as it specifies intent in a better way.

**@Repository:** This annotation is a specialization of the @Component annotation with similar use and functionality. It provides additional benefits specifically for DAOs. It imports the DAOs into the DI container andmakes the unchecked exceptions eligible for translation into Spring DataAccessException.

**27. What do you understand by @Required annotation?**

@Required is applied to bean property setter methods. This annotation simply indicates that the affected bean property must be populated at the configuration time with the help of an explicit property value in a bean definition or with autowiring. If the affected bean property has not been populated, the container will throw BeanInitializationException.

For example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public class Employee  {  private String name;  @Required  public void setName(String name)  {this.name=name; }  public string getName()  { return name; }  } |

**28. What do you understand by @Autowired annotation?**

The **@Autowired** annotation provides more accurate control over where and how autowiring should be done. This annotation is used to autowire bean on the setter methods, constructor, a property or methods with arbitrary names or multiple arguments. By default, it is a type driven injection.

For Example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public class Employee  {  private String name;  @Autowired  public void setName(String name)  {this.name=name; }  public string getName()  { return name; }  } |

**29. What do you understand by @Qualifier annotation?**

When you create more than one bean of the same type and want to wire only one of them with a property  you can use the **@Qualifier** annotation along with **@Autowired** to remove the ambiguity by specifying which exact bean should be wired.

For example, here we have two classes, Employee and EmpAccount respectively. In EmpAccount, using @Qualifier its specified that bean with id emp1 must be wired.

Employee.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public class Employee  {  private String name;  @Autowired  public void setName(String name)  { this.name=name; }  public string getName()  { return name; }  } |

EmpAccount.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public class EmpAccount  {  private Employee emp;  @Autowired  @Qualifier(emp1)  public void showName()  {  System.out.println(“Employee name : ”+emp.getName);  }  } |

**30.  What do you understand by @RequestMapping annotation?**

@RequestMapping annotation is used for mapping a particular HTTP request method to a specific class/ method in controller that will be handling the respective request. This annotation can be applied at both levels:

* **Class level**: Maps the URL of the request
* **Method level**: Maps the URL as well as HTTP request method

Next section of Spring Interview Questions is on *Data Access*.

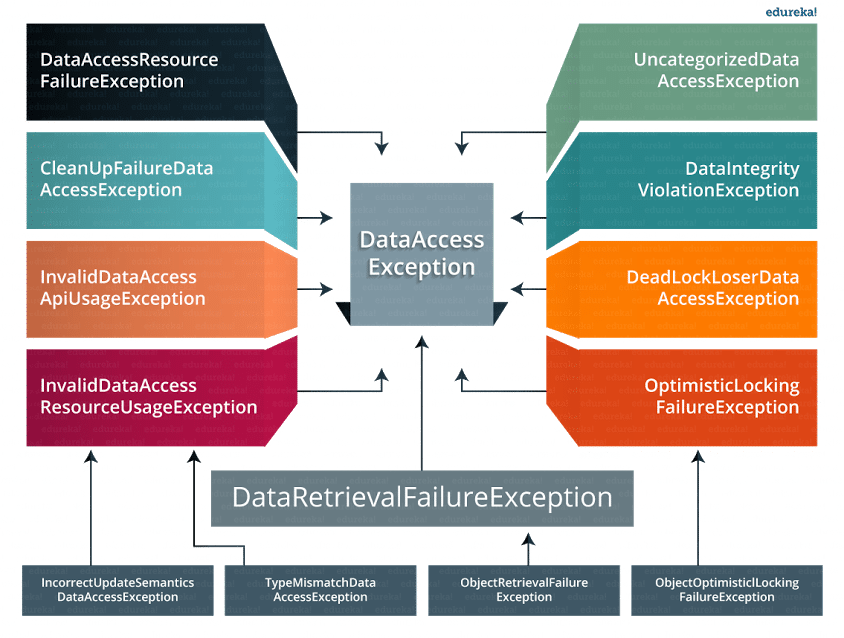
**Data Access – Spring Interview Questions**

**31. Describe Spring DAO support?**

The Data Access Object (DAO) support in Spring makes it easy to work with data access technologies like JDBC, Hibernate or JDO in a consistent way. This allows one to switch between the persistence technologies easily. It also allows you to code without worrying about catching exceptions that are specific to each ofthese technology.

**32. Name the exceptions thrown by the Spring DAO classes.**

See the below diagram, it depicts all the Spring DAO classes in the hierarchical order.



**33.  Which classes are present in spring JDBC API?**

Classes present in JDBC API are as follows:

1. JdbcTemplate
2. SimpleJdbcTemplate
3. NamedParameterJdbcTemplate
4. SimpleJdbcInsert
5. SimpleJdbcCall

**34. What are the ways by which Hibernate can be accessed using Spring?**

There are two ways by which we can access Hibernate using Spring:

1. Inversion of Control with a Hibernate Template and Callback
2. Extending HibernateDAOSupport and Applying an AOP Interceptor node

**35. Name the types of transaction management that Spring supports.**

Two types of transaction management are supported by Spring. They are:

1. **Programmatic transaction management:**In this, the transaction is managed with the help of programming. It provides you extreme flexibility, but it is very difficult to maintain.
2. **Declarative transaction management:**In this, the transaction management is separated from the business code. Only annotations or XML based configurations are used to manage the transactions.

**36. What are the different ORM’s supported by Spring?**

Different ORM’s supported by Spring are depicted via the below diagram:

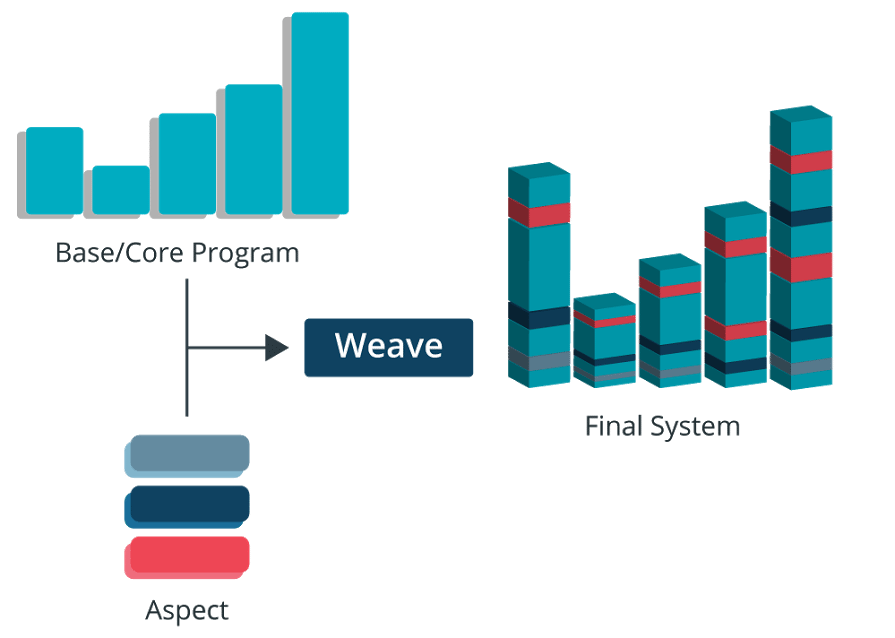
The next section of Spring interview questions discusses on *Spring AOP Interview Questions*.

[**Spring Framework Certification TrainingWatch The Course Preview**](https://www.edureka.co/spring-framework)

**Aspect Oriented Programming (AOP) – Spring Interview Questions**

**37. Describe AOP.**

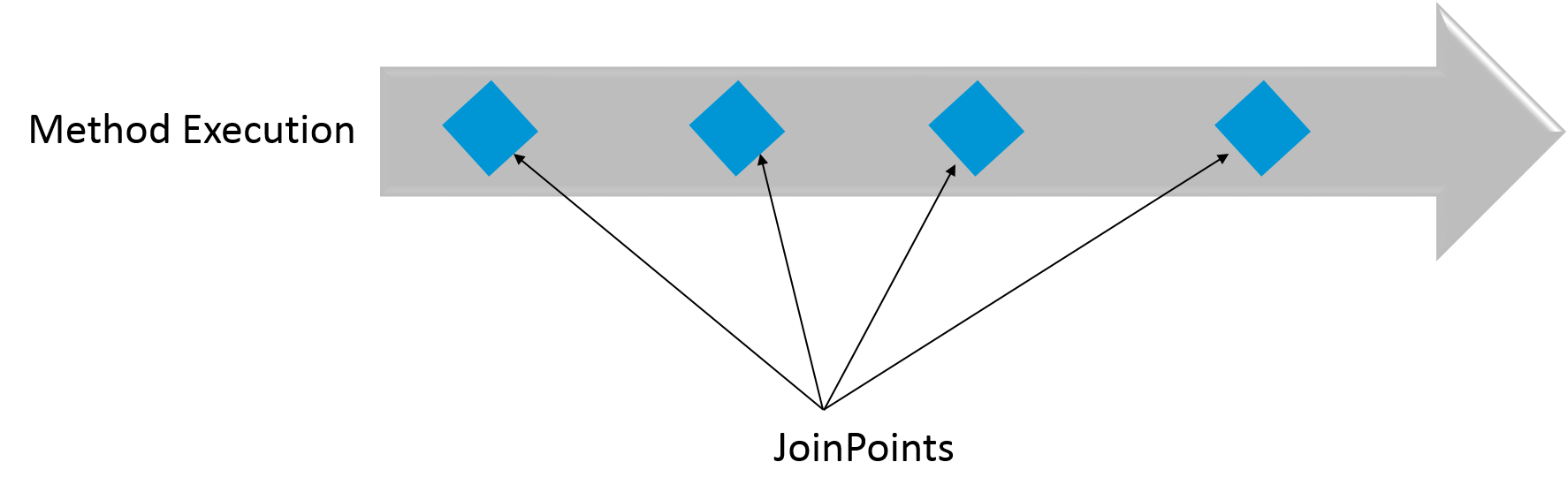
Aspect-oriented programming or AOP is a programming technique which allows programmers to modularize crosscutting concerns or behavior that cuts across the typical divisions of responsibility. Examples of cross-cutting concerns can be logging and transaction management. The core of AOP is an *aspect*. It encapsulates behaviors that can affect multiple classes into reusable modules.



**38. What do you mean by Aspect?**

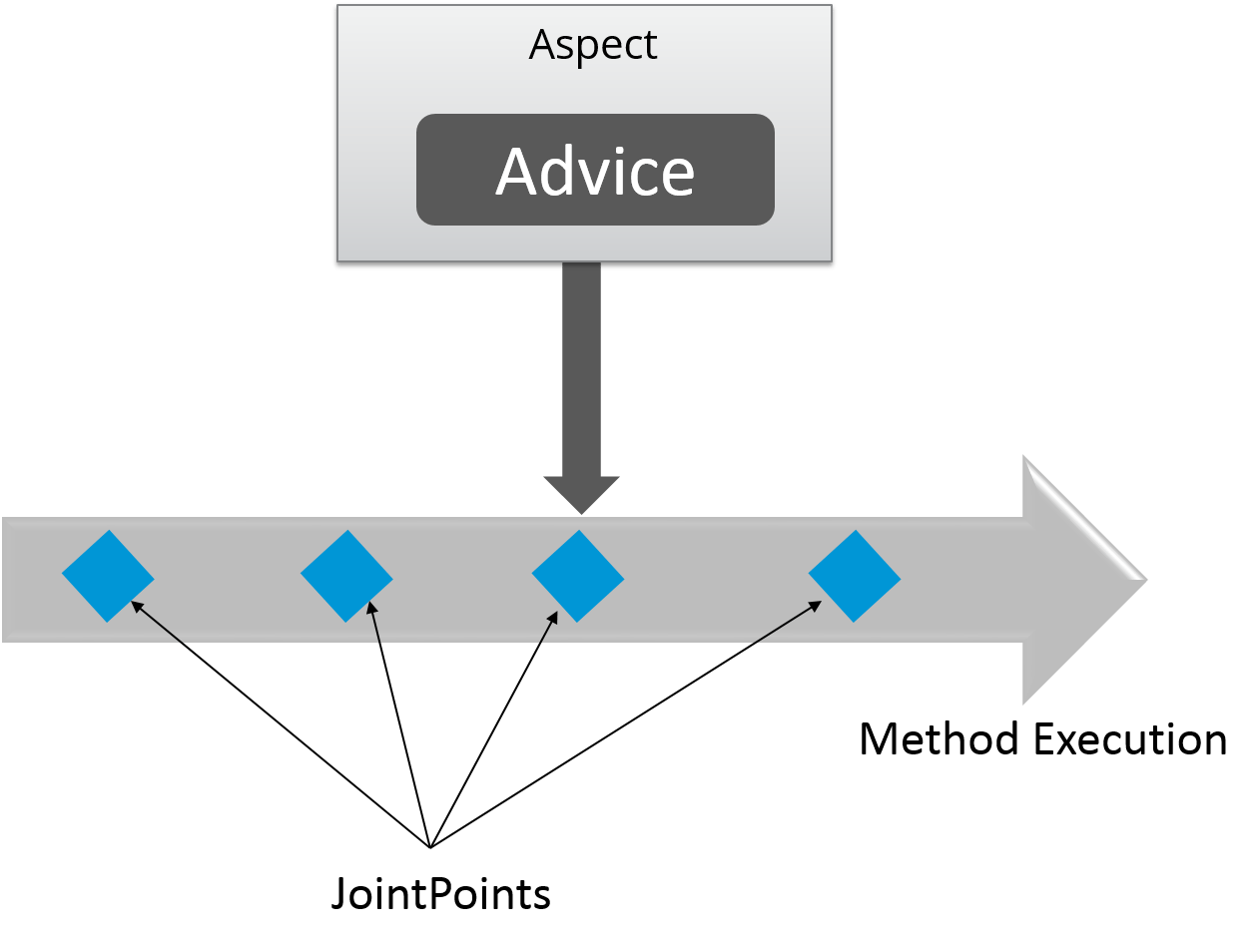
Aspect is a modularization of concern which cuts across multiple objects. Transaction management is a good example of a crosscutting concern in J2EE applications. Aspects are implemented using regular classes or regular classes annotated with the @Aspect annotation in Spring Framework.

**39. Explain JoinPoint.**

A point during the execution of a program is called JoinPoint, such as the execution of a method or the handling of an exception. In Spring AOP, a joinpoint always represents a method execution.

**40. What is an Advice?**

An Action taken by an aspect at a particular joinpoint is known as an Advice. Spring AOP uses an advice as an interceptor, maintaining a chain of interceptors “around” the join point.



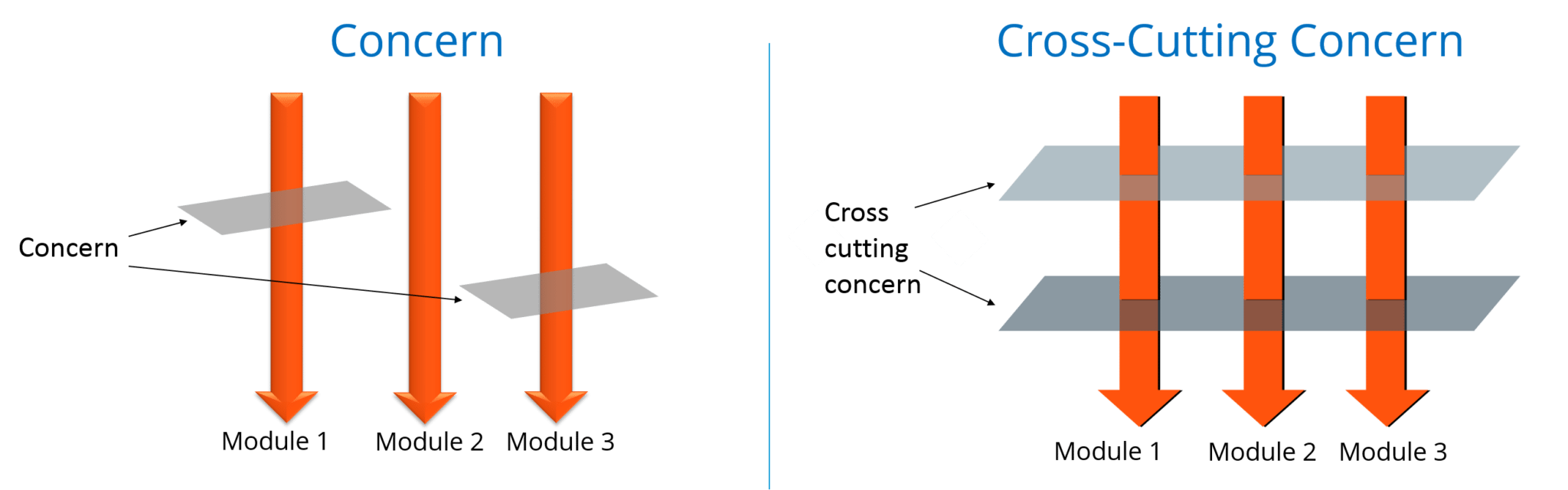
**41. What are the different types of Advices?**

1. **Before:** These types of advices execute before the joinpoint methods and are configured using **@Before**annotation mark.
2. **After returning:**These types of advices execute after the joinpoint methods completes executing normally and are configured using @AfterReturning annotation mark.
3. **After throwing:** These types of advices execute only if joinpoint method exits by throwing an exception and are configured using @AfterThrowing annotation mark.
4. **After (finally):**These types of advices execute after a joinpoint method, regardless of the method’s exit whether normally or exceptional return and are configured using @After annotation mark.
5. **Around:**These types of advices execute before and after a joinpoint and are configured using @Around annotation mark.

**42. Point out the difference between concern and cross-cutting concern in Spring AOP?**

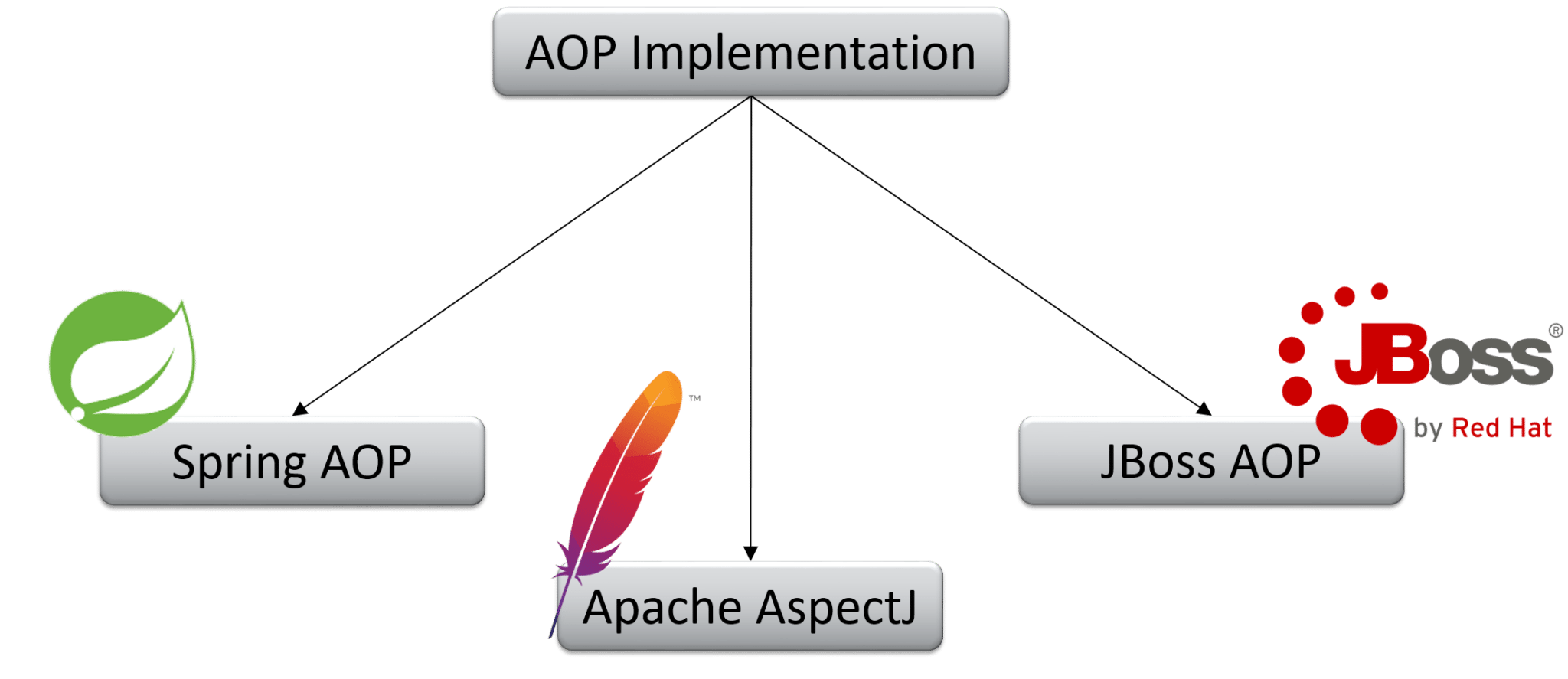
The concern is the behavior we want to have in a particular module of an application. It can be defined as a functionality we want to implement.

The cross-cutting concern is a concern which is applicable throughout the application. This affects the entire application. For example, logging, security and data transfer are the concerns needed in almost every module of an application, thus they are the cross-cutting concerns.



**43. What are the different AOP implementations?**

Different AOP implementations are depicted by the below diagram:



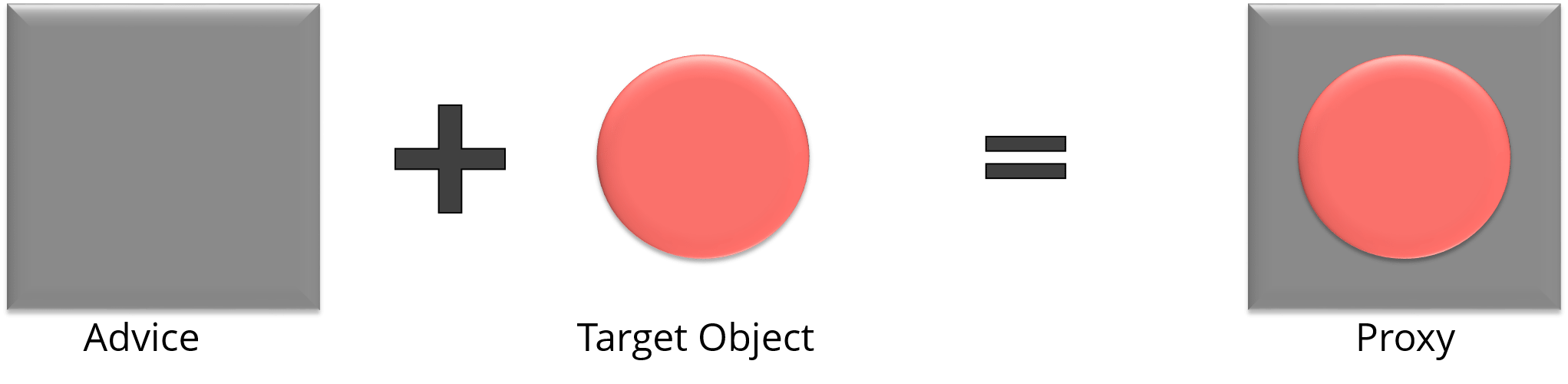
**44. What are the difference between Spring AOP and AspectJ AOP?**

**Spring AOP vs AspectJ AOP**

|  |  |
| --- | --- |
| **Spring AOP** | **AspectJ AOP** |
| Runtime weaving through proxy is done | Compile time weaving through AspectJ Java tools is done |
| It supports only method level PointCut | It suports field level Pointcuts |
| It is DTD based | It is schema based and Annotation configuration |

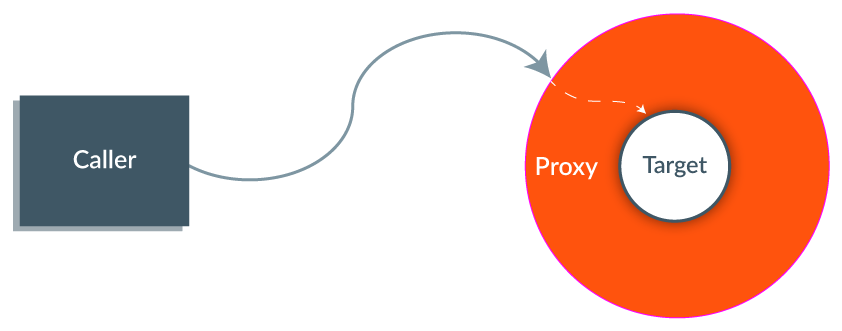
**45. What do you mean by Proxy in Spring Framework?**

An object which is created after applying advice to a target object is known as a Proxy. In case of client objects the target object and the proxy object are the same.



**46. In Spring, what is Weaving?**

The process of linking an aspect with other application types or objects to create an advised object is called Weaving. In Spring AOP, weaving is performed at runtime. Refer the below diagram:

The last section of Spring interview questions is on *Spring MVC Interview Questions*.

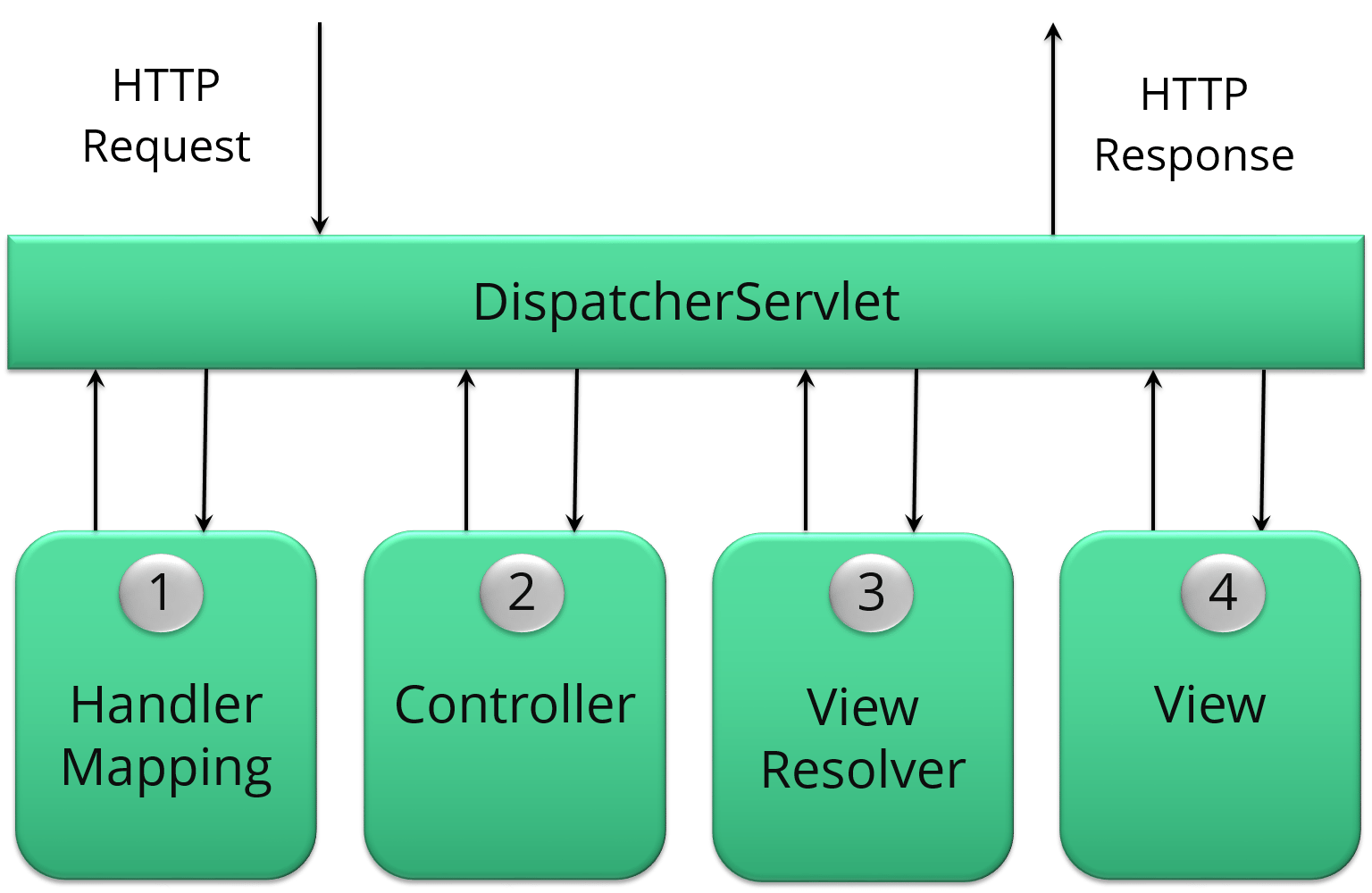
**MVC (Model-View-Controller) – Spring Interview Questions**

**47. What do you mean by Spring MVC framework?**

The Spring web MVC framework provides model-view-controller architecture and ready to use components that are used to develop flexible and loosely coupled web applications. The MVC pattern helps in separating the different aspects of the application like input logic, business logic and UI logic, while providing a loose coupling between all these elements.

**48. Describe DispatcherServlet.**

The DispatcherServlet is the core of Spring Web MVC framework. It handles all the HTTP requests and responses. The DispatcherServlet receives the entry of handler mapping from the configuration file and forwards the request to the controller. The controller then returns an object of Model And View. The DispatcherServlet checks the entry of view resolver in the configuration file and calls the specified view component.

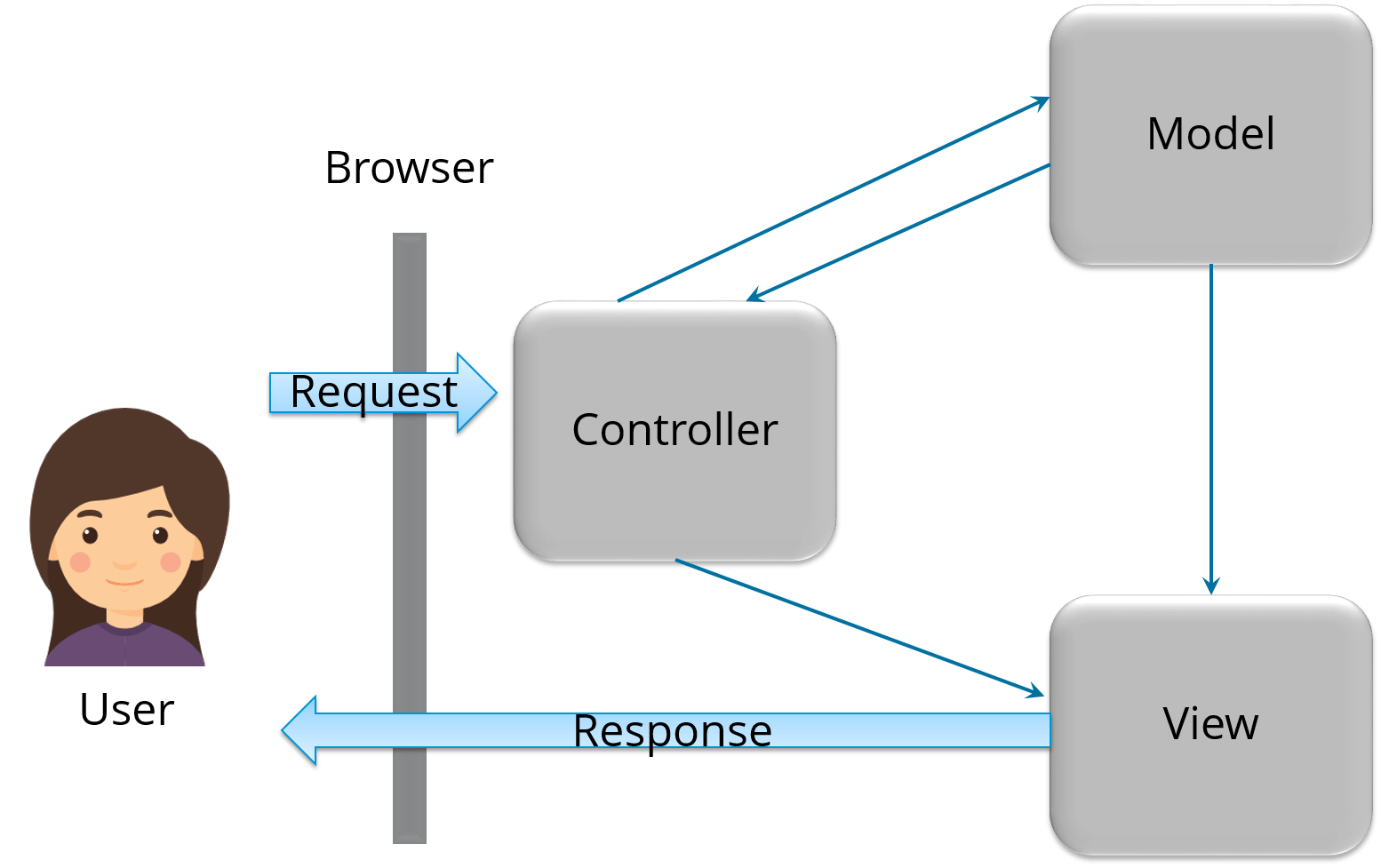


**49. Explain WebApplicationContext.**

The WebApplicationContext is an extension of the plain ApplicationContext. It has some extra features that are necessary for web applications. It differs from a normal ApplicationContext in terms of its capability of resolving themes and in deciding which servlet it is associated with.

**50. In Spring MVC framework, what is controller?**

Controllers provide access to the application behavior. These behaviors are generally defined through a service interface. Controllers interpret the user input and transform it into a model which is represented to the user by the view. In Spring, controller is implemented in a very abstract way. It also enables you to create a wide variety of controllers.



**Different ways to break circular dependency**

**Approach1:**

By using Autowiring by property / setter method instead of constructor Autowiring. Because by this approach dependency injection happens after the bean objects are created.

@Component

public class CircularDependencyA {

    private CircularDependencyB circB;

    @Autowired

    public void setCircB(CircularDependencyB circB) {

        this.circB = circB;

    }

    public CircularDependencyB getCircB() {

        return circB;

    }

}

@Component

public class CircularDependencyB {

    private CircularDependencyA circA;

    private String message = "Hi!";

    @Autowired

    public void setCircA(CircularDependencyA circA) {

        this.circA = circA;

    }

    public String getMessage() {

        return message;

    }

}

**Approach2:**

Second approach @Lazy annotation in constructor injection in one of the bean.

@Component

public class CircularDependencyA {

    private CircularDependencyB circB;

    @Autowired

    public CircularDependencyA(@Lazy CircularDependencyB circB) {

        this.circB = circB;

    }

}

**Approach3:**

By using @PostConstruct

@Component

public class CircularDependencyA {

    @Autowired

    private CircularDependencyB circB;

    @PostConstruct

    public void init() {

        circB.setCircA(this);

    }

    public CircularDependencyB getCircB() {

        return circB;

    }

}

@Component

public class CircularDependencyB {

    private CircularDependencyA circA;

    private String message = "Hi!";

    public void setCircA(CircularDependencyA circA) {

        this.circA = circA;

    }

    public String getMessage() {

        return message;

    }

}

**Spring Annotation**

The Java Programming language provided support for Annotations from Java 5.0. Leading Java frameworks were quick to adopt annotations and the Spring Framework started using annotations from the release 2.5. Due to the way they are defined, annotations provide a lot of context in their declaration.

Prior to annotations, the behavior of the Spring Framework was largely controlled through XML configuration. Today, the use of annotations provide us tremendous capabilities in how we configure the behaviors of the Spring Framework.

In this post, we’ll take a look at the annotations available in the Spring Framework.

Core Spring Framework Annotations

**@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!)

public class Customer {

@Autowired

private Person person;

private int type;

}

public class Customer {

@Autowired

private Person person;

private int type;

}

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.

public class Customer {

private Person person;

@Autowired

public void setPerson (Person person) {

this.person=person;

}

}

public class Customer {

private Person person;

@Autowired

public void setPerson (Person person) {

this.person=person;

}

}

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.

@Component

public class Customer {

private Person person;

@Autowired

public Customer (Person person) {

this.person=person;

}

}

@Component

public class Customer {

private Person person;

@Autowired

public Customer (Person person) {

this.person=person;

}

}

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.

@Component

public class BeanB1 implements BeanInterface {

//

}

@Component

public class BeanB2 implements BeanInterface {

//

}

@Component

public class BeanB1 implements BeanInterface {

//

}

@Component

public class BeanB2 implements BeanInterface {

//

}

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.

@Component

public class BeanA {

@Autowired

@Qualifier("beanB2")

private BeanInterface dependency;

...

}

@Component

public class BeanA {

@Autowired

@Qualifier("beanB2")

private BeanInterface dependency;

...

}

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

Spring Framework 5Learn the Spring Framework with my Spring Framework 5 – Beginner to Guru Course!

**@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:

@Configuration

public class DataConfig{

@Bean

public DataSource source(){

DataSource source = new OracleDataSource();

source.setURL();

source.setUser();

return source;

}

@Bean

public PlatformTransactionManager manager(){

PlatformTransactionManager manager = new BasicDataSourceTransactionManager();

manager.setDataSource(source());

return manager;

}

}

@Configuration

public class DataConfig{

@Bean

public DataSource source(){

DataSource source = new OracleDataSource();

source.setURL();

source.setUser();

return source;

}

@Bean

public PlatformTransactionManager manager(){

PlatformTransactionManager manager = new BasicDataSourceTransactionManager();

manager.setDataSource(source());

return manager;

}

}

**@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.

Checkout this post for an in depth look at the Component Scan 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:

@Configuration

public class AppConfig{

@Bean

public Person person(){

return new Person(address());

}

@Bean

public Address address(){

return new Address();

}

}

@Configuration

public class AppConfig{

@Bean

public Person person(){

return new Person(address());

}

@Bean

public Address address(){

return new Address();

}

}

**@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.

**Spring Framework Stereotype Annotations**

**@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 an 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**

**@Controller**

This annotation is used on Java classes that play the role of controller in your application. The @Controller annotation allows autodetection of component classes in the classpath and auto-registering bean definitions for them. To enable autodetection of such annotated controllers, you can add component scanning to your configuration. The Java class annotated with @Controller is capable of handling multiple request mappings.

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

**@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:

@Controller

@RequestMapping("/welcome")

public class WelcomeController{

@RequestMapping(method = RequestMethod.GET)

public String welcomeAll(){

return "welcome all";

}

}

@Controller

@RequestMapping("/welcome")

public class WelcomeController{

@RequestMapping(method = RequestMethod.GET)

public String welcomeAll(){

return "welcome all";

}

}

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.

The @RequestMapping annotation is very versatile. Please see my in depth post on Request Mapping bere.

**@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:

@RequestMapping("/cookieValue")

public void getCookieValue(@CookieValue "JSESSIONID" String cookie){

}

@RequestMapping("/cookieValue")

public void getCookieValue(@CookieValue "JSESSIONID" String cookie){

}

**@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.

@CrossOrigin (maxAge = 3600)

@RestController

@RequestMapping("/account")

public class AccountController {

@CrossOrigin(origins = "http://example.com")

@RequestMapping("/message")

public Message getMessage() {

// ...

}

@RequestMapping("/note")

public Note getNote() {

// ...

}

}

@CrossOrigin(maxAge = 3600)

@RestController

@RequestMapping("/account")

public class AccountController {

@CrossOrigin(origins = "http://example.com")

@RequestMapping("/message")

public Message getMessage() {

// ...

}

@RequestMapping("/note")

public Note getNote() {

// ...

}

}

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**

Spring framework 4.3 introduced the following method-level variants of @RequestMapping annotation to better express the semantics of the annotated methods. Using these annotations have become the standard ways of defining the endpoints. They act as wrapper to @RequestMapping.

These annotations can be used with Spring MVC and Spring WebFlux.

**@GetMapping**

This annotation is used for mapping HTTP GET requests onto specific handler methods. @GetMapping is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.GET)

**@PostMapping**

This annotation is used for mapping HTTP POST requests onto specific handler methods. @PostMapping is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.POST)

**@PutMapping**

This annotation is used for mapping HTTP PUT requests onto specific handler methods. @PutMapping is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.PUT)

**@PatchMapping**

This annotation is used for mapping HTTP PATCH requests onto specific handler methods. @PatchMapping is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.PATCH)

**@DeleteMapping**

This annotation is used for mapping HTTP DELETE requests onto specific handler methods. @DeleteMapping is a composed annotation that acts as a shortcut for @RequestMapping (method = RequestMethod.DELETE)

**@ExceptionHandler**

This annotation is used at method levels to handle exception at the controller level. The @ExceptionHandler annotation is used to define the class of exception it will catch. You can use this annotation on methods that should be invoked to handle an exception. The @ExceptionHandler values can be set to an array of Exception types. If an exception is thrown that matches one of the types in the list, then the method annotated with matching @ExceptionHandler will be invoked.

**@InitBinder (spring mvc related)**

This annotation is a method level annotation that plays the role of identifying the methods which initialize the WebDataBinder - a DataBinder that binds the request parameter to JavaBean objects. To customise request parameter data binding, you can use @InitBinder annotated methods within our controller. The methods annotated with @InitBinder all argument types that handler methods support.

The @InitBinder annotated methods will get called for each HTTP request if you don’t specify the value element of this annotation. The value element can be a single or multiple form names or request parameters that the init binder method is applied to.

**@Mappings and @Mapping**

This annotation is used on fields. The @Mapping annotation is a meta annotation that indicates a web mapping annotation. When mapping different field names, you need to configure the source field to its target field and to do that you have to add the @Mappings annotation. This annotation accepts an array of @Mapping having the source and the target fields.

**@MatrixVariable**

Can be used in both GET and POST call

This annotation is used to annotate request handler method arguments so that Spring can inject the relevant bits of matrix URI. Matrix variables can appear on any segment each separated by a semicolon. If a URL contains matrix variables, the request mapping pattern must represent them with a URI template. The @MatrixVariable annotation ensures that the request is matched with the correct matrix variables of the URI.

http://localhost:8080/spring-mvc-java/employeeArea/workingArea=rh,informatics,admin

or like this:

http://localhost:8080/spring-mvc-java

/employeeArea/workingArea=rh;workingArea=informatics;workingArea=admin

<http://localhost:8080/spring-mvc-java/employeesContacts/contactNumber=223334411>

@RequestMapping (value = "/employeesContacts/{contactNumber}",

  method = RequestMethod.GET)

@ResponseBody

public ResponseEntity<List<Employee>> getEmployeeBycontactNumber(

  @MatrixVariable(required = true) String contactNumber) {

    List<Employee> employeesList = new ArrayList<Employee>();

    ...

    return new ResponseEntity<List<Employee>>(employeesList, HttpStatus.OK);

}

<http://localhost:8080/spring-mvc-java/employees/John;beginContactNumber=22001>

@RequestMapping(value = "/employees/{name}", method = RequestMethod.GET)

@ResponseBody

public ResponseEntity<List<Employee>> getEmployeeByNameAndBeginContactNumber(

@PathVariable String name, @MatrixVariable String beginContactNumber) {

List<Employee> employeesList = new ArrayList<Employee>();

...

return new ResponseEntity<>(employeesList, HttpStatus.OK);

}

<http://localhost:8080/spring-mvc-java/employeeData/id=1;name=John;contactNumber=2200112334>

@GetMapping("employeeData/{employee}")

@ResponseBody

public ResponseEntity<Map<String, String>> getEmployeeData(

  @MatrixVariable Map<String, String> matrixVars) {

    return new ResponseEntity<>(matrixVars, HttpStatus.OK);

}

<http://localhost:8080/spring-mvc-java/>

  companyEmployee/id=2;name=Xpto/employeeData/id=1;name=John;

  contactNumber=2200112334

@RequestMapping(

 value = "/companyEmployee/{company}/employeeData/{employee}",

 method = RequestMethod.GET)

@ResponseBody

public ResponseEntity<Map<String, String>> getEmployeeDataFromCompany(

  @MatrixVariable(pathVar = "employee") Map<String, String> matrixVars) {

  ...

}

Partial Reading

<http://localhost:8080/spring-mvc-java/>

  companyData/id=2;name=Xpto/employeeData/id=1;name=John;

  contactNumber=2200112334

@MatrixVariable(value="name", pathVar="company") String name

**@PathVariable**

This annotation is used to annotate request handler method arguments. The @RequestMapping annotation can be used to handle dynamic changes in the URI where certain URI value acts as a parameter. You can specify this parameter using a regular expression. The @PathVariable annotation can be used declare this parameter.

**@RequestAttribute**

This annotation is used to bind the request attribute to a handler method parameter. Spring retrieves the named attributes value to populate the parameter annotated with @RequestAttribute. While the @RequestParam annotation is used bind the parameter values from query string, the @RequestAttribute is used to access the objects which have been populated on the server side.

**@RequestBody**

This annotation is used to annotate request handler method arguments. The @RequestBody annotation indicates that a method parameter should be bound to the value of the HTTP request body. The HttpMessageConveter is responsible for converting from the HTTP request message to object.

**@RequestHeader**

This annotation is used to annotate request handler method arguments. The @RequestHeader annotation is used to map controller parameter to request header value. When Spring maps the request, @RequestHeader checks the header with the name specified within the annotation and binds its value to the handler method parameter. This annotation helps you to get the header details within the controller class.

**@RequestParam**

This annotation is used to annotate request handler method arguments. Sometimes you get the parameters in the request URL, mostly in GET requests. In that case, along with the @RequestMapping annotation you can use the @RequestParam annotation to retrieve the URL parameter and map it to the method argument. The @RequestParam annotation is used to bind request parameters to a method parameter in your controller.

**@RequestPart**

This annotation is used to annotate request handler method arguments. The @RequestPart annotation can be used instead of @RequestParam to get the content of a specific multipart and bind to the method argument annotated with @RequestPart. This annotation takes into consideration the “Content-Type” header in the multipart(request part).

**@ResponseBody**

This annotation is used to annotate request handler methods. The @ResponseBody annotation is similar to the @RequestBody annotation. The @ResponseBody annotation indicates that the result type should be written straight in the response body in whatever format you specify like JSON or XML. Spring converts the returned object into a response body by using the HttpMessageConveter.

**@ResponseStatus**

This annotation is used on methods and exception classes. @ResponseStatus marks a method or exception class with a status code and a reason that must be returned. When the handler method is invoked the status code is set to the HTTP response which overrides the status information provided by any other means. A controller class can also be annotated with @ResponseStatus which is then inherited by all @RequestMapping methods.

**@ControllerAdvice**

This annotation is applied at the class level. As explained earlier, for each controller you can use @ExceptionHandler on a method that will be called when a given exception occurs. But this handles only those exception that occur within the controller in which it is defined. To overcome this problem you can now use the @ControllerAdvice annotation. This annotation is used to define @ExceptionHandler, @InitBinder and @ModelAttribute methods that apply to all @RequestMapping methods. Thus if you define the @ExceptionHandler annotation on a method in @ControllerAdvice class, it will be applied to all the controllers.

**@RestController**

This annotation is used at the class level. The @RestController annotation marks the class as a controller where every method returns a domain object instead of a view. By annotating a class with this annotation you no longer need to add @ResponseBody to all the RequestMapping method. It means that you no more use view-resolvers or send html in response. You just send the domain object as HTTP response in the format that is understood by the consumers like JSON.

@RestController is a convenience annotation which combines @Controller and @ResponseBody.

**@RestControllerAdvice**

This annotation is applied on Java classes. @RestControllerAdvice is a convenience annotation which combines @ControllerAdvice and @ResponseBody. This annotation is used along with the @ExceptionHandler annotation to handle exceptions that occur within the controller.

**@SessionAttribute**

This annotation is used at method parameter level. The @SessionAttribute annotation is used to bind the method parameter to a session attribute. This annotation provides a convenient access to the existing or permanent session attributes.

**@SessionAttributes**

This annotation is applied at type level for a specific handler. The @SessionAtrributes annotation is used when you want to add a JavaBean object into a session. This is used when you want to keep the object in session for short lived. @SessionAttributes is used in conjunction with @ModelAttribute.

Consider this example.

@ModelAttribute("person")

public Person getPerson(){}

// within the same controller as above snippet

@Controller

@SeesionAttributes(value="person", types={Person.class})

public class PersonController{}

1

2

3

4

5

6

@ModelAttribute("person")

public Person getPerson(){}

// within the same controller as above snippet

@Controller

@SeesionAttributes(value="person", types={Person.class})

public class PersonController{}

The @ModelAttribute name is assigned to the @SessionAttributes as value. The @SessionAttributes has two elements. The value element is the name of the session in the model and the types element is the type of session attributes in the model.

Spring Cloud Annotations

@EnableConfigServer

This annotation is used at the class level. When developing a project with a number of services, you need to have a centralized and straightforward manner to configure and retrieve the configurations about all the services that you are going to develop. One advantage of using a centralized config server is that you don’t need to carry the burden of remembering where each configuration is distributed across multiple and distributed components.

You can use Spring cloud’s @EnableConfigServer annotation to start a config server that the other applications can talk to.

@EnableEurekaServer

This annotation is applied to Java classes. One problem that you may encounter while decomposing your application into microservices is that, it becomes difficult for every service to know the address of every other service it depends on. There comes the discovery service which is responsible for tracking the locations of all other microservices.

Netflix’s Eureka is an implementation of a discovery server and integration is provided by Spring Boot. Spring Boot has made it easy to design a Eureka Server by just annotating the entry class with @EnableEurekaServer.

@EnableDiscoveryClient

This annotation is applied to Java classes. In order to tell any application to register itself with Eureka you just need to add the @EnableDiscoveryClientannotation to the application entry point. The application that’s now registered with Eureka uses the Spring Cloud Discovery Client abstraction to interrogate the registry for its own host and port.

@EnableCircuitBreaker

This annotation is applied on Java classes that can act as the circuit breaker. The circuit breaker pattern can allow a micro service continue working when a related service fails, preventing the failure from cascading. This also gives the failed service a time to recover.

The class annotated with @EnableCircuitBreaker will monitor, open, and close the circuit breaker.

@HystrixCommand

This annotation is used at the method level. Netflix’s Hystrix library provides the implementation of Circuit Breaker pattern. When you apply the circuit breaker to a method, Hystrix watches for the failures of the method. Once failures build up to a threshold, Hystrix opens the circuit so that the subsequent calls also fail. Now Hystrix redirects calls to the method and they are passed to the specified fallback methods.

Hystrix looks for any method annotated with the @HystrixCommand annotation and wraps it into a proxy connected to a circuit breaker so that Hystrix can monitor it.

Consider the following example:

@Service

public class BookService{

private final RestTemplate restTemplate;

public BookService(RestTemplate rest){

this.restTemplate = rest;

}

@HystrixCommand(fallbackMethod = "newList") public String bookList(){

URI uri = URI.create("http://localhost:8081/recommended"); return this.restTemplate.getForObject(uri, String.class);

}

public String newList(){

return "Cloud native Java";

}

}

@Service

public class BookService{

private final RestTemplate restTemplate;

public BookService(RestTemplate rest){

this.restTemplate = rest;

}

@HystrixCommand(fallbackMethod = "newList") public String bookList(){

URI uri = URI.create("http://localhost:8081/recommended"); return this.restTemplate.getForObject(uri, String.class);

}

public String newList(){

return "Cloud native Java";

}

}

Here @HystrixCommand is applied to the original method bookList(). The @HystrixCommand annotation has newList as the fallback method. So for some reason if Hystrix opens the circuit on bookList(), you will have a placeholder book list ready for the users.

Spring Framework 5Learn Spring Framework 5 with my Spring Framework 5: Beginner to Guru course!

Spring Framework DataAccess Annotations

@Transactional

This annotation is placed before an interface definition, a method on an interface, a class definition, or a public method on a class. The mere presence of @Transactional is not enough to activate the transactional behaviour. The @Transactional is simply a metadata that can be consumed by some runtime infrastructure. This infrastructure uses the metadata to configure the appropriate beans with transactional behaviour.

The annotation further supports configuration like:

The Propagation type of the transaction

The Isolation level of the transaction

A timeout for the operation wrapped by the transaction

A read only flag - a hint for the persistence provider that the transaction must be read only

The rollback rules for the transaction

Cache-Based Annotations

@Cacheable

This annotation is used on methods. The simplest way of enabling the cache behaviour for a method is to annotate it with @Cacheable and parameterize it with the name of the cache where the results would be stored.

@Cacheable("addresses")

public String getAddress(Book book){...}

1

2

@Cacheable("addresses")

public String getAddress(Book book){...}

In the snippet above , the method getAddress is associated with the cache named addresses. Each time the method is called, the cache is checked to see whether the invocation has been already executed and does not have to be repeated.

@CachePut

This annotation is used on methods. Whenever you need to update the cache without interfering the method execution, you can use the @CachePut annotation. That is, the method will always be executed and the result cached.

@CachePut("addresses")

public String getAddress(Book book){...}

1

2

@CachePut("addresses")

public String getAddress(Book book){...}

Using @CachePut and @Cacheable on the same method is strongly discouraged as the former forces the execution in order to execute a cache update, the latter causes the method execution to be skipped by using the cache.

@CacheEvict

This annotation is used on methods. It is not that you always want to populate the cache with more and more data. Sometimes you may want remove some cache data so that you can populate the cache with some fresh values. In such a case use the @CacheEvict annotation.

@CacheEvict(value="addresses", allEntries="true")

public String getAddress(Book book){...}

1

2

**@CacheEvict(value="addresses", allEntries="true")**

public String getAddress(Book book){...}

Here an additional element allEntries is used along with the cache name to be emptied. It is set to true so that it clears all values and prepares to hold new data.

**@CacheConfig**

This annotation is a class level annotation. The @CacheConfig annotation helps to streamline some of the cache information at one place. Placing this annotation on a class does not turn on any caching operation. This allows you to store the cache configuration at the class level so that you don’t have declare things multiple times.

Task Execution and Scheduling Annotations

@Scheduled

This annotation is a method level annotation. The @Scheduled annotation is used on methods along with the trigger metadata. A method with @Scheduled should have void return type and should not accept any parameters.

There are different ways of using the @Scheduled annotation:

@Scheduled(fixedDelay=5000)

public void doSomething() {

// something that should execute periodically

}

1

2

3

4

@Scheduled(fixedDelay=5000)

public void doSomething() {

// something that should execute periodically

}

In this case, the duration between the end of last execution and the start of next execution is fixed. The tasks always wait until the previous one is finished.

@Scheduled(fixedRate=5000)

public void doSomething() {

// something that should execute periodically

}

1

2

3

4

@Scheduled(fixedRate=5000)

public void doSomething() {

// something that should execute periodically

}

In this case, the beginning of the task execution does not wait for the completion of the previous execution.

@Scheduled(initialDelay=1000,fixedRate=5000)

public void doSomething() {

// something that should execute periodically after an initial delay

}

1

2

3

4

@Scheduled(initialDelay=1000,fixedRate=5000)

public void doSomething() {

// something that should execute periodically after an initial delay

}

The task gets executed initially with a delay and then continues with the specified fixed rate.

@Async

This annotation is used on methods to execute each method in a separate thread. The @Async annotation is provided on a method so that the invocation of that method will occur asynchronously. Unlike methods annotated with @Scheduled, the methods annotated with @Asynccan take arguments. They will be invoked in the normal way by callers at runtime rather than by a scheduled task.

@Async can be used with both void return type methods and the methods that return a value. However methods with return value must have a Future typed return values.

Spring Framework Testing Annotations

@BootstrapWith

This annotation is a class level annotation. The @BootstrapWith annotation is used to configure how the Spring TestContext Framework is bootstrapped. This annotation is used as a metadata to create custom composed annotations and reduce the configuration duplication in a test suite.

@ContextConfiguration

This annotation is a class level annotation that defines a metadata used to determine which configuration files to use to the load the ApplicationContext for your test. More specifically @ContextConfiguration declares the annotated classes that will be used to load the context. You can also tell Spring where to locate for the file.

@ContextConfiguration(locations={"example/test-context.xml", loader = Custom ContextLoader.class})

@WebAppConfiguration

This annotation is a class level annotation. The @WebAppConfiguration is used to declare that the ApplicationContext loaded for an integration test should be a WebApplicationContext. This annotation is used to create the web version of the application context. It is important to note that this annotation must be used with the @ContextConfiguration annotation.The default path to the root of the web application is src/main/webapp. You can override it by passing a different path to the <span class="theme:classic lang:default decode:true crayon-inline">@WebAppConfiguration.

@Timed

This annotation is used on methods. The @Timed annotation indicates that the annotated test method must finish its execution at the specified time period(in milliseconds). If the execution exceeds the specified time in the annotation, the test fails.

@Timed(millis=10000)

public void testLongRunningProcess() { ... }

1

2

@Timed(millis=10000)

public void testLongRunningProcess() { ... }

In this example, the test will fail if it exceeds 10 seconds of execution.

@Repeat

This annotation is used on test methods. If you want to run a test method several times in a row automatically, you can use the @Repeat annotation. The number of times that test method is to be executed is specified in the annotation.

@Repeat(10)

@Test

public void testProcessRepeatedly() { ... }

1

2

3

@Repeat(10)

@Test

public void testProcessRepeatedly() { ... }

In this example, the test will be executed 10 times.

@Commit

This annotation can be used as both class-level or method-level annotation. After execution of a test method, the transaction of the transactional test method can be committed using the @Commit annotation. This annotation explicitly conveys the intent of the code. When used at the class level, this annotation defines the commit for all test methods within the class. When declared as a method level annotation @Commit specifies the commit for specific test methods overriding the class level commit.

@RollBack

This annotation can be used as both class-level and method-level annotation. The @RollBack annotation indicates whether the transaction of a transactional test method must be rolled back after the test completes its execution. If this true @Rollback(true), the transaction is rolled back. Otherwise, the transaction is committed. @Commit is used instead of @RollBack(false).

When used at the class level, this annotation defines the rollback for all test methods within the class.

When declared as a method level annotation @RollBack specifies the rollback for specific test methods overriding the class level rollback semantics.

@DirtiesContext

This annotation is used as both class-level and method-level annotation. @DirtiesContext indicates that the Spring ApplicationContext has been modified or corrupted in some manner and it should be closed. This will trigger the context reloading before execution of next test. The ApplicationContext is marked as dirty before or after any such annotated method as well as before or after current test class.

The @DirtiesContext annotation supports BEFORE\_METHOD, BEFORE\_CLASS, and BEFORE\_EACH\_TEST\_METHOD modes for closing the ApplicationContext before a test.

NOTE: Avoid overusing this annotation. It is an expensive operation and if abused, it can really slow down your test suite.

@BeforeTransaction

This annotation is used to annotate void methods in the test class. @BeforeTransaction annotated methods indicate that they should be executed before any transaction starts executing. That means the method annotated with @BeforeTransaction must be executed before any method annotated with @Transactional.

@AfterTransaction

This annotation is used to annotate void methods in the test class. @AfterTransaction annotated methods indicate that they should be executed after a transaction ends for test methods. That means the method annotated with @AfterTransaction must be executed after the method annotated with @Transactional.

@Sql

This annotation can be declared on a test class or test method to run SQL scripts against a database. The @Sql annotation configures the resource path to SQL scripts that should be executed against a given database either before or after an integration test method. When @Sql is used at the method level it will override any @Sql defined in at class level.

@SqlConfig

This annotation is used along with the @Sql annotation. The @SqlConfig annotation defines the metadata that is used to determine how to parse and execute SQL scripts configured via the @Sql annotation. When used at the class-level, this annotation serves as global configuration for all SQL scripts within the test class. But when used directly with the config attribute of @Sql, @SqlConfig serves as a local configuration for SQL scripts declared.

@SqlGroup

This annotation is used on methods. The @SqlGroup annotation is a container annotation that can hold several @Sql annotations. This annotation can declare nested @Sql annotations.

In addition, @SqlGroup is used as a meta-annotation to create custom composed annotations. This annotation can also be used along with repeatable annotations, where @Sql can be declared several times on the same method or class.

@SpringBootTest

This annotation is used to start the Spring context for integration tests. This will bring up the full autoconfigruation context.

@DataJpaTest

The @DataJpaTest annotation will only provide the autoconfiguration required to test Spring Data JPA using an in-memory database such as H2.

This annotation is used instead of @SpringBootTest

@DataMongoTest

The @DataMongoTest will provide a minimal autoconfiguration and an embedded MongoDB for running integration tests with Spring Data MongoDB.

@WebMVCTest

The @WebMVCTest will bring up a mock servlet context for testing the MVC layer. Services and components are not loaded into the context. To provide these dependencies for testing, the @MockBean annotation is typically used.

@AutoConfigureMockMVC

The @AutoConfigureMockMVC annotation works very similar to the @WebMVCTest annotation, but the full Spring Boot context is started.

@MockBean

Creates and injects a Mockito Mock for the given dependency.

@JsonTest

Will limit the auto configuration of Spring Boot to components relevant to processing JSON.

This annotation will also autoconfigure an instance of JacksonTester or GsonTester.

@TestPropertySource

Class level annotation used to specify property sources for the test class.

**Java Spring Bean Lifecycle**

Spring Beans are Instantiated / Managed by Spring IoC Container. These beans can be created by providing bean specific configuration metadata to the container. Configuration Metadata can be provided in any of below formats.

* XML
* Annotation
* Java Code

The container will contain beans as long as they are required by Application. Beans created outside Spring container can also be registered with AC(Application Context).BeanFactory is the root interface for accessing the bean container. Other interfaces are also available for the specific purpose.

BeanFactory is a central registry of application components (Beans).

These components (Beans) have lifecycle interfaces and methods which will be invoked in some order before Bean can be handed over to the application and before Bean is getting destroyed.

**Bean LifeCycle:**

When a bean is initialized it might require to perform some action before it can come into a usable state (State in which application can use it) and when a bean is getting destroyed there might be some cleanup activity required for given bean. These activities are known as bean Lifecycle.

Standard bean lifecycle interfaces & the standard order of execution are given below.

1- IoC container will look for the configuration metadata of given Bean.

2- Once found, the container will create the instance of Bean (Using reflection API).

3- After instance, creation dependency will be injected (DI).

If Bean Class implements any of the below-highlighted interfaces then corresponding method will be invoked in below order (Point 4 – 13).

4- setBeanName method of BeanNameAware Interface. It sets the name of the bean in the bean factory that created this bean.

5- setBeanClassLoader method of BeanClassLoaderAware Interface. Callback that supplies the bean to a bean instance.

6- setBeanFactory method of BeanFactoryAware Interface. Callback that supplies the owning factory to a bean instance.

Below method execution will be applicable when running in an application context. (Points 7 – 11)

7- setResourceLoader method of ResourceLoaderAware Interface. It set the ResourceLoader that this object runs in.

8- setApplicationEventPublisher method of ApplicationEventPublisherAware Interface. Set the ApplicationEventPublisher that this object runs in.

9- setMessageSource method of MessageSourceAware Interface. Set the MessageSource that this object runs in.

10- setApplicationContext method of ApplicationContextAware Interface. Set the ApplicationContext that this object runs in.

11- setServletContext method of ServletContextAware Interface. Set the ServletContext that this object runs in.

12- postProcessBeforeInitialization method of BeanPostProcessor Interface. Apply this BeanPostProcessor to the given new bean instance before any bean initialization callbacks.

13- afterPropertiesSet method of InitializingBean Interface. Invoked by a BeanFactory after it has set all bean properties supplied.

In case Bean class has custom init method defined (via init-method attribute)

14- Custom init method will be invoked.

15- postProcessAfterInitialization methods of BeanPostProcessors. Apply this BeanPostProcessor to the given new bean instance after any bean initialization callbacks

When Bean Factory is getting shut down following lifecycle methods will be executed.

1- destroy method DisposableBean Invoked by a BeanFactory on the destruction of a singleton.

2- Custome destroy method will be executed if there is any defined via destroy-method attributes

**Singleton bean=** Only once per spring container

**Prototype=** New bean created with every request or reference

**Request=** New bean per servlet request

**Session =** new bean per session

**Global session=** globalSession is something which is connected to Portlet applications. When your application works in Portlet container it is built of some amount of portlets. Each portlet has its own session, but if your want to store variables global for all portlets in your application than you should store them in globalSession. This scope doesn't have any special effect different from session scope in Servlet based applications.

If parent bean in singleton and referred child beans are prototype still for the parent bean only once the child beans will be initialized.

**Spring Bean Eager vs Lazy Initialization Configurations**

By default, Spring “application context” eagerly creates and initializes all ‘singleton scoped‘ beans during application startup itself. It helps in detecting the bean configuration issues at early stage, in most of the cases. But sometimes, you may need to mark some or all beans to be lazy initialized due to different project requirements.

Spring provides two easy ways to configure lazy initialization of beans based on which kind of configuration you are employing i.e. XML based configuration or java based configuration.

Table of Contents

Lazy initialized beans in XML based configuration

Lazy initialized beans in Java based configuration

Lazy initialized beans in XML based configuration

**i) Lazy load specific beans only**

To enable lazy loading for specific beans, use lazy-init="true" attribute on bean definitions in bean configuration xml files.

**beans.xml**

<beans>

<bean id="employeeManager" class="com.howtodoinjava.spring.service.impl.EmployeeManagerImpl"

lazy-init="true"/>

<beans>

**ii) Lazy load all beans**

To enable lazy loading for all beans, use default-lazy-init="true" attribute on beans tag in bean configuration xml files.

beans.xml

<beans default-lazy-init="true">

<bean id="employeeManager" class="com.howtodoinjava.spring.service.impl.EmployeeManagerImpl" />

<beans>

Lazy initialized beans in Java based configuration

i) Lazy load specific beans only

To lazy load only specific beans, use @Lazy annotation along with @Bean annotation in java config.

**AppConfig.java**

import org.springframework.context.annotation.Lazy;

@Configuration

public class AppConfig {

@Lazy

@Bean

public EmployeeManager employeeManager() {

return new EmployeeManagerImpl();

}

}

ii) Lazy load all beans

To lazy load all beans, use @Lazy annotation along with @Bean annotation in java config.

AppConfig.java

import org.springframework.context.annotation.Lazy;

@Lazy

@Configuration

public class AppConfig {

@Bean

public EmployeeManager employeeManager() {

return new EmployeeManagerImpl();

}

}

Demo

Let’s see the code of bean, we are trying to lazy load.

public class EmployeeManagerImpl implements EmployeeManager {

@Override

public Employee create() {

Employee emp = new Employee();

emp.setId(1);

emp.setName("Lokesh");

return emp;

}

@PostConstruct

public void onInit(){

System.out.println("EmployeeManagerImpl Bean is Created !!");

}

}

I have put the @PostConstruct annotation to detect when bean is created.

Lets initialize the application context with –

**a) “Without Lazy loading” enabled**

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import com.howtodoinjava.spring.model.Employee;

import com.howtodoinjava.spring.service.EmployeeManager;

public class Main

{

public static void main( String[] args )

{

ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);

System.out.println("Bean Factory Initialized !!");

EmployeeManager empManager = ctx.getBean(EmployeeManager.class);

Employee emp = empManager.create();

System.out.println(emp);

}

}

Output:

EmployeeManagerImpl Bean is Created !!

Bean Factory Initialized !!

Employee [id=1, name=Lokesh]

Here, first bean has been created and initilized before the bean factory got fully initialized.

**b) “With Lazy loading” enabled**

Bean Factory Initialized !!

EmployeeManagerImpl Bean is Created !!

Employee [id=1, name=Lokesh]

After enable bean lazy loading, bean factory first for fully initialized. Later when we requested the EmployeeManager bean, factory then created the instance and returned it.

**@EnableDiscoveryClient  and @EnableEurekaClient**

There are multiple implementations of "Discovery Service" (eureka, [consul](https://github.com/spring-cloud/spring-cloud-consul), [zookeeper](https://github.com/spring-cloud/spring-cloud-zookeeper)).@EnableDiscoveryClient lives in [spring-cloud-commons](https://github.com/spring-cloud/spring-cloud-commons) and picks the implementation on the classpath. @EnableEurekaClient lives in [spring-cloud-netflix](https://github.com/spring-cloud/spring-cloud-netflix/) and only works for eureka. If eureka is on your classpath, they are effectively the same.

**Bean Lifecycle**

The lifecycle of a Spring bean consists the following steps:

1. Instantiation
2. Properties population
3. Call of setBeanName() method of BeanNameAware
4. Call of setBeanFactory() method of BeanFactoryAware
5. Call of setApplicationContext() of ApplicationContextAware
6. Pre-initialization with BeanPostProcessor
7. Call of afterPropertiesSet() method of InitializingBean
8. Custom init method
9. Post-initialization with BeanPostProcessor
10. Bean is ready to use
11. Call of destroy() method of DisposableBean
12. Custom destroy method

Numbers 11-12 are actual for all scopes except prototype, since Spring does not manage the complete lifecycle of a prototype bean: the container instantiates, configures, and otherwise assembles a prototype object and hands it to the client with no further record of that prototype instance.

The init() method is called when the bean is loaded to the container via the init-method attribute in the xml configuration with the @PostConstruct annotation. The destroy() method is called when the bean is unloaded from the container, through the destroy-method attribute in the xml configuration with the @PreDestroy annotation. If a bean is a prototype-scoped, the client code must clean up objects and release expensive resources that the prototype beans are holding. To get the Spring container to release resources held by prototype-scoped beans, try using a custom BeanPostProcessor, which holds a reference to beans that need to be cleaned up.

A developer can implement various interfaces to invoke specific behavior during a bean’s life cycle, such as InitializingBean and DisposableBean, as well as BeanNameAware, BeanFactoryAware and ApplicationContextAware.

**Spring Prototype Bean Scope**

**-----------------------------**

Prototype scope in the spring framework creates a new instance of a bean, every time; a request for that specific bean is made. The Prototype scope is preferred for the stateful beans, and the spring container does not manage the complete lifecycle of a prototype bean i.e. destruction lifecycle methods are uncalled. Like so, a developer is responsible for cleaning up the prototype-scoped bean instances and any resources it holds. Below snippet shows how to specify the prototype scope bean in the configuration file.

**Code Snippet**

**--------------**

1

2

<!-- Setting the bean scope to 'Prototype' -->

<bean id="id" class="com.spring.model.Bean" scope="prototype" />

But developers can define the scope of a bean using the @Scope(value= ConfigurableBeanFactory.SCOPE\_PROTOTYPE) annotation. Below snippet shows how to specify the prototype scope bean using the Java configuration.

Code Snippet

1

2

3

4

5

6

---------------------

@Component

@Scope("prototype")

public class Bean {

......

}

Always remember, to use the Prototype scope for the stateful beans and the Singleton scope for the stateless beans.

---------------------

**There are different ways through which we can autowire a spring bean**

**----------------------------------------------------------------------**

autowire byName – For this type of autowiring, setter method is used for dependency injection. Also the variable name should be same in the class where we will inject the dependency and in the spring bean configuration file.

autowire byType – For this type of autowiring, class type is used. So there should be only one bean configured for this type in the spring bean configuration file.

autowire by constructor – This is almost similar to autowire byType, the only difference is that constructor is used to inject the dependency.

autowire by autodetect – If you are on Spring 3.0 or older versions, this is one of the autowire options available. This option was used for autowire by constructor or byType, as determined by Spring container. Since we already have so many options, this option is deprecated. I will not cover this option in this tutorial.

@Autowired annotation – We can use Spring @Autowired annotation for spring bean autowiring. @Autowired annotation can be applied on variables and methods for autowiring byType. We can also use @Autowired annotation on constructor for constructor based spring autowiring.

For @Autowired annotation to work, we also need to enable annotation based configuration in spring bean configuration file. This can be done by context:annotation-config element or by defining a bean of type org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor.

@Qualifier annotation – This annotation is used to avoid conflicts in bean mapping and we need to provide the bean name that will be used for autowiring. This way we can avoid issues where multiple beans are defined for same type. This annotation usually works with the @Autowired annotation. For constructors with multiple arguments, we can use this annotation with the argument names in the method.

**Spring guarantees exactly one shared bean instance for singleton bean for the given id per IoC container**

**---------------------------------------------------------------------------------------------------------**

ie we can create more than one spring singleton bean in a container as long as as long as they have different ids(autowitred with different names)

@Autowired

BeanName mybean;

@Autowired

BeanName mybean2;

Have you ever wonder why singleton is the default scope for Spring beans? Why isn’t it prototype?

It’s not a random choice. It’s because the vast majority of business logic we create can be safely kept in stateless objects. And the best choice for stateless beans is the singleton scope. The prototype scope is better for stateful beans to avoid multithreading issues.

Stereotype is a class-level annotation denoting the roles of types or methods in the overall architecture (at a conceptual level, rather than implementation). In Spring, these annotations live in the package org.springframework.stereotype.

Currently, this package has the following annotations:

* @Component indicates that an annotated class is a “component”. Such classes are considered as candidates for auto-detection when using annotation-based configuration and classpath scanning.
* @Controller indicates that an annotated class is a “Controller” (e.g. a web controller).
* @Repository indicates that an annotated class is a “Repository”, originally defined by Domain-Driven Design (Evans, 2003) as “a mechanism for encapsulating storage, retrieval, and search behavior which emulates a collection of objects”.
* @Service indicates that an annotated class is a “Service”, originally defined by Domain-Driven Design (Evans, 2003) as “an operation offered as an interface that stands alone in the model, with no encapsulated state.” May also indicate that a class is a Business Service Facade (in the Core J2EE patterns sense) or something similar.

These different types primarily allow a developer easily distinguish the purpose of the annotated classes. Starting with Spring 2.5, @Controller, @Repository and @Service serve as a specialization of @Component, allowing for implementation classes to be autodetected through classpath scanning.

Let’s say we have a custom.properties file that defines a database connection timeout property called connection.timeout. To load this property into a Spring context, we need to define a propertyConfigurer bean:

<bean id="propertyConfigurer" class="org.springframework.context.support.PropertySourcesPlaceholderConfigurer">

<property name="location" value="custom.properties" />

</bean>

After that we can use Spring Expression Language to inject properties into other beans:

<bean class="com.toptal.spring.ConnectionFactory">

<property name="timeout" value="${connection.timeout}"/>

</bean>

The same is available in the annotation based configuration, like so:

@Value("${connection.timeout}")

private int timeout;

**@Autowired and Optional Dependencies**

Spring expects *@Autowired* dependencies to be available when the dependent bean is being constructed. If the framework cannot resolve a bean for wiring, it will throw the below-quoted exception and prevent the Spring container from launching successfully:

|  |  |
| --- | --- |
| 1  2  3  4  5 | Caused by: org.springframework.beans.factory.NoSuchBeanDefinitionException:  No qualifying bean of type [com.autowire.sample.FooDAO] found for dependency:  expected at least 1 bean which qualifies as autowire candidate for this dependency.  Dependency annotations:  {@org.springframework.beans.factory.annotation.Autowired(required=true)} |

To avoid this from happening, a bean can optional be specified as below:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class FooService {        @Autowired(required = false)      private FooDAO dataAccessor;    } |