SPRING

**Q) What is Spring?**

Spring framework is an application framework and inversion of control for Java Applications. Spring is lightweight framework. It can think of framework of frameworks because it provides support to various frameworks such as Hibernate, EJB, JSF, etc...

Spring is a framework that helps you to “WIRE” different component together. It is more than just a dependency injection. It is something which will let you build Enterprise Java Application.

**Q) 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.

**Q) 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.

## **Q)** **Spring vs. Spring Boot?**

**Spring:** The main feature of the Spring Framework is **dependency Injection** or **Inversion of Control** (IOC). With the help of Spring Framework, we can develop a **loosely** coupled application.

Aspect Oriented Programming (AOP) is another strong side of the Spring Framework.

Web application development becomes easy with the help of these concepts in Spring, like Dispatcher Servlet, ModelAndView, and View Resolver.

## **If Spring Can Solve so Many Problems, Why Do We Need Spring Boot?**

There was lot of difficulty to setup Hibernate Datasource, Entity Manager, Session Factory, and Transaction Management. (All these in XML)

## **How Does Spring Boot Solve This Problem?**

1. Spring Boot does all of those using AutoConfiguration and will take care of all the internal dependencies that your application needs.
2. During web-application development, we would need the jars that we want to use, which versions of the jars to use, and how to connect them together.  So, we had to choose the compatible versions of all these jars. In order to decrease the complexity, Spring Boot has introduced what we call **Spring Boot Starters. (Starters** are a set of convenient dependencies that you can include in your Spring Boot application.**)**

**Example: In spring boot we just include spring-boot-starter-web for dependency in spring like spring-webmvc, Jackson-databind, hibernate-validator and log4j.**

|  |  |
| --- | --- |
| **Spring** | **Spring Boot** |
| **Spring Framework** is a widely used Java EE framework for building applications. | **Spring Boot Framework** is widely used to develop **REST APIs**. |
| It aims to simplify Java EE development that makes developers more productive. | It aims to shorten the code length and provide the easiest way  to develop **Web Applications**. |
| The primary feature of the Spring Framework is **dependency injection**. | The primary feature of Spring Boot is **Autoconfiguration**.  It automatically configures the classes based on the requirement. |
| It helps to make things simpler by allowing us to develop **loosely coupled** applications. | It helps to create a **stand-alone** application with less configuration. |
| The developer writes a lot of code (**boilerplate code**) to do the minimal task. | It **reduces** boilerplate code. |
| To test the Spring project, we need to set up the server explicitly. | Spring Boot offers **embedded server** such as **Jetty** and **Tomcat**, etc. |
| It does not provide support for an in-memory database. | It offers several plugins for working with an embedded and  **in-memory** database such as **H2**. |

### **Q) What is Inversion of Control?**

### Inversion of Control is a principle in software engineering by which the control of objects or portions of a program is transferred to a container or framework.

### Inversion of Control can be achieved through various mechanisms such as: Strategy design pattern, Service Locator pattern, Factory pattern, and Dependency Injection (DI).

### **Q) What is Spring IOC Container?**

## ioc - Spring Interview Questions - Edureka!

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.

There are basically two types of IOC Containers in Spring:

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

**Bean Factory Application Context**

|  |  |
| --- | --- |
| Bean instantiation/wiring | Bean instantiation/wiring |
| BeanFactory**loads beans on-demand (Lazy Loading)** | ApplicationContext**loads all beans at startup** (Eager Loading) |
| Only supports two scopes — Singleton and Prototype. | Supports almost all types of bean scopes. |
|  | Automatic BeanPostProcessor registration |
|  | Automatic BeanFactoryPostProcessor registration |
|  | Convenient MessageSource access (for i18n) |
|  | Event publication functionality |
|  | **Easy integration with Spring AOP features** |
|  | Ability to resolve textual message from “properties file” |

Spring provides two kinds of IOC container, one is XMLBeanFactory and other is ApplicationContext.

+-----------------------------------------+ ------------------- +------------------------------------------------------------------+

| | BeanFactory | ApplicationContext |

+--------------------------------------- + ----------------- +------------------------------------------------------------------+

| Annotation support | No | Yes |

|BeanPostProcessor Registration | Manual | Automatic |

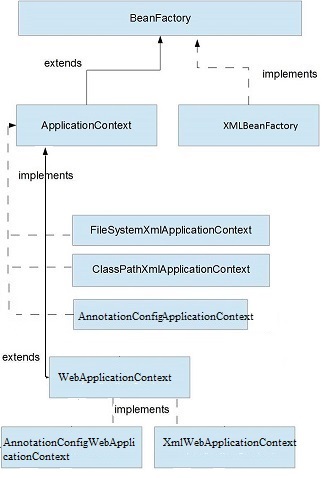
| implementation | XMLBeanFactory| ClassPath/FileSystem/WebXmlApplicationContext |

| internationalization | No | Yes |

| Enterprise services | No | Yes |

| ApplicationEvent publication | No | Yes |

+---------------------------------------+-----------------+-------------------------------------------------------------------------+

[](https://i.stack.imgur.com/EweA3.jpg)

* FileSystemXmlApplicationContext Beans loaded through the full path.
* ClassPathXmlApplicationContext Beans loaded through the CLASSPATH
* AnnotationConfigApplicationContext Loading Spring beans from Annotation based configuration.
* XMLWebApplicationContext and AnnotationConfigWebApplicationContext beans loaded through the web application context.

**Example**:

ApplicationContext applicationContext = new AnnotationConfigApplicationContext(BeansConfiguration.class);

ApplicationContext is the container initialized by a ContextLoaderListener or ContextLoaderServlet defined in a web.xml and ContextLoaderPlugin defined in struts-config.xml

## **Q) Dependency Injection (DI)?**

Dependency injection is a pattern through which to implement IOC (means IOC uses or implement dependency injection pattern  Dependency injection is a pattern used to create instances of objects that other objects rely on without knowing at compile time which class will be used to provide that functionality. IOC relies on dependency injection because a mechanism is needed in order to activate the components providing the specific functionality.

Java components / classes should be as independent as possible of other Java classes. This increases the possibility to reuse these classes and to test them independently of other classes (Unit Testing). To decouple Java components from other Java components the dependency to a certain other class should get injected into them rather that the class itself creates / finds this object. A class A has a dependency to class B if class uses class B as a variable.

If dependency injection is used, then the class B is given to class A via

* the constructor of the class A - this is then called construction injection
* a setter - this is then called setter injection

The general concept between dependency injection is called Inversion of Control. A class should not configure itself but should be configured from outside. A design based on independent classes / components increases the re-usability and possibility to test the software. For example, if a class A expects a Dao (Data Access object) for receiving the data from a database you can easily create another test object which mocks the database connection and inject this object into A to test A without having an actual database connection.

**Why do we need dependency injection in Spring?**

* **Spring** provides a light-weight container, e.g. the **Spring** core container, for **dependency injection** (DI). This container lets you **inject** required objects into other objects. This results in a design in which the Java class are not hard coupled
* **DI** makes our programming code loosely coupled.

**Two ways to perform dependency injection**

1. **By Constructor**
2. **By Setter Method**

# **Difference between constructor and setter injection**

# There are many key differences between constructor injection and setter injection.

1. **Partial dependency**: can be injected using setter injection but it is not possible by constructor. Suppose there are 3 properties in a class, having 3 arg constructor and setters’ methods. In such case, if you want to pass information for only one property, it is possible by setter method only.
2. **Overriding**: Setter injection overrides the constructor injection. If we use both constructor and setter injection, IOC container will use the setter injection.
3. **Changes**: We can easily change the value by setter injection. It doesn't create a new bean instance always like constructor. So, setter injection is flexible than constructor injection.

**Q) What is @autowired in spring?**

Autowiring feature of spring framework enables you to inject the object dependency implicitly.

### @Autowired will tell Spring to search for a Spring bean which implements the required interface and place it automatically into the setter. This annotation allows Spring to resolve and inject collaborating beans into your bean. Autowiring can't be used to inject primitive and string values. It works with reference only.

### 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 many parameters.
5. **autodetect:**First the container tries to wire using autowire by *constructor*, if it can’t then it tries to autowire by *byType*.

### **Autowiring can be used on properties, setters, and constructors.**

### **1)@Autowired on Properties: -**

@Component("fooFormatter")

public class FooFormatter {

    public String format () {

        return "foo";

    }

}

@Component

public class FooService {

    @Autowired

    private FooFormatter fooFormatter;

}

### **2)@Autowired on Setters**

The @Autowired annotation can be used on setter methods. In the below example, when the annotation is used on the setter method, the setter method is called with the instance of FooFormatter when FooService is created:

public class FooService {

    private FooFormatter fooFormatter;

    @Autowired

    public void setFooFormatter(FooFormatter fooFormatter) {

            this.fooFormatter = fooFormatter;

    }

}

### **3)@Autowired on Constructors**

The @Autowired annotation can also be used on constructors. In the below example, when the annotation is used on a constructor, an instance of FooFormatter is injected as an argument to the constructor when FooService is created:

public class FooService {

    private FooFormatter fooFormatter;

    @Autowired

    public FooService(FooFormatter fooFormatter) {

        this.fooFormatter = fooFormatter;

    }

}

## **4)@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:

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.

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

public class FooService {

    @Autowired(required = false)

    private FooDAO dataAccessor;

}

### **5)Autowiring by @Qualifier**

The @Qualifier annotation can be used to hint at and narrow down the required bean:

@Component("fooFormatter")

public class FooFormatter implements Formatter {

    public String format () {

        return "foo";

    }

}

@Component("barFormatter")

public class BarFormatter implements Formatter {

    public String format () {

        return "bar";

    }

}

public class FooService {

    @Autowired

    private Formatter formatter;

}

### **Since there are two concrete implementations of Formatter available for the Spring container to inject,** **Spring will throw a NoUniqueBeanDefinitionException exception when constructing the FooService:**

### This can be avoided by narrowing the implementation using a @Qualifier annotation:

public class FooService {

    @Autowired

    @Qualifier("fooFormatter")

    private Formatter formatter;

}

By specifying the @Qualifier with the name of the specific implementation, in this case as fooFormatter, we can avoid ambiguity when Spring finds multiple beans of the same type.

Please note that the value of the @Qualifier annotation matches with the name declared in the @Component annotation of our FooFormatter implementation.

### **6)Autowiring by Custom Qualifier**

Spring allows us to create our own @Qualifier annotation. To create a custom Qualifier, define an annotation and provide the @Qualifier annotation within the definition as below:

@Qualifier

@Target ({ElementType.FIELD, ElementType.METHOD, ElementType.TYPE, ElementType.PARAMETER})

@Retention (RetentionPolicy.RUNTIME)

public @interface FormatterType {

    String value ();

}

### Once defined, the FormatterType can be used within various implementations to specify custom value:

@FormatterType("Foo")

@Component

public class FooFormatter implements Formatter {

    public String format () {

        return "foo";

    }

}

@FormatterType("Bar")

@Component

public class BarFormatter implements Formatter {

    public String format () {

        return "bar";

    }

}

### Once the implementations are annotated, the custom Qualifier annotation can be used as below:

@Component

public class FooService {

    @Autowired

    @FormatterType("Foo")

    private Formatter formatter;

}

The value specified in the @Target annotation restrict where the qualifier can be used to mark injection points.

In the above code snippet, the qualifier can be used to disambiguate the point where Spring can inject the bean into a field, a method, a type, and a parameter.

### **7)Autowiring by Name**

As a fallback Spring uses the bean name as a default qualifier value.

So, by defining the bean property name, in this case as fooFormatter, Spring matches that to the FooFormatter implementation and injects that specific implementation when FooService is constructed:

public class FooService {

    @Autowired

    private Formatter fooFormatter;

}

### **NOTE: Although both @Qualifier and bean name fallback match can be used to narrow down to a specific bean, autowiring is really all about injection by type and this is how best to use this container feature.**

### **Q) What are the limitations with autowiring?**

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.

**Q) @Primary?**

**we use @Primary to give higher preference to a bean when there are multiple beans of the same type.**

In some cases, **we need to register more than one bean of the same type**.

In this example we have JohnEmployee() and TonyEmployee() beans of the Employee type:

@Configuration

public class Config {

    @Bean

    public Employee JohnEmployee() {

        return new Employee("John");

    }

  @Bean

    public Employee TonyEmployee() {

        return new Employee("Tony");

    }

}

**Spring throws NoUniqueBeanDefinitionException if we try to run the application.**

To access beans with the same type we usually use @Qualifier(“beanName”) annotation.

We apply it at the injection point along with @Autowired.

In our case, we select the beans at the configuration phase so @Qualifier can't be applied here. To resolve this issue Spring offers the @Primary annotation.

## **a) Use**@Primary **With**@Bean

Let's have a look at configuration class:

@Configuration

public class Config {

    @Bean

    public Employee JohnEmployee() {

        return new Employee("John");

    }

    @Bean

    @Primary

    public Employee TonyEmployee() {

        return new Employee("Tony");

    }

}

**We mark TonyEmployee() bean with @Primary. Spring will inject TonyEmployee() bean preferentially over the JohnEmployee().**

Now, let's start the application context and get the Employee bean from it:

AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(Config.class);

Employee employee = context.getBean(Employee.class);

System.out.println(employee);

After we run the application:

**Output**: Employee{name='Tony'}

## **b) Use**@Primary**With**@Component

**We can use @Primary directly on the beans**. Let's have a look at the following scenario:

public interface Manager {

    String getManagerName();

}

We have a Manager interface and two subclass beans, DepartmentManager:

@Component

public class DepartmentManager implements Manager {

    @Override

     public String getManagerName() {

        return "Department manager";

    }

}

And the GeneralManager bean:

@Component

@Primary

public class GeneralManager implements Manager {

    @Override

    public String getManagerName() {

        return "General manager";

    }

}

They both override the getManagerName() of the Manager interface. Also, note that we mark the GeneralManager bean with @Primary.

This time, **@Primary only makes sense when we enable the component scan**:

@Configuration

@ComponentScan(basePackages="org.baeldung.primary")

public class Config {

}

Let's create a service to use dependency injection while finding the right bean: @Service

public class ManagerService {

    @Autowired

    private Manager manager;

    public Manager getManager() {

        return manager;

    }

}

Here, both beans DepartmentManager and GeneralManager are eligible for autowiring.

**As we marked GeneralManager bean with @Primary, it will be selected for dependency injection**:

ManagerService service = context.getBean(ManagerService.class);

Manager manager = service.getManager();

System.out.println(manager.getManagerName());

**The output is**: “General manager”.

**Q) Bean and @bean?**

* **Bean**: Any normal java class that is initialized by spring IOC container is called spring bean. ApplicationContext is used to get the spring bean instance.
* @**bean**: Spring @Bean Annotation is applied on a method to specify that it returns a bean to be managed by Spring context. Spring Bean annotation is usually declared in Configuration classes methods. In this case, bean methods may reference other @Bean methods in the same class by calling them directly.

## Spring Bean Name

We can specify the @Bean name and use it to get them from spring context. Let’s say we have MyFileSystemBean class defined as:

Spring Bean Name

We can specify the @Bean name and use it to get them from spring context. Let’s say we have MyFileSystemBean class defined as:

package com.journaldev.spring;

public class MyFileSystemBean

{

@Override

public String toString() {

return "MyFileSystemBean"+this.hashCode();

}

public void init() {

System.out.println("init method called");

}

public void destroy () {

System.out.println("destroy method called");

}

}

**Now define a @Bean method in the configuration class:**

@Bean (name= {"getMyFileSystemBean","MyFileSystemBean"})

public MyFileSystemBean getMyFileSystemBean() {

return new MyFileSystemBean();

}

**We can get this bean from context by using the bean name.**

MyFileSystemBean myFileSystemBean = (MyFileSystemBean) context.getBean("getMyFileSystemBean");

MyFileSystemBean myFileSystemBean1 = (MyFileSystemBean) context.getBean("MyFileSystemBean");

**NOTE:** While defining bean we have options of declaring a “**SCOPE**”.

**Q) SCOPE in spring?**

In Spring, scope can be defined using spring bean **@Scope** annotation.

The Spring Framework supports five scopes. They are:

* **Singleton:**This provides scope for the bean definition to single instance per Spring IOC container.
* **Prototype:**Opposite to singleton, it produces a new instance every time a bean is requested.
* **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 a Global HTTP-session.

NOTE:

Request: new bean creation per object (if in scope of that request then will get same bean for that request)

Session: new bean creation per user (if user does not log out then will get same object for that user)

@SpringBootApplication

public class DemoApplication

{

public static void main (String [] args)

{

ConfigurableApplicationContext context = SpringApplication.run(DemoApplication.class, args);

/\* This run method return the object of ConfigurableApplicationContext(IOC container).this line will execute it will create a container and inside the container all the objects which is annoated with @Componenet/@Controller/@Service etc\*/

Alien a= context.getBean(Alien.class);

/\*These spring beans will be created according to the scopes specified (default scope is Singleton). So as soon as this will run, spring will create one object of Alien.class as it is singleton by Default. It is just available in IOC and the moment we set getBean(Alien.class),it will be returned\*/

Alien a1= context.getBean(Alien.class);

/\*By doing this, new object will **not** create because by default sf using a concept of singleton. We can change the scopes like below. \*/

}

}

@Componenet

@Scope("prototype")

/\*This mean that it will not create object by default for you in container, the moment you write getBean() only then it will create . Now every time you use getBean() it will create new object for you. \*/

Class Alien {

Private int aid;

Private String aname;

Private String tech;

------

Setters/ Getters

------

}

**Request:** New Bean will be created per servlet/JSP requests. Means after submitting the value from the page, in entire code flow IOC will create new object for every @Autowired whether it is already created or not. In request scope, container creates a new instance for each HTTP request. So, if server is currently handling 50 requests, then container can have at most 50 individual instances of bean class. Any state change to one instance, will not be visible to other instances. These instances are destructed as soon as the request is completed.

@Component

@Scope("request")

public class BeanClass {

}

**or**

@Component

@RequestScope

public class BeanClass {

}

**Session:** In session scope, container creates a new instance for each HTTP session. So, if server has 20 active sessions, then container can have at most 20 individual instances of bean class. All HTTP requests within single session lifetime will have access to same single bean instance in that session scope.

Any state change to one instance, will not be visible to other instances. These instances are destructed as soon as the session is destroyed/end on server.

Java config example of session bean scope –

|  |
| --- |
| @Component  @Scope("session")  public class BeanClass {  }  **Or**  @Component  @SessionScope  public class BeanClass {  } |

**globalSession**: globalSession is something which is connected to Portlet applications. When your application works in Portlet container it is built of some number of portlets.

### **Q) Does Spring Bean provide thread safety?**

The default scope of Spring bean is singleton, so there will be only one instance per context. That means that all the having a class level variable that any thread can update will lead to inconsistent data. Hence in default mode spring beans are not thread safe.

However, we can change spring bean scope to request, prototype or session to achieve thread-safety at the cost of performance. It’s a design decision and based on the project requirements.

### **Q) 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 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 setApplicationContext(ApplicationContext context) by passing an instance of itself, if the bean implements the ApplicationContextAware 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.

Note:

* BeanNameAware**makes the object aware of the bean name defined in the container.**
* ApplicationContextAware**is used to inject the**ApplicationContext**object**. This way we get access to the ApplicationContext which created the object. (We override ApplicationContextAware interface and override its method setApplicationContext(Applicationcontext context) to get container instance, which we can use to get object from container).

A BeanPostProcessors allows for custom modification of new bean instances created by spring bean factory. If you want to implement some custom logic after the Spring container finishes instantiating, configuring, and initializing a bean, we can plug in one or more BeanPostProcessor implementations. BeanPostProcessor interface consists of exactly two callback methods i.e. postProcessBeforeInitialization() and postProcessAfterInitialization(). An ApplicationContext automatically detects any beans which implement the BeanPostProcessor interface.

To understand it in better way, check the below diagram:



**Q) Difference b/w JPA and Hibernate?**

A JPA (Java Persistence API) is a specification of Java which is used to access, manage, and persist data between Java object and relational database. It is just a specification. Various ORM tools implement it for data persistence. JPA can be seen as a bridge between object-oriented domain models and relational database systems. Being a specification, JPA doesn't perform any operation by itself. Thus, it requires implementation. So, ORM tools like Hibernate, TopLink, and iBatis implements JPA specifications for data persistence.

**Hibernate** is an Object-Relational Mapping (ORM) tool which is used to save the state of Java object into the database.

## **Need of JPA**

As we have seen so far, JPA is a specification. It provides common prototype and functionality to ORM tools. By implementing the same specification, all ORM tools (like Hibernate, TopLink, iBatis) follows the common standards. In the future, if we want to switch our application from one ORM tool to another, we can do it easily.

**Spring Data JPA** is a JPA Data Access Abstraction. **Spring Data JPA** API provides JpaTemplate class to integrate **spring** application with **JPA.**

|  |  |
| --- | --- |
| **JPA** | **Hibernate** |
| It is just a specification. Various ORM tools implement it for data persistence. | It is one of the most frequently used JPA implementation. |
| It is defined in **javax.persistence** package. | It is defined in **org.hibernate** package. |
| The **EntityManagerFactory** interface is used to interact with the entity manager factory for the persistence unit. Thus, it provides an entity manager. | It uses **SessionFactory** interface to create Session  instances. |
| It uses **EntityManager** interface to create, read, and delete operations for instances of mapped entity classes. This interface interacts with the persistence context. | It uses **Session** interface to create, read, and delete  operations for instances of mapped entity classes.  It behaves as a runtime interface between a Java application  and Hibernate. |
| It uses **Java Persistence Query Language** (JPQL) as an object-oriented query language to perform database operations. | It uses **Hibernate Query Language** (HQL) as an  object-oriented query language to perform database  operations. |

**Q) Spring Boot Starters**

Spring Boot Starters are a set of convenient dependency descriptors that you can include in your application. When building RESTful Web Services, we often need to add additional functionality to our project. For example, our application might need to work with a database or be able to cache data or work with LDAP or MongoDB. For our application be able to work with LDAP or MongoDB there are third-party libraries which we will need to add to a pom.xml file of our project. Spring Boot simplifies this work for us by providing Spring Boot Starters by adding which we bring into our project needed libraries. For example, if we needed to create a Spring Boot project with support for RESTful Web Services, there is a starter for that. If we need to add caching support to our project, there is also a starter for that. Or we need to make our Spring Boot application persist data into a database using Spring Data JPA with Hibernate, there is also a started for that.

|  |  |
| --- | --- |
| **Name** | **Description** |
| spring-boot-starter-web-services | It is used for Spring Web Services. |
| spring-boot-starter-mail | It is used to support Java Mail and Spring Framework's email sending. |
| spring-boot-starter-web | It is used for building the web application, including RESTful applications using  Spring MVC. It uses Tomcat as the default embedded tomcat as the default  embedded container. |
| spring-boot-starter-data-elasticsearch | It is used in Elasticsearch search and analytics engine and Spring Data Elasticsearch. |
| spring-boot-starter-test | It is used to test Spring Boot applications with libraries, including JUnit, Hamcrest,  and Mockito. |
| spring-boot-starter-jdbc | It is used for JDBC with the Tomcat JDBC connection pool. |
| spring-boot-starter-validation | It is used for Java Bean Validation with Hibernate Validator. |

**Q) Spring Boot Actuator?**

Spring Boot Actuator Endpoints are exposed over [JMX](https://www.journaldev.com/1352/what-is-jmx-mbean-jconsole-tutorial) and HTTP, most of the times we use HTTP based Actuator endpoints because they are easy to access over the browser, CURL command, shell scripts etc.

Some of the useful actuator endpoints are:

* beans: this endpoint returns the list of all the beans configured in our application.
* env: provides information about the Spring Environment properties.
* health: Shows application health
* info: Displays application information, we can configure this in Spring environment properties.
* mappings: Displays the list of all @RequestMapping paths.
* shutdown: allows us to gracefully shutdown the application.

**Q) POJO and JAVA Bean (Not spring bean)?**

1. Normal Class: A Java class
2. Java Beans:
   * All properties private (use getters/setters)
   * A public no-argument constructor
   * Implements Serializable.
3. POJO: Plain Old Java Object is a Java object not bound by any restriction other than those forced by the Java Language Specification. i.e., a POJO should not have to
   * Extend prespecified classes
   * Implement prespecified interface
   * Contain prespecified annotations

**Beans are special type of POJO. There are some restrictions on POJO to be a bean.**

1. All JavaBeans are POJOs but not all POJOs are JavaBeans.
2. Serializable i.e. they should implement Serializable interface. Still some POJOs who don’t implement Serializable interface are called POJOs because Serializable is a marker interface and therefore not of much burden.
3. Fields should be private. This is to provide the complete control on fields.
4. Fields should have getters or setters or both.
5. A no-argument constructor should be there in a bean.
6. Fields are accessed only by constructor or getter setters.

## **Q) What is spring-boot-starter-parent dependency?**

The spring-boot-starter-parent dependency is the parent POM providing dependency and plugin management for Spring Boot-based applications. It contains the default versions of Java to use, the default versions of dependencies that Spring Boot uses, and the default configuration of the Maven plugins.

It allows you manage following things:

* Configuration
* Dependency Management
* Default Plugin Configuration

**Q) @SpringBootApplication?**

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

It does a lot of thing such as:

* Sets up default configuration
* Starts the SpringApplicationContext (IOC)
* Starts Tomcat Server

**Q) Session?**

* Session simply means an interval of time.
* Session Tracking is a way to maintain state (data) of a user. It is also known as session management in servlet.
* Http protocol is a stateless, so we need to maintain state using session tracking techniques. Each time user requests to the server, server treats the request as the new request. So, we need to maintain the state of a user to recognize to user.

**//index.html** (When we click on submit the request will got add. Then add will call AddServlet in doGet() will execute from internal execute method. If we want any other method, then we will have to override execute method and from that execute method we will call desired method)

<html>

<Body>

<form action="add" method="get">

Enter 1st number: <input type="text" name="num1"><br>

Enter 2nd number: <input type="text" name="num2"><br>

<input type="Submit">

</form>

</Body>

</html>

**//AddServlet.class**

public class AddServlet

{

public void doGet (HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException

{

int i= Integer.parseInt(req.getParameter("num1"));

int j= Integer.parseInt(req.getParameter("num2"));

int k=i+j;

HttpSession session=req.getSession(); //Session will be given by tomcat we just have to get the hold on it.

session.setAttribute("K",K); //We have set the K value here, now access the value in SqServlet.

//res.sendRedirect("sq"); //Here by this line will send signal to client that send the signal to "sq" servlet.

}

}

/\*This SqServlet will try to find out square of a number, but which number we have to send this k. for that one way is to pass k res.sendRedirect("sq") like : res.sendRedirect("sq?k=8"). but what if we have multiple value. We can use Session here as Session will be there throughout your visit there, if you visit 1-Servlet, 2-Servlet,3-Servlet it will be there for you. There is concept of session that whenever you go to web application, it will try to maintain a session for you and that will be done by Tomcat. \*/

public class SqServlet extends HttpServlet

{

public void doGet(HttpServletRequest req, HttpServletResponse res) throws IOException

{

//int k=Integer.parseInt(req.getParameter("k")); //We will not use this to recieve value of K.

HttpSession session=req.getSession(); //This is the way we will get hold on session object.

int k = (int)session.getAttribute("K"); /\*getAttribute("K") will return object, but we do not want object we want int that ia why we type cast it.\*/

k = k\*k ;

PrintWriter out=res.getWriter();

System.out.println("Result is"+k);

System.out.println("sq called");

}

}

**Q) Spring MVC @ControllerAdvice Annotation?**

If we want to centralize the exception handling logic to one class which is capable to handle exceptions thrown from any handler class/Controller class – then we can use @ControllerAdvice annotation. By default the methods in an **@ControllerAdvice** apply globally to all Controllers.

The classes annotated with @ControllerAdvice are auto detected by classpath scanning. The use of @ControllerAdvice is advising all or selected controllers for @ExceptionHandler, @InitBinder and @ModelAttribute.

What we must do is create a class annotated with @ControllerAdvice and create required method which will be annotated with @ExceptionHandler for global exception handling, @InitBinder for global init binding and @ModelAttribute for global model attributes addition.

Whenever a request comes to controller and its method with @RequestMapping and if there is not locally defined @ExceptionHandler, @InitBinder and @ModelAttribute, then the globally defined class annotated with @ControllerAdvice is served.

(NOTE: If we declare any local handler with specific Exception then every time a controller encounter that exception in request processing for any web request in this controller, then control will automatically come to this handler method.

@ExceptionHandler({NullPointerException.class, ArrayIndexOutOfBoundsException.class, IOException.class})

public ModelAndView handleException(NullPointerException ex)

{

    ModelAndView modelAndView = new ModelAndView();

    modelAndView.setViewName("error");

    modelAndView.addObject("message", ex.getMessage());

    return modelAndView;

}

).

@ControllerAdvice has attributes annotations, assignableTypes and basePackages.

To use @ControllerAdvice, we should use @EnableWebMvc annotation in our java config.

**//GlobalControllerAdvice.java**

@ControllerAdvice(basePackages = {"com.concretepage.controller"} )

public class GlobalControllerAdvice

{

@InitBinder

public void dataBinding(WebDataBinder binder)

{

SimpleDateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy");

dateFormat.setLenient(false);

binder.registerCustomEditor(Date.class, "dob", new CustomDateEditor(dateFormat, true));

}

@ModelAttribute

public void globalAttributes(Model model)

{

model.addAttribute("msg", "Welcome to My World!");

}

@ExceptionHandler(FileNotFoundException.class)

public ModelAndView myError(Exception exception)

{

ModelAndView mav = new ModelAndView();

mav.addObject("exception", exception);

mav.setViewName("error");

return mav;

}

}

**Create Controller**

Find the controller with a @RequestMapping method for the demo. Here we have also defined a controller specific @InitBinder method.

**//MyWorldController.java**

@Controller

@RequestMapping("/myworld")

public class MyWorldController

{

@Autowired

private UserValidator userValidator;

@RequestMapping(value="signup", method = RequestMethod.GET)

public ModelAndView user ()

{

return new ModelAndView("user","user",new User());

}

@InitBinder

public void dataBinding(WebDataBinder binder)

{

binder.addValidators(userValidator);

}

@RequestMapping(value="save", method = RequestMethod.POST)

public String createUser(@ModelAttribute("user") @Valid User user, BindingResult result, ModelMap model) throws FileNotFoundException

{

if(result.hasErrors())

{

return "user";

}

if(user.getName().equals("exception"))

{

throw new FileNotFoundException("Error found.");

}

System.out.println("Name:"+ user.getName());

System.out.println("Date of Birth:"+ user.getDob());

return "success";

}

}

The exception FileNotFoundException thrown in the controller is not being handled, so it will be handled by global exception handling class that is annotated by @ControllerAdvice.

For the date validation,we have not local WebDataBinder configuration,it will be handled by global @InitBinder method. @ModelAttribute for "msg" key is not added in the controller locally. So, the method annotated by @ModelAttribute in

the GlobalControllerAdvice class is used.

**Q) How to Use @Initbinder in Spring MVC.**

**The Problem: -**

If the user inserts some blank spaces for a particular value and submits the form, the validator block will not treat it as Null as spaces are present, Hence the system will take the value as spaces and the subsequent processing will be done. But we do not want a value with spaces to be present in our Database. That will be a total disaster.

We will add @Initbinder annotated method to the controller, to add a initbinder method we have to declare a method with @initbinder annotation, this method should have WebDatabinder as parameter.

So, in our example we will add-

@InitBinder

public void initBinder(WebDataBinder dataBinder)

{

StringTrimmerEditor stringTrimmerEditor = new StringTrimmerEditor(true);

dataBinder.registerCustomEditor(String.class, stringTrimmerEditor);

}

We will register StringTrimmerEditor as a custom editor to the databinder with String class as target source.StringTrimmerEditor is a PropertyEditor which we are using in our demo that trims the string values. After registering the custom editor, initbinder will trim all the String values coming as part of request.

Hence, now if the Name is sent as blank white spaces, Our application will not allow the values to pass through the validator system,avoiding a leak in the system.

NOTE: @Initbinder filter the fields which is annotated with the @Not Null, etc..

**Conclusion**

This was a use case where initbinder was used as a preprocessor to trim the string values in the request before reaching the controller. Similarly, many more operations like converting the String object to a Date object, adding some extra values to the input can be done.

**Q) Task scheduling?**

Spring provides excellent support for both task scheduling and asynchronous method execution based on [cron expression](https://en.wikipedia.org/wiki/Cron) using @Scheduled annotation

@Scheduled annotation is used for task scheduling. The trigger information needs to be provided along with this annotation.

#### fixedDelay vs fixedRate vs cron

1. **fixedRate** makes Spring run the task on periodic intervals even if the last invocation may be still running.
2. **fixedDelay** specifically controls the next execution time when the last execution finishes.
3. **cron** is a feature originating from Unix cron utility and has various options based on your requirements.

Example usage can be as below:

|  |
| --- |
|  |
| @Scheduled (fixedDelay =30000)  public void demoServiceMethod () {...}    @Scheduled (fixedRate=30000)  public void demoServiceMethod () {...}    @Scheduled (cron="0 0 \* \* \* \*")  public void demoServiceMethod () {...} |

**Q) Query ()?**

**1.**

Use Native=true to use oracle query (In this query will have to use db table/column names otherwise use pojo class names and fields)

**@Query (value = "SELECT \* FROM todos t where t.title = ?0 AND t.description = ?1", nativeQuery=true)**

**public Optional<Todo> findByTitleAndDescription(String title, String description);**

**2.**

If we *want* that our query *method* is executed asynchronously, we *must* annotate it with the @Async annotation *and* return a Future<T> object. like:

**@Async**

**@Query ("SELECT t.title FROM Todo t where t.id = :id")**

**Future<String> findTitleById(@Param("id") Long id);**

**3.**

The *position*-based parameter binding means that *the* order of our method parameters *decides* which placeholders are replaced with *them*.

**@Query ("SELECT t FROM Todo t where t.title = ?1 AND t.description = ?2")**

**public Optional<Todo> findByTitleAndDescription(String title, String description);**

**4. Simple @Query is recommended over nativequery. With this we have to use pojo fields values.**

We *can* use named parameters by replacing the numeric placeholders found from our *database* queries with concrete parameter names and *annotating* our method parameters with the @Param annotation.

**@Query ("SELECT t FROM Todo t where t.title = :title AND t.description = :description")**

**public Optional<Todo> findByTitleAndDescription(@Param("title") String title, @Param("description") String description);**

**5. For Insert Query:**

In JPA, every entity going from a transient to managed state is automatically handled by the [*EntityManager*](https://www.baeldung.com/hibernate-entitymanager).

The *EntityManager* checks whether a given entity already exists and then decides if it should be inserted or updated. Because of this automatic management, **the only statements allowed by JPA are SELECT, UPDATE and DELETE.**

**So, the way to** managing and bypassing this limitation is:

Additionally, we'll apply the [@Transactional](https://www.baeldung.com/transaction-configuration-with-jpa-and-spring) annotation to handle transactions automatically by Spring. This way, we won't have to worry about creating transactions with our EntityManager, committing our changes, or performing rollback manually in the case of an exception.

For manually created queries, we can use the EntityManager.createNativeQuery () method. It allows us to create any type of SQL query, not only ones supported by JPA. Let's add a new method to our repository class:

@Override

@Transactional

**public** **void** saveCustomerAccount(Account account) {

account.setUserRole("CUSTOMER");

account.setUserRole("CUSTOMER");

entityManager.createNativeQuery("insert into Accounts (User\_Name, Active, Password, User\_Role, Name, Phone, Email) values (?,?,?,?,?,?,?)")

.setParameter(1, account.getUserName())

.setParameter(2, **true**)

.setParameter(3, account.getPassword())

.setParameter(4, account.getUserRole())

.setParameter(5, account.getName())

.setParameter(6, account.getPhone())

.setParameter(7, account.getEmail())

.executeUpdate();

entityManager.flush();

}

**Q) Lazy Loading and Eager Loading?**

With Eager Loading, all the data is retrieved in a single query, which can then be cached to improve the Application performance. With Eager Loading, we are trading memory consumption for the database round trips. With Lazy Loading, we only retrieve just the amount of data, which we need in a single query.

**In Spring:**

It is called as eager or aggressive loading because of loading all the bean, instance instantiated and initializing at the container start-up only. ... In the case of lazy loading all the beans will be loaded, instantiated and initialized when the container try to use them by calling getBeans() method.

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.

**1. @Lazy load specific bean**

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();

}

}

**2. 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();

}

}

**3. @Autowired lazy beans**

Generally, beans are injected into other components using @Autowired annotation. In this case, we must use the lazy annotation at both places:

The bean definition which you want to lazy load. The place it is injected along with @Autowired annotation

@Lazy

@Service

public class EmployeeManagerImpl implements EmployeeManager

{

}

@Controller

public class EmployeeController

{

@Lazy

@Autowired

EmployeeManager employeeManager;

}

**4. Lazy load all beans globally**

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

**5. Spring lazy loading demo**

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

@Lazy

@Service

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.

Let’s initialize the application context with –

**6. Without Lazy loading**

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);

}

}

**Program output.**

Console

EmployeeManagerImpl Bean is Created!!

Bean Factory Initialized!!

Employee [id=1, name=Lokesh]

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

With Lazy loading

Console

Bean Factory Initialized!!

EmployeeManagerImpl Bean is Created!!

Employee [id=1, name=Lokesh]

**Q) FILTER?**

Filters as the name suggest used to perform filtering on either the request to a resource or on the response from a resource, or both. Spring Boot provides few options to register custom filters in the Spring Boot application.

In order to create a filter, we need to simply implement the Filter interface:

**A)**

@Component

@Order (1)

public class TransactionFilter implements Filter

{

@Override

public void init(FilterConfig filterConfig) throws ServletException

{

LOGGER.info ("########## Initiating Custom filter ##########");

}

@Override

public void doFilter(ServletRequest request,ServletResponse response,FilterChain chain)throws IOException,Servl ServletException

{

HttpServletRequest req = (HttpServletRequest) request;

HttpServletResponse res = (HttpServletResponse) response;

int aid=Integer.parseInt(request.getParameter("aid"));

if (aid >1)

{

chain.doFilter(request, response);

}

}

@Override

public void destroy ()

{

}

}

@Component

@Order (2)

public class RequestResponseLoggingFilter implements Filter

{

@Override

public void init(FilterConfig filterConfig) throws ServletException

{

LOGGER.info ("########## Initiating Custom filter ##########");

}

@Override

public void doFilter( ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException

{

HttpServletRequest req = (HttpServletRequest) request;

HttpServletResponse res = (HttpServletResponse) response;

//Do something

}

@Override

public void destroy ()

{

}

}

For Spring to be able to recognize a filter, we needed to define it as a bean with the @Component annotation. And, to have the filters fire in the right order – we needed to use the @Order annotation. This chain.doFilter() call all other filter and in the end if there is no other filter then calls the servlet.

**B) Filter With URL Pattern**

In the example above, our filters are registered by default for all the URL's in our application. However, we may sometimes want a filter to only apply to certain URL patterns.

In this case, we must remove the @Component annotation from the filter class definition and register the filter using a FilterRegistrationBean.

@Bean

public FilterRegistrationBean<RequestResponseLoggingFilter> loggingFilter()

{

FilterRegistrationBean<RequestResponseLoggingFilter> registrationBean = new FilterRegistrationBean<>();

registrationBean.setFilter(new RequestResponseLoggingFilter());

registrationBean.addUrlPatterns("/users/\*");

return registrationBean;

}

Now the filter will only apply for paths that match the "/users/\*" pattern.

**C) Using @WebFilter and @ServletComponentScan**

With the Servlet 3.0 specification, we can use the @WebFilter annotation among others. Spring MVC has special support for scanning classes annotated with these annotations.

We’ll first create a filter using @WebFilter:

@WebFilter(urlPatterns = "/\*", dispatcherTypes = {DispatcherType.REQUEST})

public class MethodLoggingFilter extends HttpFilter

{

@Override

protected void doFilter(HttpServletRequest request, HttpServletResponse response, FilterChain chain) throws IOException, ServletException

{

log.info ("New request with {}", request.getMethod());

chain.doFilter(request, response);

}

}

Note that we’re specifying the URL patterns and dispatcher types inside the annotation. After creating the filter, we’ll make Spring scan these classes using @ServletComponentScan.

**NOTE:**

**>There is two ways of creating Filters:**

**1) By implementing Filter interface and overriding its methods as above.**

**2) By extending any of 17 Filters that Spring provides. We can extend any of them as per requirements and can override its method and write our logic. List of Security Filters registered when using <http> requests are:**

11:18:39.123 FilterChainProxy - /user/login at position 1 of 17 in additional filter chain; firing Filter: 'BasicUserApprovalFilter'

11:18:39.123 FilterChainProxy - /user/login at position 2 of 17 in additional filter chain; firing Filter: 'SecurityContextPersistenceFilter'

11:18:39.124 FilterChainProxy - /user/login at position 3 of 17 in additional filter chain; firing Filter: 'LogoutFilter'

11:18:39.124 FilterChainProxy - /user/login at position 4 of 17 in additional filter chain; firing Filter: 'UsernamePasswordAuthenticationFilter'

11:18:39.124 FilterChainProxy - /user/login at position 5 of 17 in additional filter chain; firing Filter: 'BasicAuthenticationFilter'

11:18:39.124 FilterChainProxy - /user/login at position 6 of 17 in additional filter chain; firing Filter: 'RequestCacheAwareFilter'

11:18:39.124 FilterChainProxy - /user/login at position 7 of 17 in additional filter chain; firing Filter: 'SecurityContextHolderAwareRequestFilter'

11:18:39.124 FilterChainProxy - /user/login at position 8 of 17 in additional filter chain; firing Filter: 'RememberMeAuthenticationFilter'

11:18:39.125 FilterChainProxy - /user/login at position 9 of 17 in additional filter chain; firing Filter: 'ForgotPasswordAuthenticationFilter'

11:18:39.125 FilterChainProxy - /user/login at position 10 of 17 in additional filter chain; firing Filter: 'AnonymousAuthenticationFilter'

11:18:39.125 FilterChainProxy - /user/login at position 11 of 17 in additional filter chain; firing Filter: 'SessionManagementFilter'

11:18:39.125 FilterChainProxy - /user/login at position 12 of 17 in additional filter chain; firing Filter: 'ExceptionTranslationFilter'

11:18:39.125 FilterChainProxy - /user/login at position 13 of 17 in additional filter chain; firing Filter: 'OAuth2ExceptionHandlerFilter'

11:18:39.125 FilterChainProxy - /user/login at position 14 of 17 in additional filter chain; firing Filter: 'VerificationCodeFilter'

11:18:39.125 FilterChainProxy - /user/login at position 15 of 17 in additional filter chain; firing Filter: 'OAuth2AuthorizationFilter'

11:18:39.125 FilterChainProxy - /user/login at position 16 of 17 in additional filter chain; firing Filter: 'OAuth2ProtectedResourceFilter'

11:18:39.125 FilterChainProxy - /user/login at position 17 of 17 in additional filter chain; firing Filter: 'FilterSecurityInterceptor'

**Q) Interceptor?**

You can use the Interceptor in Spring Boot to perform operations under the following situations −

* Before sending the request to the controller
* Before sending the response to the client

The DispatcherServlet is an actual Servlet and inherits from the HttpServlet base class. When a request is sent to the web application, the workflow of handling a request is as below:

When DispatcherServlet receives the request, it dispatches the task of selecting an appropriate controller to HandlerMapping. Then, HandlerMapping selects the controller which is mapped to the incoming request URL and returns the (selected Handler) and Controller to DispatcherServlet.

DispatcherServlet dispatches the task of executing the business logic by Controller to HandlerAdapter. HandlerAdapter calls the business logic process of Controller. When the HandlerAdapter calls the controller, Interceptor is invoked to intercept the request.

Then, Controller executes the business logic and sets the processing result in Model. Then, if there is any Interceptor to intercept the response, is invoked as a post process. Then Controller returns the logical name of view to HandlerAdapter.

DispatcherServlet dispatches the task of resolving the View corresponding to the View name to ViewResolver. ViewResolver returns the View mapped to View name. DispatcherServlet dispatches the rendering process to returned View. View renders Model data and returns the response.

**2.** Intercepting the request

Spring’s handler mapping mechanism includes handler interceptors, which are useful when you want to apply specific functionality to certain requests, for example, checking for a principal. Interceptors must implement HandlerInterceptor from the org.springframework.web.servlet package. This interface defines three methods:

* preHandle is executed before the execution of the target resource.
* postHandle is called after the handler is executed.
* afterCompletion is called after the complete request has finished.

These three methods should provide enough flexibility to do all kinds of pre-processing and post-processing.

The preHandle method returns a boolean value. You can use this method to break or continue the processing of the execution chain. When this method returns true, the handler execution chain will continue; when it returns false, the DispatcherServlet assumes the interceptor itself has taken care of requests (and, for example, rendered an appropriate view) and does not continue executing the other interceptors and the actual handler in the execution chain.

**Q) What is the difference between Spring MVC Interceptor and Servlet Filter?**

HandlerInterceptor is basically like a Servlet Filter, but it just allows custom pre-processing with the option of prohibiting the execution of the Handler itself, and custom post-processing.

Filters allow for exchanging the request and response objects that are passed to the chain. It means Filters work more in the request/response domain while HandlerInterceptors are spring bean and can access other beans in the application.

Note that a Filter gets configured in web.xml, a HandlerInterceptor in the application context.

**Example:**

* For authentication of web pages, you would use a servlet filter.
* For security stuff in your business layer or logging/bug tracing (a.k.a. independent of the web layer) you would use an Interceptor.

**Q) Difference between @Valid and @Validated?**

@ **Validated annotations** can be used at the class level to support Spring for parameter verification at the method level

@ **Validated** can only be used on classes, methods, and parameters, while @**Valid** can be used on methods, fields, constructors, and parameters.

**Q) Spring MVC FLOW?**

A Spring MVC is a Java framework which is used to build web applications. It follows the Model-View-Controller design pattern. It implements all the basic features of a core spring framework like Inversion of Control, Dependency Injection.

* **Model** - A model contains the data of the application. A data can be a single object or a collection of objects.
* **Controller** - A controller contains the business logic of an application. Here, the @Controller annotation is used to mark the class as the controller.
* **View** - A view represents the provided information in a particular format. Generally, JSP+JSTL is used to create a view page. Although spring also supports other view technologies such as Apache Velocity, Thymeleaf and FreeMarker.
* **Front Controller** - In Spring Web MVC, the DispatcherServlet class works as the front controller. It is responsible to manage the flow of the Spring MVC application.

## **Understanding the flow of Spring Web MVC**



**Q) AOP (Aspect Oriented programming)?**

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.

> AOP work as support for OOP.

Example: -

@Componenet

Public class Alien

{

//Logging code…

//Transaction code…

//Security code…

Public void show ()

{

System.out.println(“Hello World”);

}

}

Logging, transaction, security these kinds of code is necessary, but we must write it again and again in every method, which is not good practice. We can create Helper class and put all these method into that and then call it from classes where we need them. But still we must write code everywhere to call them. That is why **ASPECT** came into picture.

**Aspect**: Aspect is a class that implements cross-cutting concerns, such as transaction management. Aspects can be a normal class configured and then configured in Spring Bean configuration file or we can use Spring AspectJ support to declare a class as Aspect using @Aspect annotation.

@Componenet

@EnableAspectJAutoProxy

@Aspect

Class Helper

{

Public void log (){

System.out.println(“Show called”);

}

//Transaction code…………

}

>Now by running application only “Hello World” will print. It will not print “Show Called”. That means it is not getting called automatically.

**ADVICE**:

**Definition (**Advice is the action taken for a particular join point. In terms of programming, they are methods that gets executed when a specific join point with matching pointcut is reached in the application. You can think of Advices as [Spring interceptors](https://www.journaldev.com/2676/spring-mvc-interceptor-example-handlerinterceptor-handlerinterceptoradapter) or [Servlet Filters](https://www.journaldev.com/1933/java-servlet-filter-example-tutorial).

**Join Point**: A join point is a specific point in the application such as method execution, exception handling, changing object variable values etc. In Spring AOP a join point is always the execution of a method.**)**

**ADVICE**: **is what**, means when you called show() what should be called. So, what I want here is when I call show () method of Alien, I want to call this log (). So, this is my advice. And to make it an advice we have to use **“@Before, @After,etc**.”. Now how will this @Before will know that before which method it should get get execute. So now “Pointcut comes in the picture”.

**Pointcut**: Definition (Pointcut are regular expressions that are matched with join points to determine whether advice needs to be executed or not.)

it defines When or Where. Means at what point you want to call this. So, want to call this on execution of show ().

Like this: @Before (“execution (public void show ())”)

@Componenet

@EnableAspectJAutoProxy

@Aspect

Class Helper

{

@Before (“execution (public void show())”)

Public void log ()

{

System.out.println(“Show called”);

}

//Transaction methods…………….

}

### **Q) What are the different types of Advices?**

Different types of Advices in Spring AOP are:

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.

**Q) Spring Transaction Management?**

The transaction can be defined with ACID properties.

**Atomicity** – All success or none.  
**Consistency** – Database constraints should not be violated.  
**Isolation** – One transaction should not affect another one.  
**Durability** – It should in Database after commit.

**Different ways of transaction management in Spring.**

1. Programmatic transaction management.
2. Declarative transaction management.

**Programmatic transaction management** – Here we need to write some extra code for transaction management. When we say some extra code what does it mean? We need to take care of –

* Creating Transaction reference
* Begin transaction
* Commit or rollback of the transaction

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | Transaction transactionRef = entityManager.getTransaction()  try {     transactionRef.begin();     //business logic     transactionRef. commit ();  }  catch (Exception e) {     transactionRef.rollback();     e.printStackTrace();  } |

**Declarative transaction management –** No need to write extra code for getting a transaction, we can use annotations or XML-based approach to manage the transactions and we can avoid unnecessary code. If we use annotation-based approach we can use @Transactional and if we use the XML-based approach we need to configure DataSourceTransactionManager or any other transaction manager in XML as a bean. In this article and the next upcoming article, we will see the annotation-based approach.

|  |  |
| --- | --- |
| 1  2  3  4 | @Transactional (readOnly=true, propagation = Propagation.REQUIRES\_NEW)      public Book findOneString objectId) {          return repository.findOne(objectId);      } |

@Transactional this is the key annotation which is responsible for the declarative transaction management. Apart from this annotation, we use @EnableTransactionManagement. @Transactional annotation used with class or method only.

**Q) Can we use @Transactional with class?**

@Transactional on a class applies to each method on the service. It is a shortcut. Typically, youcan set @**Transactional** (readOnly = true) on a service class, if you know that all methods will access the repository layer. You can then override the behavior with @**Transactional** on methods performing changes in your model.

Consider you have class BookServiceImpl.java where you have logic to save Book entity.

|  |  |
| --- | --- |
|  | @Service("bookServiceImpl")  public class BookServiceImpl implements BookService{    @Autowired  private BookRepository bookRepository;    public Book saveBook(Book book) {  Book book1 = bookRepository.save(book);  int a = 10/0;  System.out.println(a);  return book1;  }  } |

In the above code snippet line number 9 will throw ArithmeticException but before that bookRepository.save(book) will get executed. Now the question comes would we have records in the database or not? In the above scenario even, we have an exception in our code have a record in Database.

Which we might don’t want to happen. If we have something wrong in our code data should not persist in the database. Now just modify the code snippet as below, use @Transactional annotation with saveBook() method.

|  |  |
| --- | --- |
|  | @Service("bookServiceImpl")  public class BookServiceImpl implements BookService{    @Autowired  private BookRepository bookRepository;    @Transactional  public Book saveBook(Book book) {  Book book1 = bookRepository.save(book);  int a = 10/0;  System.out.println(a);  return book1;  }  } |

Since we are using @Transactional annotation with saveBook() method and we have ArithmeticException at line number 10, our transaction will get rollbacked and we will not have any record in the database.

**NOTE: @Transactional annotation internally does the JDBC code like:**

**Connection** connection = dataSource.getConnection(); // **(1)**

**try** (connection) {

connection.setAutoCommit(**false**); // **(2)**

*// execute some SQL statements...*

connection.commit(); // **(3)**

} **catch** (**SQLException** e) {

connection.rollback(); // **(4)**

}

The @Transactional annotation has a different attribute and corresponding value.

@Transactional (isolation = Isolation.DEFAULT, propagation=Propagation.REQUIRES\_NEW,  
 readOnly=true, noRollbackFor =ArithmeticException.class, timeout = 30000,  
 value=” txManager2″, rollbackFor = { Exception.class },

rollbackForClassName = {“Exception”}, noRollbackForClassName={“Exception”})

Let’s see all attribute/property one by one.

**propagation –** propagation can have different possible value as below.

* Propagation.REQUIRED – Support a current transaction, create a new one if none exists.
* Propagation.REQUIRES\_NEW – Always create a new transaction and suspend the current transaction if already exist.
* Propagation.MANDATORY – Support a current transaction, throw an exception if none exists.
* Propagation.NESTED – Execute within a nested transaction if a current transaction exists.
* Propagation.NEVER – Execute non-transactionally, throw an exception if a transaction exists.
* Propagation.NOT\_SUPPORTED – Execute non-transactionally, suspend the current transaction if one exists.
* Propagation.SUPPORTS – Support a current transaction, execute non-transactionally if none exists.

Propagation.REQUIRED and Propagation.REQUIRES\_NEW is frequently used in real-time development.

**Default Propagation value is Propagation.REQUIRED**

**isolation –**isolation can have different possible value as below.

* Isolation.READ\_UNCOMMITTED – It allows dirty reads, non-repeatable reads, and phantom reads.
* Isolation.READ\_COMMITTED – Dirty reads are prevented, allows non-repeatable and phantom reads.
* Isolation.REPEATABLE\_READ – Dirty reads and non-repeatable prevented, phantom reads allowed.
* Isolation.SERIALIZABLE – Dirty reads, non-repeatable reads, and phantom reads are prevented. (complete isolation, ex in booking seats)

**Default isolation value is Isolation.DEFAULT.**

**rollbackFor** – We can define zero, one or multiple exceptions for which we want our transaction to be rollbacked.

@Transactional (rollbackFor = {RuntimeException.class})

**noRollbackFor** – We can define zero, one or multiple exceptions for which we don’t want our transaction to be rollbacked.

@Transactional (noRollbackFor = {RuntimeException.class})

**rollbackForClassName –** We can define zero, one or multiple exceptions as String for which we want our transaction to be rollbacked.

@Transactional (rollbackForClassName = {“NullPointerException”})

**noRollbackForClassName –** We can define zero, one or multiple exceptions as String for which we don’t want our transaction to be rollbacked.

@Transactional (noRollbackForClassName = {“NullPointerException”})

**readOnly –** Its value can be true or false If we don’t provide any value for readOnly in @Transactional, then the default value will be false.

Generally, we use @Transactional (readOnly = true) for search or retrieval operation to make sure we can only perform the read-only operation.

@Transactional (readOnly = false)

We can override readOnly behaviour using @**Modifying** annotation. For example, suppose @Transactional annotation has been used with class level or interface level as below and we want to override readOnly behaviour for one method (we don’t want to apply readOnly true for deleteOldBooks() method).

|  |  |
| --- | --- |
|  | @Repository  @Transactional (readOnly = true)  public interface BookRepository extends CrudRepository<Book,Serializable> {      List<Book> findByBookName(String bookName);      @Modifying    @Transactional    @Query ("delete from Book b where b.old= true")    void deleteOldBooks();    } |

In the above example, we will able to delete record since we are overriding

@Transactional (readOnly = true) nature using @Modifying annotation.

If we use @Transactional (readOnly = true) to a method which is performing create or update operation, then we will not have newly created or updated record into the database, but we will have the response data. Below code will not give any exception but using @Transactional (readOnly = true) while create or update operation doesn’t make any sense. Right!

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Transactional (readOnly = true)  public Student updateStudent(Student entity) {  Student studentResponse = studentRepository.save(entity);  return studentResponse;  } |

**timeout:** When we define @Transactional (timeout = 100) then we are saying our transaction should complete in given time frame otherwise we will get TransactionException(transaction time expired error).

The time value type should be int and it would be considered in milliseconds.

By default, value of timeout is -1. That means no timeouts has been supported.

NOTE: Spring's declarative transaction is enabled with AOP proxies so when calling a bean method, we must make sure that the call goes through the proxy. Assuming we are using a Spring managed bean which is not annotated with @Transactional itself, calling its method (say A) which is also not annotated with @Transactional will not be in a transaction. Now if this method calls another method B (in the same bean class) which is annotated with @Transactional, will still not enable transaction in B, that's because bean first call is not made via its transactional AOP proxy.

**Spring Annotations:**

### **1. What’s the difference between @Component, @Controller, @Repository & @Service annotations in Spring?**annotations - Spring Framework Tutorial - Edureka!

Spring @Component annotation is used to denote a class as Component. It means that Spring framework will autodetect these classes for dependency injection when annotation-based configuration and class path scanning is used.

From Spring 2.5 annotation-based dependency injection was introduced, which automatically scans and register classes as Spring bean which is annotated using @Component annotation.  
  
**@Service, @Controller and @Repository** annotation are nothing but the specialized form of **@Component** annotation for certain situations. Instead of using @Component on a controller class in Spring MVC, we use @Controller, which is more readable and appropriate.  
  
By using that annotation we do two things, first we declare that this class is a Spring bean and should be created and maintained by Spring ApplicationContext, but also we indicate that it’s a controller in MVC setup. This latter property is used by web-specific tools and functionalities. For example, DispatcherServlet will look for @RequestMapping on classes which are annotated using @Controller but not with @Component.  
  
This means @Component and @Controller are same with respect to bean creation and dependency injection but later is a specialized form of former.

Even if you replace @Controller annotation with @Compoenent, Spring can automatically detect and register the controller class (**There wouldn’t be any error**) but it may not work as you expect with respect to request mapping.

Same is true for @Service and @Repository annotation, they are a specialization of @Component in service and persistence layer. A Spring bean in the service layer should be annotated using @Service instead of @Component annotation and a spring bean in the persistence layer should be annotated with @Repository annotation.  
  
By using a specialized annotation we hit two birds with one stone. First, they are treated as Spring bean and second you can put special behaviour required by that layer.  
  
For example, @Repository's not only helping in annotation based configure but also catch Platform specific exceptions and re-throw them as one of Spring’s unified unchecked exception.  
  
This bean post processor adds an advisor to any bean that’s annotated with @Repository so that any platform-specific exceptions are caught and then rethrown as one of Spring’s unchecked data access exceptions.

**Thus, it’s not wrong to say that @Controller, @Service, and @Repository are special types of @Component annotation. They are scanned because they themselves are annotated with @Component annotation.** If you define your own custom annotation and annotate it with @Component, then it will also get scanned with <context:component-scan>.

**@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 behaviour 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.

**@Configuration:** Used to indicate that a class declares one or more @Bean methods. These classes are processed by the Spring container to generate bean definitions and service requests for those beans at runtime.

**@Bean:** Indicates that a method produces a bean to be managed by the Spring container. This is one of the most used and important spring annotations. @Bean annotation also can be used with parameters like name, initMethod and destroyMethod.

* name – allows you give name for bean
* initMethod – allows you to choose method which will be invoked on context register
* destroyMethod – allows you to choose method which will be invoked on context shutdown

**@ComponentScan:** Configures component scanning directives for use with @Configuration classes. Here we can specify the base packages to scan for spring components.

**@PropertySource:** provides a simple declarative mechanism for adding a property source to Spring’s Environment. There is a similar annotation for adding an array of property source files i.e. @PropertySource

## **Q) Spring @RequestMapping?**

**@RequestMapping with Class**: We can use it with class definition to create the base URI. For example:

@Controller

@RequestMapping

Public class HomeController

{

}

Now /home is the URI for which this controller will be used. This concept is very similar to servlet context of a web application.

**@RequestMapping with Method**: We can use it with method to provide the URI pattern for which handler method will be used. For example:

@RequestMapping(value=”/method”)

@ResponseBody

Public String method0()

{

Return “method0”;

}

Above annotation can also be written as @RequestMapping("/method0"). On a side note, I am using @ResponseBody to send the String response for this web request.

**@RequestMapping with Multiple URI**: We can use a single method for handling multiple URIs, for example:

@RequestMapping(value={”/method”, “/method1/second”})

@ResponseBody

Public String method1()

{

Return “method1”;

}

If you will look at the source code of **RequestMapping annotation**, you will see that all its variables are arrays. We can create String array for the URI mappings for the handler method.

**@RequestMapping with HTTP Method**: Sometimes we want to perform different operations based on the HTTP method used, even though request URI remains same. We can use @RequestMapping method variable to narrow down the HTTP methods for which this method will be invoked. For example:

@RequestMapping(value=”/method2”, “method=RequestMethod.POST)

@ResponseBody

Public String method2()

{

Return “method2”;

}

@RequestMapping(value=”/method3”, “method=RequestMethod.POST, RequestMethod.GET)

@ResponseBody

Public String method3()

{

Return “method3”;

}

**@RequestMapping with Headers**: We can specify the headers that should be present to invoke the handler method. For example:

@RequestMapping(value=”/method4”, “headers=”name=pankaj”)

@ResponseBody

Public String method4()

{

Return “method4”;

}

@RequestMapping(value=”/method5”, “headers={”name=pankaj”, “id=1”})

@ResponseBody

Public String method5()

{

Return “method5”;

}

**@RequestMapping with Produces and Consumes**: We can use header Content-Type and Accept to find out request contents and what is the mime message it wants in response. For clarity, @RequestMapping provides **produces** and **consumes** variables where we can specify the request content-type for which method will be invoked and the response content type. For example:

@RequestMapping(value=”/method6”, produces={“application/json”,”application/xml”}, consumes=”text/html”)

@ResponseBody

Public String method6()

{

Return “method6”;

}

Above method can consume message only with **Content-Type as text/html** and is able to produce messages of type **application/json** and **application/xml**.

## **Spring @PathVariable**

**@RequestMapping with @PathVariable**: RequestMapping annotation can be used to handle dynamic URIs where one or more of the URI values works as a parameter. We can even specify Regular Expression for URI dynamic parameter to accept only specific type of input. It works with **@PathVariable annotation** through which we can map the URI variable to one of the method arguments. For example:

@RequestMapping(value=”/method7/{id}”)

@ResponseBody

Public String method7(@PathVariable(“id”) int id)

{

return “method7 with id=” +id;

}

@RequestMapping(value=”/method8/{id: [\\d]+}/{name}”)

@ResponseBody

Public String method7(@PathVariable(“id”) long id, @PathVariable(“name”) String name)

{

Return “method8 with id=”+id+” and name=”+name;

}

**Spring @RequestParam**

**@RequestMapping with @RequestParam for URL parameters**: Sometimes we get parameters in the request URL, mostly in GET requests. We can use @RequestMapping with **@RequestParam annotation** to retrieve the URL parameter and map it to the method argument. For example:

@RequestMapping(value=”/method9”)

@ResponseBody

Public String method9(@RequestParam(“id”) int id)

{

return “method9 with id=” +id ;

}

For this method to work, the parameter name should be “id” and it should be of type int.

**@RequestMapping default method**: If value is empty for a method, it works as default method for the controller class. For example:

@RequestMapping()

@ResponseBody

Public String defaultMethod()

{

return “Default Method”;

}

As you have seen above that we have mapped /home to HomeController, this method will be used for the default URI requests.

**@RequestMapping fallback method**: We can create a fallback method for the controller class to make sure we are catching all the client requests even though there are no matching handler methods. It is useful in sending custom 404 response pages to users when there are no handler methods for the request.

@RequestMapping(“\*”)

@ResponseBody

Public String fallbackMethod()

{

return “fallbackMethod Method”;

}

**@QueryParam**: To consume values from query string we use @QueryParam. It is applied in method argument level. For more than one keys in query string, we need to use more than one @QueryParam. Suppose query string is like:-  
*?name=Ram&collegeName=UP College*  
  
Then we can use @QueryParam as below:-  
*(@QueryParam("name") String name,@QueryParam("collegeName") String collegeName)*

getStudentDetails(@QueryParam("name") String name, @QueryParam("collegeName") String collegeName)

{

}

**Difference:**

1)

@**QueryParam** is used to access key/value pairs in the query string of the URL (the part after the ?). For example in the url <http://stackoverflow.com/questions?q=query>, you can use @QueryParam("q") to get the value of q.

@**PathParam** is used to match a part of the URL as a parameter. For example in an url of the form <http://stackoverflow.com/questions/>{questionid}, you can use @PathParam("questionid") to get the id of a question

2) @RequestParam will always expect a value to bind. Hence, if value is not passed, it will give error. This is not the case in @QueryParam

To explicitly give the option, use required = false while using @RequestParam

**Q) @Data?**

@Data is like having implicit @Getter, @Setter, @ToString, @EqualsAndHashCode and @RequiredArgsConstructor annotations on the class (except that no constructor will be generated if any explicitly written constructors already exist). However, the parameters of these annotations (such as callSuper, includeFieldNames and exclude) cannot be set with @Data. If you need to set non-default values for any of these parameters, just add those annotations explicitly; @Data is smart enough to defer to those annotations.

**NOTE: @Data is available in Lombok jars.**

**Q)** **@Retryable**

We can build applications using **spring retry module** facility where we must call some methods where exception is sometimes expected, and we have to retry the request.

#### Spring retry annotations

* **@EnableRetry** – to enable spring to retry in spring boot project
* **@Retryable** – to indicate any method to be a candidate of retry
* **@Recover** – to specify fallback method!

#### @EnableRetry annotation

To enable spring-retry we need to put one annotation in the Spring Boot Application class. So open SpringRetryApplication class and add @EnableRetry in class level.

@Retryable(value = { RemoteServiceNotAvailableException.class }, maxAttempts = 3, backoff = @Backoff(delay = 1000))

public String getBackendResponse(boolean simulateretry, boolean simulateretryfallback);

@Recover

public String getBackendResponseFallback(RuntimeException e);

We have to add @Retryable annotation on the method with expected expectation and if that annotation occurs then method will be called at specified maxAttempts values.

**@Recover** – in the fallback method indicates that if we don’t get any success response after 3 retry, response will come from this fallback method. Make sure you pass expected exception as parameter; else spring will have hard time finding the exact method.

**Q)@RequestMapping, @GetMapping, @PostMapping, @PutMapping, @DeleteMapping and @PatchMapping?**

**@GetMapping** is specialized version of @RequestMapping annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.GET).

**@PostMapping** is specialized version of @RequestMapping annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.POST).

**Q) @Value?**

Use to get values from application.properties. example

**//application.properties**

value.from.file=Sagar

**//In class**

@Value("${value.from.file}")

private String name;

**Q) Spring Security?**

Spring Security is a framework which provides various security features like:

* Authentication: Username, Password
* Authorization: Position (Ex: Manager, Employee)
* Principal: Currently Logged in user
* Granted Authority: This is to give Authorization an access.
* Roles: Like User, Admin, Manager, Employee.

ThreadLocal

Authentication

UserDetailsService

loadByUserName()

AuthenticationProvider

Supports()

Authenticate()

AuthenticationProvider

Supports()

Authenticate()

AuthenticationProvider

Supports()

Authenticate()

Authentication Manager

Authenticate()

Authentication

Credential

F

i

L

ter

input

output

Authentication

Credential

Authentication filter intercept authentication request, it creates the authentication object with creadentials and passes it to the AuthenticationManager, AuthenticationManager finds the right AuthenticationProvider that can handle the authentication using the supports() , then the AuthenticationManager calls the authenticate() on that AuthenticationProvider and passes the authentication object.

The AuthenticationProvider looks up the correspnding user in the system by using the UserDetailService. UserDetailService returns the UserDetails instance which the authentication provider then verifies and then the other indication happens, if the authentication is successful the authentication object is returned back with principle and the authorities filled in the principle often being just that instance of the UserDetails object then this authentication object goes all the way back to the authentication filter that started at all.

**Q) what happens after this the authentication filter gets the authentication object with the principal now what does it do?**

It takes that object and saves in the thread context. as always need to know who logged user is. so, their Security Context that is associated with the current thread, this authentication object as a result of successful authentication is put into the Security Context in the Thread Local object for use in authorization for use in identifying the current principal and more.

**Q) what is .m2 in maven?**

The local repository of **Maven** is a folder location on the developer's machine, where all the project artifacts are stored locally. When **maven** build is executed, **Maven** automatically downloads all the dependency jars into the local repository. Usually this folder is named. **m2**.

**Q) Why character array is better than String for Storing password in Java?**

To avoid password being displayed in plain text in thread dump/heap analyser, during debugging and attacks, we should zero memory containing sensitive information immediately after use instead of waiting for the garbage collection to kick in.

**Why String should not be used to store passwords?**  
It seems logical and relatively easier to store a password in a String object rather than in a character array. But since String is immutable in Java, we can’t change its contents after usage. That means if we use a String object for storing passwords, we can’t get rid of the password until Garbage collector clears it, which poses a big security threat. So, the immutable property of String makes it vulnerable for storing passwords.

Secondly, there’s always a risk of accidentally printing the password to the application logs.

String password=”pwd@1234”;

logger.info(password); //prints pwd@1234

**Why character array is preferred over String?**

A character array should always be used to collect and store sensitive information. Character arrays are mutable data structures, which can be cleared immediately after use and the password won’t have any trace left anywhere in the application, even before the garbage collection. So, character array is less vulnerable than a String, even though it only reduces the attack window for the successful hack and doesn’t eliminate the risk.

Even this is not secure as password can be still logged in several other ways:

Char [] chars= new char [] {‘p’,’w’,’d’,’@’,’1’,’2’,’3’,’4’};

Sop(cars); //prints pwd@1234

logger.info (Arrays.toString(chars)); //prints [p, w, d, @, 1, 2, 3, 4]

So, we can conclude that a character array is more secure than the String object even though it also can be exploited. In order to avoid any leaks, we should always encrypt a password rather than storing it in a plain text and clear it from heap as soon as user is authenticated.

## **Q)** **@DependsOn?**

We should use this annotation for specifying bean dependencies. **Spring guarantees that the defined beans will be initialized before attempting an initialization of the current bean.**

Let's say we have a FileProcessor which depends on a FileReader and FileWriter. In this case, FileReader and FileWriter should be initialized before the FileProcessor.

|  |
| --- |
| @Configuration  @ComponentScan("com.baeldung.dependson")  public class Config {        @Bean      @DependsOn({"fileReader","fileWriter"})      public FileProcessor fileProcessor(){          return new FileProcessor();      }        @Bean("fileReader")      public FileReader fileReader() {          return new FileReader();      }        @Bean("fileWriter")      public FileWriter fileWriter() {          return new FileWriter();      }  } |

***FileProcessor* specifies its dependencies with *@DependsOn***. We can also annotate a *Component*with *@DependsOn:*

@Component

@DependsOn({"filereader", "fileWriter"})

public class FileProcessor {}

NOTE: **While using @DependsOn, we must use component-scanning**

**Q) @CrossOrigin?**

**Cross-origin** resource sharing (**CORS**) is a standard protocol that defines the interaction between a browser and a server for safely handling **cross-origin** HTTP requests.

### 1)@CrossOrigin**on a**@RequestMapping-**Annotated Handler Method**

@RestController

@RequestMapping("/account")

public class AccountController {

    @CrossOrigin

    @RequestMapping(method = RequestMethod.GET, path = "/{id}")

    public Account retrieve (@PathVariable Long id) {

        // ...

    }

    @RequestMapping(method = RequestMethod.DELETE, path = "/{id}")

    public void remove (@PathVariable Long id) {

        // ...

    }

}

CORS is only enabled for the *retrieve ()* method. We can see that we didn't set any configuration for the *@CrossOrigin* annotation, so the default configuration takes place:

* All origins are allowed
* The HTTP methods allowed are those specified in the *@RequestMapping* annotation (for this example is GET)
* The time that the preflight response is cached (*maxAge)* is 30 minutes

### 2)@CrossOrigin**on the Controller**

@CrossOrigin(origins = "[http://example.com](http://example.com/)", maxAge = 3600)

@RestController

@RequestMapping("/account")

public class AccountController {

    @RequestMapping(method = RequestMethod.GET, path = "/{id}")

    public Account retrieve (@PathVariable Long id) {

        // ...

    }

 @RequestMapping(method = RequestMethod.DELETE, path = "/{id}")

    public void remove (@PathVariable Long id) {

        // ...

    }

}

# **Q) Hystrix Circuit Breaker Pattern – Spring Cloud?**

Hystrix configuration is done in four major steps.

1. Add Hystrix starter and dashboard dependencies.

|  |
| --- |
|  |

1. Add @EnableCircuitBreaker annotation
2. Add @EnableHystrixDashboard annotation
3. Add annotation @HystrixCommand(fallbackMethod = "myFallbackMethod")

If we design our systems on microservice based architecture, we will generally develop many Microservices and those will interact with each other heavily in achieving certain business goals. Now, all of us can assume that this will give expected result if all the services are up and running and response time of each service is satisfactory.

Now what will happen if any service, of the current Eco system, has some issue and stopped servicing the requests. It will result in timeouts/exception and the whole Eco system will get unstable due to this single point of failure.

Here circuit breaker pattern comes handy and it redirects traffic to a fall-back path once it sees any such scenario. Also, it monitors the defective service closely and restore the traffic once the service came back to normalcy.

@RestController

public class SchoolServiceController {

    @Autowired

    StudentServiceDelegate studentServiceDelegate;

    @RequestMapping(value = "/getSchoolDetails/{schoolname}", method = RequestMethod.GET)

    public String getStudents(@PathVariable String schoolname) {

        System.out.println("Going to call student service to get data!");

        return studentServiceDelegate.callStudentServiceAndGetData(schoolname);

    }

}

@Service

public class StudentServiceDelegate {

    @Autowired

    RestTemplate restTemplate;

    @HystrixCommand(fallbackMethod = "callStudentServiceAndGetData\_Fallback")

    public String callStudentServiceAndGetData(String schoolname) {

        System.out.println("Getting School details for " + schoolname);

        String response = restTemplate

                .exchange("<http://localhost:8098/getStudentDetailsForSchool/>{schoolname}"

                , HttpMethod.GET

                , null

                , new ParameterizedTypeReference<String>() {

            }, schoolname).getBody();

        System.out.println("Response Received as " + response + " -  " + new Date());

        return "NORMAL FLOW!!! - School Name -” + schoolname + " :::  " +

                    " Student Details " + response + " - ” + new Date();

    }

      @SuppressWarnings("unused")

     private String callStudentServiceAndGetData\_Fallback(String schoolname) {

         System.out.println("Student Service is down!!! fallback route enabled...");

         return "CIRCUIT BREAKER ENABLED!!! No Response from Student Service at this moment. " +

                    " Service will be back shortly - " + new Date ();

     }

     @Bean

     public RestTemplate restTemplate() {

         return new RestTemplate();

     }

}

If getStudentDetailsForSchool() is down then fallback method will execute.

**Q) Event Handling?**

Spring has inbuilt support for creating application events, publish them and then listen to it in event handlers.

**Event**: class that contain info about event which the publisher publishes and listener listen to (event should extend ApplicationEvent)

**Event publisher**: The entity which publishes the event. (publisher should inject an ApplicationEventPublisher object)

**Event Listner:** The entity which listening to the event (implement the ApplicationListener interface)

//DrawEvent.java

class DrawEvent extends ApplicationEvent

{

public DrawEvent(Object source){

super(source);

}

public String toString(){ //Overriding this to return custom message

return "Draw Event Occured";

}

}

//In order to publish event, we have to get "ApplicationEventPublisher"

//Circle.java

public class Circle implements ApplicationEventPublisherAware//spring internally will provide publisher object to Circle, anf for that we have to implement its method setApplicationEventPublisher()

{

private ApplicationEventPublisher publisher; //we must ask the spring to set publisher into this ref.

@Override

public void setApplicationEventPublisher(ApplicationEventPublisher publisher) //This publisher that spring gives us is actually the ApplicationContext

{

this.publisher= publisher;

}

@Override

public void draw (){

DrawEvent drawEvent= new DrawEvent(this); //create event and publish it by using this "publishEvent(drawEvent)"

publisher.publishEvent(drawEvent); //Here we are actually calling method from ApllicationContext as ApllicationContext implements ApplicationEventPublisher

}

}

//MyEventListner.java

@Component

class MyEventListner implements ApplicationListener

{

@Override

public void onApplicationEvent(ApplicationEvent event) {//spring is gonna send ApplicationEvent object to this method so whatever code we will write into this method will get executed upon every event that publish in the spring framework.

sop(event.toString());//as of now it will catch all the framework related event

}

}