SPRING FRAMEWORK

Java Framework is the body or platform of pre-written codes used by Java developers to develop Java applications or web applications. In other words, **Java Framework** is a collection of predefined classes and functions that is used to process input, manage hardware devices interacts with system software. It acts like a skeleton that helps the developer to develop an application by writing their own code.

It was **developed by Rod Johnson in 2003**. Spring framework makes the easy development of JavaEE application.

It is helpful for beginners and experienced persons.

Spring Framework

Spring is a *lightweight* framework. It can be thought of as a *framework of frameworks* because it provides support to various frameworks such as Struts, Hibernate, Tapestry, EJB, JSF, etc. The framework, in broader sense, can be defined as a structure where we find solution of the various technical problems.

The Spring framework comprises several modules such as IOC, AOP, DAO, Context, ORM, WEB MVC etc. We will learn these modules in next page. Let's understand the IOC and Dependency Injection first.

Advantages of Spring Framework

There are many advantages of Spring Framework. They are as follows:

1) Predefined Templates

Spring framework provides templates for JDBC, Hibernate, JPA etc. technologies. So there is no need to write too much code. It hides the basic steps of these technologies.

Let's take the example of JdbcTemplate, you don't need to write the code for exception handling, creating connection, creating statement, committing transaction, closing connection etc. You need to write the code of executing query only. Thus, it save a lot of JDBC code.

2) Loose Coupling

The Spring applications are loosely coupled because of dependency injection.

3) Easy to test

The Dependency Injection makes easier to test the application. The EJB or Struts application require server to run the application but Spring framework doesn't require server.

4) Lightweight

Spring framework is lightweight because of its POJO implementation. The Spring Framework doesn't force the programmer to inherit any class or implement any interface. That is why it is said non-invasive.

5) Fast Development

The Dependency Injection feature of Spring Framework and it support to various frameworks makes the easy development of JavaEE application.

6) Powerful abstraction

It provides powerful abstraction to JavaEE specifications such as JMS, JDBC, JPA and JTA.

7) Declarative support

It provides declarative support for caching, validation, transactions and formatting.

Spring - Dependency Injection

Every Java-based application has a few objects that work together to present what the end-user sees as a working application. When writing a complex Java application, application classes should be as independent as possible of other Java classes to increase the possibility to reuse these classes and to test them independently of other classes while unit testing. Dependency Injection (or sometime called wiring) helps in gluing these classes together and at the same time keeping them independent.

Consider you have an application which has a text editor component and you want to provide a spell check. Your standard code would look something like this –

```
public class TextEditor {
   private SpellChecker spellChecker;

   public TextEditor() {
      spellChecker = new SpellChecker();
   }
```

What we've done here is, create a dependency between the TextEditor and the SpellChecker. In an inversion of control scenario, we would instead do something like this −

```
public class TextEditor {
   private SpellChecker spellChecker;

   public TextEditor(SpellChecker spellChecker) {
      this.spellChecker = spellChecker;
   }
}
```

Here, the TextEditor should not worry about SpellChecker implementation. The SpellChecker will be implemented independently and will be provided to the TextEditor at the time of TextEditor instantiation. This entire procedure is controlled by the Spring Framework.

Here, we have removed total control from the TextEditor and kept it somewhere else (i.e. XML configuration file) and the dependency (i.e. class SpellChecker) is being injected into the class TextEditor through a **Class Constructor.** Thus the flow of control has been "inverted" by Dependency Injection (DI) because you have effectively delegated dependances to some external system.

The second method of injecting dependency is through **Setter Methods** of the TextEditor class where we will create a SpellChecker instance. This instance will be used to call setter methods to initialize TextEditor's properties.

Dependency Injection is the main functionality provided by Spring IOC(Inversion of Control). The Spring-Core module is responsible for injecting dependencies through either Constructor or Setter methods. The design principle of Inversion of Control emphasizes keeping the Java classes independent of each other and the container frees them from object creation and maintenance. These classes, managed by Spring, must adhere to the standard definition of Java-Bean. Dependency Injection in Spring also ensures loose coupling between the classes. There are two types of Spring Dependency Injection.

1.**Constructor-based dependency injection**
Constructor-based DI is accomplished when the container invokes a class constructor with a number of arguments, each representing a dependency on the other class.

2.**Setter-based dependency injection**
Setter-based DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.

You can mix both, Constructor-based and Setter-based DI but it is a good rule of thumb to use constructor arguments for mandatory dependencies and setters for optional dependencies.

The code is cleaner with the DI principle and decoupling is more effective when objects are provided with their dependencies. The object does not look up its dependencies and does not know the location or class of the dependencies, rather everything is taken care by the Spring Framework.

Spring Dependency Injection with Example

What is Dependency Injection:
Dependency Injection is the main functionality provided by
Spring

IOC(Inversion of Control). The Spring-Core module is responsible for injecting dependencies through either Constructor or Setter methods. The design principle of Inversion of Control emphasizes keeping the Java classes independent of each other and the container frees them from object creation and maintenance. These classes, managed by
Spring

, must adhere to the standard definition of Java-Bean. Dependency Injection in
Spring

also ensures loose-coupling between the classes.
Need for Dependency Injection:
Suppose class One needs the object of class Two to instantiate or operate a method, then class One is said to be
dependent
on class Two. Now though it might appear okay to depend a module on the other but, in the real world, this could lead to a lot of problems, including system failure. Hence such dependencies need to be avoided.
Spring

IOC resolves such dependencies with Dependency Injection, which makes the code
easier to test and reuse
.
Loose coupling

between classes can be possible by defining

Need for Dependency Injection:
Suppose class One needs the object of class Two to instantiate or operate a method, then class One is said to be
dependent
on class Two. Now though it might appear okay to depend a module on the other but, in the real world, this could lead to a lot of problems, including system failure. Hence such dependencies need to be avoided.
Spring

IOC resolves such dependencies with Dependency Injection, which makes the code
easier to test and reuse
.
Loose coupling

between classes can be possible by defining
interfaces

for common functionality and the injector will instantiate the objects of required implementation. The task of instantiating objects is done by the container according to the configurations specified by the developer.
Types of Spring Dependency Injection:
There are two types of Spring Dependency Injection. They are:

1. **Setter Dependency Injection (SDI)**: This is the simpler of the two DI methods. In this, the DI will be injected with the help of setter and/or getter methods. Now to set the DI as SDI in the bean, it is done through the **bean-configuration file** For this, the property to be set with the SDI is declared under the **<property>** tag in the bean-config file. **Example: Let us say there is class GFG that uses SDI and sets the property geeks.**
   2. This injects the 'CsvGFG' bean into the 'GFG' object with the help of a setter method ('setGeek')
   3. **Constructor Dependency Injection (CDI)**: In this, the DI will be injected with the help of contructors. Now to set the DI as CDI in bean, it is done through the **bean-configuration file** For this, the property to be set with the CDI is declared under the **<constructor-arg>** tag in the bean-config file.

Spring IoC (Inversion of Control)

Spring IoC (Inversion of Control) Container is the core of Spring Framework. It creates the objects, configures and assembles their dependencies, manages their entire life cycle. The Container uses Dependency Injection(DI) to manage the components that make up the application. It gets the information about the objects from a configuration file(XML) or Java Code or Java Annotations and Java POJO class. These objects are called Beans. Since the Controlling of Java objects and their lifecycle is not done by the developers, hence the name

Inversion Of Control.
There are 2 types of IoC containers:

- BeanFactory
- ApplicationContext

That means if you want to use an IoC container in spring whether we need to use a BeanFactory or ApplicationContext. The BeanFactory is the most basic version of IoC containers, and the ApplicationContext extends the features of BeanFactory. The followings are some of the main features of Spring IoC,

- Creating Object for us,
- Managing our objects,
- Helping our application to be configurable,
- Managing dependencies

  - **Implementation:** So now let's understand what is IoC in Spring with an example. Suppose we have one interface named Sim and it has some abstract methods calling() and data().
  -

```
// Java Program to Illustrate Sim Interface

public interface Sim

{

    void calling();

    void data();

}
```

  - Now we have created another two classes Airtel and Jio which implement the Sim interface and override the interface methods.

```
// Java Program to Illustrate Airtel Class



// Class

// Implementing Sim interface
```

```
public class Airtel implements Sim {



    @Override public void calling()

    {

        System.out.println("Airtel Calling");

    }



    @Override public void data()

    {

        System.out.println("Airtel Data");

    }

}
```

```
// Java Program to Illustrate Jio Class



// Class

// Implementing Sim interface

public class Jio implements Sim{

    @Override

    public void calling() {

        System.out.println("Jio Calling");

    }
    @Override

    public void data() {

        System.out.println("Jio Data");

    }

}
```

- So let’s now call these methods inside the main method. So by implementing the Run time polymorphism concept we can do something like this

```
// Java Program to Illustrate Mobile Class



// Class

public class Mobile {



    // Main driver method

    public static void main(String[] args)

    {



        // Creating instance of Sim interface

        // inside main() method

        // with reference to Jio class constructor

        // invocation
        Sim sim = new Jio();



        // Sim sim = new Airtel();



        sim.calling();

        sim.data();

    }

}
```

-
  But what happens if in the future another new Sim Vodafone came and we need to change again to the child class name in the code, like this
- `Sim sim = new Vodafone();`
- So we have to do our configuration in the source code. So how to make it configurable? We don't want to touch the source code of this. The source code should be constant. And how can we make it? Here Spring IoC comes into the picture. So in this example, we are going to use ApplicationContext to implement an IoC container. First, we have to create an XML file and name the file as "**beans.xml**".
- **Example:** beans.xml File
- **XML**

```xml

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

       xsi:schemaLocation="http://www.springframework.org/schema/beans

       https://www.springframework.org/schema/beans/spring-beans.xsd">



  <bean id="sim" class="Jio"></bean>

```

```
</beans>
```

- **Output Explanation:** In the beans.xml file, we have created **beans**. So inside the id, we have to pass the unique id and inside the class, we have to pass the Class name for which you want to create the bean. Later on, inside the main method, we can tweek it out that will be described in the upcoming program.
- ***Bean Definition**: In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container.*
- **Java**

```
import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;



public class Mobile {

    public static void main(String[] args) {

        // Using ApplicationContext tom implement Spring IoC

        ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("beans.xml");



        // Get the bean

        Sim sim = applicationContext.getBean("sim", Sim.class);



        // Calling the methods

        sim.calling();
        sim.data();

    }

}
```

- **Output:**

-
  ```
  Jio Calling
  ```

-
  ```
  Jio Data
  ```
- And now if you want to use the Airtel sim so you have to change only inside the **beans.xml** file. The main method is going to be the same.

-
  ```
  <bean id="sim" class="Airtel"></bean>
  ```

```
import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;



public class Mobile {

    public static void main(String[] args) {



        // Using ApplicationContext tom implement Spring IoC

        ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("beans.xml");



        // Get the bean

        Sim sim = applicationContext.getBean("sim", Sim.class);



        // Calling the methods

        sim.calling();
```

```
          sim.data();

      }

  }
```

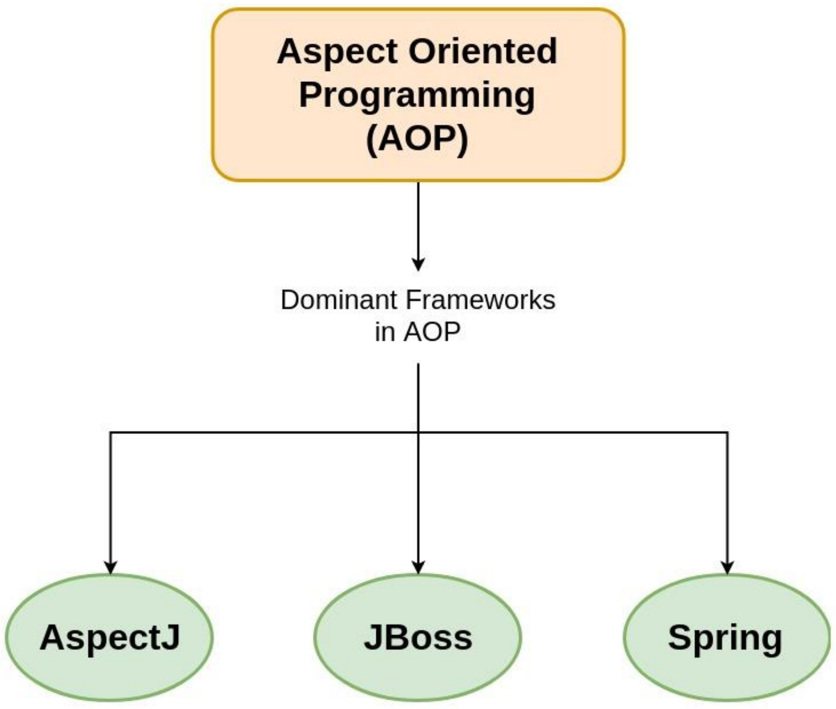
- **Output:**
- `Airtel Calling`
- `Airtel Data`

AOP in Spring Framework

Aspect oriented programming(AOP) as the name suggests uses aspects in programming. It can be defined as the breaking of code into different modules, also known as modularisation, where the aspect is the key unit of modularity. Aspects enable the implementation of crosscutting concerns such as- transaction, logging not central to business logic without cluttering the code core to its functionality. It does so by adding additional behaviour that is the advice to the existing code. For example- Security is a crosscutting concern, in many methods in an application security rules can be applied, therefore repeating the code at every method, define the functionality in a common class and control were to apply that functionality in the whole application.

Dominant Frameworks in AOP:

AOP includes programming methods and frameworks on which modularisation of code is supported and implemented. Let's have a look at



the three **dominant frameworks in AOP**:

- **AspectJ:** It is an extension for Java programming created at **PARC research centre**. It uses Java like syntax and included IDE integrations for displaying crosscutting structure. It has its own compiler and weaver, on using it enables the use of full AspectJ language.
- **JBoss:** It is an open source Java application server developed by JBoss, used for Java development.
- Spring**:** It uses XML based configuration for implementing AOP, also it uses annotations which are interpreted by using a library supplied by AspectJ for parsing and matching.

Currently, **AspectJ libraries with Spring framework** are dominant in the market, therefore let's have an understanding of how Aspect-oriented programming works with Spring.

How Aspect-Oriented Programming works with Spring:
One may think that invoking a method will automatically implement cross-cutting concerns but that is not the case. Just invocation of the method does not invoke the advice(the job which is meant to be done). Spring uses **proxy based mechanism** i.e. it creates a proxy Object which will wrap around the original object and will take up the advice which is relevant to the method call. Proxy objects can be created either manually through proxy factory bean or through auto proxy configuration in the XML file and get destroyed when the execution completes. Proxy objects are used to enrich the Original behaviour of the real object.

A **cross-cutting concern** is a concern that can affect the whole application and should be centralized in one location in code as possible, such as transaction management, authentication, logging, security etc.

Common terminologies in AOP:

1. **Aspect:** The class which implements the JEE application cross-cutting concerns(transaction, logger etc) is known as the aspect. It can be normal class configured through XML configuration or through regular classes annotated with @Aspect.
2. **Weaving:** The process of linking Aspects with an Advised Object. It can be done at load time, compile time or at runtime time. Spring AOP does weaving at runtime.
   Let's write our first aspect class but before that have a look at the jars required and the Bean configuration file for AOP.

BEAN SCOPE

***Bean Definition**: In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container.*

Bean Scopes refers to the lifecycle of Bean that means when the object of Bean will be instantiated, how long does that object live, and how many objects will be created for that bean throughout. Basically, it controls the instance creation of the bean and it is managed by the spring container.

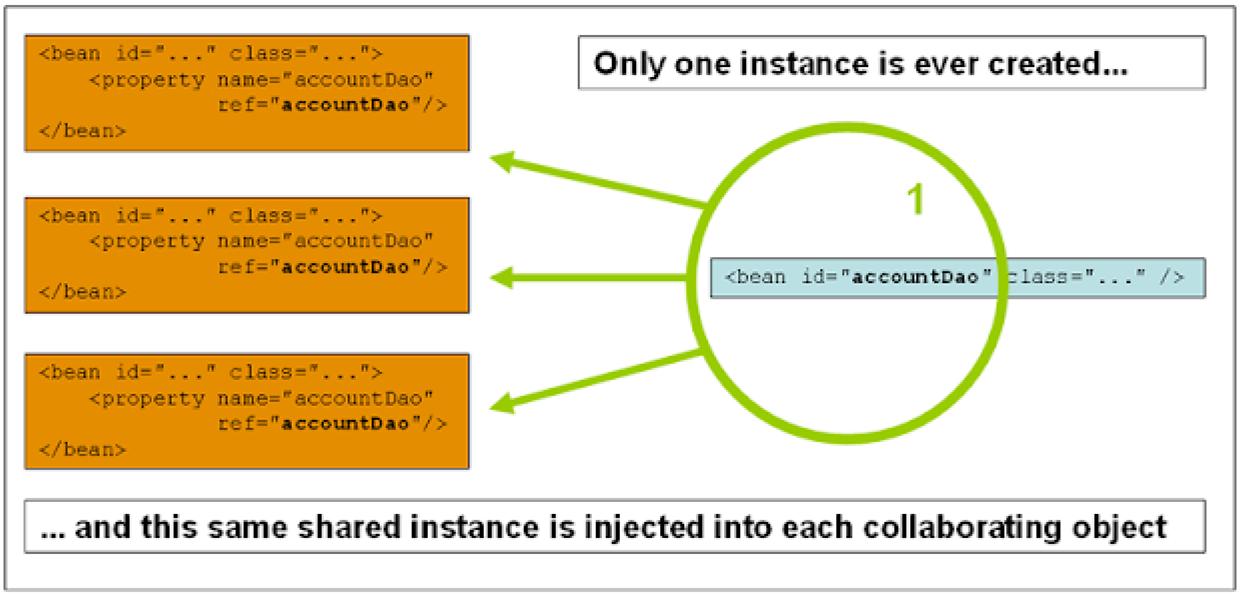
Spring Framework supports following bean scopes :

- **singleton:** (Default) Scopes a single bean definition to a single object instance per Spring IoC container.
- **prototype:** Scopes a single bean definition to any number of object instances.
- **request:** Scopes a single bean definition to the lifecycle of a single HTTP request; that is, each HTTP request has its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring ApplicationContext.
- **session:** Scopes a single bean definition to the lifecycle of an HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext.
- **application:** Scopes a single bean definition to the lifecycle of a ServletContext. Only valid in the context of a web-aware Spring ApplicationContext.
- **WebSocket:** Scopes a single bean definition to the lifecycle of a WebSocket. Only valid in the context of a web-aware Spring ApplicationContext.

The singleton scope

Only one shared an instance of a singleton bean is managed, and all requests for beans with an id or ids matching that bean definition result in that one specific bean instance being returned by the Spring container.

To put it another way, when you define a bean definition and it is scoped as a **singleton**, the **Spring IoC container** creates exactly one instance of the object defined by that bean definition. This single instance is stored in a cache of such singleton beans, and all subsequent requests and references for that named bean return the cached object.


```
<bean id="..." class="...">
    <property name="accountDao"
              ref="accountDao"/>
</bean>

<bean id="..." class="...">
    <property name="accountDao"
              ref="accountDao"/>
</bean>

<bean id="..." class="...">
    <property name="accountDao"
              ref="accountDao"/>
</bean>

<bean id="accountDao" class="..." />
```
Only one instance is ever created...
1
... and this same shared instance is injected into each collaborating object

Key points

1. Spring's concept of a singleton bean differs from the **Singleton Pattern** as defined in the Gang of Four (GoF) patterns book. The GoF Singleton hard-codes the scope of an object such that one and only one instance of a particular class is created per ClassLoader.
2. Spring Singleton beans are not threaded safe.
3. The singleton scope is the default scope in Spring.

To define a bean as a singleton in XML, we would write, for example:

```
<bean id="accountService" class="com.foo.DefaultAccountService"/>

<!-- the following is equivalent, though redundant (singleton scope is the default) -->
<bean id="accountService" class="com.foo.DefaultAccountService" scope="singleton"/>
```

To define a bean as a **singleton** in **java based bean configuration**, we would write, for example:

```
@Configuration
public class AppConfiguration {

 @Bean
 @Scope("singleton") // default scope
 public UserService userService(){
  return new UserService();
 }
}
```

The prototype scope

The non-singleton, prototype scope of bean deployment results in the creation of a new bean instance every time a request for that specific bean is made. That is, the bean is injected into another bean or you request it through a **getBean()** method call on the container.

As a rule, use the prototype scope for all stateful beans and the singleton scope for stateless beans.

The following diagram illustrates the Spring prototype scope. A data access object (DAO) is not typically configured as a prototype, because a typical **DAO** does not hold any conversational state; it was just easier for this author to reuse the core of the singleton diagram. The following example defines a bean as a prototype in XML:

~~~
<bean id="accountService" class="com.foo.DefaultAccountService" scope="prototype"/>
```xml
To define a bean as a prototype in java based bean configuration, you would write, for
example:
```java
@Configuration
public class AppConfiguration {

 @Bean
 @Scope("prototype")
 public UserService userService(){
  return new UserService();
 }
~~~

```
}
```

In contrast to the other scopes, Spring does not manage the complete lifecycle of a prototype bean: the container instantiates, configures, and otherwise assembles a prototype object, and hands it to the client, with no further record of that prototype instance.

Singleton beans with prototype-bean dependencies

When you use singleton-scoped beans with dependencies on prototype beans, be aware that dependencies are resolved at instantiation time. Thus if you dependency-inject a prototype-scoped bean into a singleton-scoped bean, a new prototype bean is instantiated and then dependency-injected into the singleton bean. The prototype instance is the sole instance that is ever supplied to the singleton-scoped bean.

However, suppose you want the singleton-scoped bean to acquire a new instance of the prototype-scoped bean repeatedly at runtime. You cannot dependency-inject a prototype-scoped bean into your singleton bean, because that injection occurs only once, when the Spring container is instantiating the singleton bean and resolving and injecting its dependencies. If you need a new instance of a prototype bean at runtime more than once, see Method injection

Request, session, application, and WebSocket scopes

The **request, session, application, and websocket** scopes are only available if you use a web-aware Spring ApplicationContext implementation (such as XmlWebApplicationContext). If you use these scopes with regular Spring IoC containers such as the ClassPathXmlApplicationContext, an IllegalStateException will be thrown complaining about an unknown bean scope.

Initial web configuration

To support the scoping of beans at the request, session, application, and websocket levels (web-scoped beans), some minor initial configuration is required before you define your beans. (This initial setup is not required for the standard scopes, singleton and prototype.)

How you accomplish this initial setup depends on your particular Servlet environment.

If you access scoped beans within Spring Web MVC, in effect, within a request that is processed by the Spring DispatcherServlet, then no special setup is necessary: DispatcherServlet already exposes all relevant state.

If you use a Servlet 2.5 web container, with requests processed outside of Spring’s DispatcherServlet (for example, when using JSF or Struts), you need to register the org.springframework.web.context.request.RequestContextListener ServletRequestListener. For Servlet 3.0+, this can be done programmatically via the WebApplicationInitializer interface. Alternatively, or for older containers, add the following declaration to your web application’s web.xml file:

```
<web-app>
    ...
    <listener>
        <listener-class>
            org.springframework.web.context.request.RequestContextListener
        </listener-class>
    </listener>
    ...
</web-app>
```

Alternatively, if there are issues with your listener setup, consider using Spring’s RequestContextFilter. The filter mapping depends on the surrounding web application configuration, so you have to change it as appropriate.

```
<web-app>
    ...
    <filter>
        <filter-name>requestContextFilter</filter-name>
        <filter-class>org.springframework.web.filter.RequestContextFilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>requestContextFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    ...
</web-app>
```

DispatcherServlet, RequestContextListener, and RequestContextFilter all do exactly the same thing, namely bind the HTTP request object to the Thread that is servicing that request. This makes beans that are request- and session-scoped available further down the call chain.

Request scope

Consider the following XML configuration for a bean definition:

```
<bean id="loginAction" class="com.foo.LoginAction" scope="request"/>
```

The Spring container creates a new instance of the LoginAction bean by using the loginAction bean definition for each and every HTTP request. That is, the **loginAction** bean is scoped at the HTTP request level. You can change the internal state of the instance that is created as much as you want because other instances created from the same loginAction bean definition will not see these changes in state; they are particular to an individual request. When the request completes processing, the bean that is scoped to the request is discarded.

When using annotation-driven components or Java Config, the @RequestScope annotation can be used to assign a component to the request scope.

```
@RequestScope
@Component
public class LoginAction {
    // ...
}
```

Session scope

Consider the following XML configuration for a bean definition:

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/>
```

The Spring container creates a new instance of the UserPreferences bean by using the userPreferences bean definition for the lifetime of a single HTTP Session. In other words, the userPreferences bean is effectively scoped at the HTTP Session level. As with request-scoped beans, you can change the internal state of the instance that is created as much as you want, knowing that other HTTP Session instances that are also using instances created from the same userPreferences bean definition do not see these changes in state, because they are particular to an individual HTTP Session. When the HTTP Session is eventually discarded, the bean that is scoped to that particular HTTP Session is also discarded.

When using annotation-driven components or Java Config, the **@SessionScope** annotation can be used to assign a component to the session scope.

```
@SessionScope
@Component
public class UserPreferences {
    // ...
}
```

Application scope

Consider the following XML configuration for a bean definition:

```
<bean id="appPreferences" class="com.foo.AppPreferences" scope="application"/>
```

The Spring container creates a new instance of the AppPreferences bean by using the appPreferences bean definition once for the entire web application. That is, the appPreferences bean is scoped at the ServletContext level, stored as a regular ServletContext attribute. This is somewhat similar to a Spring singleton bean but differs in two important ways: It is a singleton per ServletContext, not per Spring 'ApplicationContext' (for which there may be several in any given web application), and it is actually exposed and therefore visible as a ServletContext attribute.

When using annotation-driven components or Java Config, the **@ApplicationScope** annotation can be used to assign a component to the application scope.

```
@ApplicationScope
```

```
@Component
public class AppPreferences {
    // ...
}
```

WebSocket Scope

The WebSocket Protocol enables two-way communication between a client and a remote host that has opted-in to communicate with the client. WebSocket Protocol provides a single TCP connection for traffic in both directions. This is especially useful for multi-user applications with simultaneous editing and multi-user games.

In this type of web application, HTTP is used only for the initial handshake. The server can respond with HTTP status 101 (switching protocols) if it agrees – to the handshake request. If the handshake succeeds, the TCP socket remains open, and both the client and server can use it to send messages to each other.

When first accessed, *WebSocket* scoped beans are stored in the *WebSocket* session attributes. The same bean instance is then returned during the entire *WebSocket* session.

Please note that `websocket` scoped beans are typically singleton and live longer than any individual WebSocket session.

Autowiring in Spring

Autowiring in the Spring framework can inject dependencies automatically. The Spring container detects those dependencies specified in the configuration file and the relationship between the beans. This is referred to as **Autowiring in Spring**. To enable Autowiring in the Spring application we should use @Autowired annotation. Autowiring in Spring internally uses constructor injection. An autowired application requires fewer lines of code comparatively but at the same time, it provides very little flexibility to the programmer.

Modes of Autowiring

Modes	Description
No	This mode tells the framework that autowiring is not supposed to be done. It is the default mode used by Spring.

Modes	Description
byName	It uses the name of the bean for injecting dependencies.
byType	It injects the dependency according to the type of bean.
Constructor	It injects the required dependencies by invoking the constructor.
Autodetect	The autodetect mode uses two other

Advantage of Autowiring

It requires the **less code** because we don't need to write the code to inject the dependency explicitly.

Disadvantage of Autowiring

No control of programmer.

Annotations

Annotations are a form of metadata that provides data about a program. Annotations are used to provide supplemental information about a program. It does not have a direct effect on the operation of the code they annotate. It does not change the action of the compiled program. So in this article, we are going to discuss what are the main types of annotation that are available in the spring framework with some examples.

Use of java annotations

Java annotations are mainly used for the following:

- Compiler instructions
- Build-time instructions
- Runtime instructions

Compiler instructions: Java provides the 3 in built annotations which are used to give certain instructions to the compiler. Java in built annotation are @Deprecated, @Override & @SuppressWarnings.

Build-time instructions: Java annotations can be used for build time or compile time instructions. These instructions can be used by the build tools for generating source code, compiling the source, generating XML files, packaging the compiled code and files into a JAR file etc.

Runtime instructions: Normally, Java annotations are not present in your Java code after compilation. However, we can define our own annotations that can be available at runtime. These annotations can be accessed using Java Reflection.

Java annotations basics:

A java annotation always starts with the symbol @ and followed by the annotation name. The @symbol signals the compiler that this is an annotation.

Syntax:

```
@AnnotationName
```

Example:

```
@Entity
```

Here @ symbol signals the compiler that this is an annotation and the Entity is the name of this annotation.

An annotation can contain zero, one or multiple elements. We have to set values for these elements. **Example:**

```
@Entity(tableName = "USERS")
```

Where we can use annotations?

We can use java annotations above classes, interfaces, methods, fields and local variables. Here is an example annotation added above a class definition:

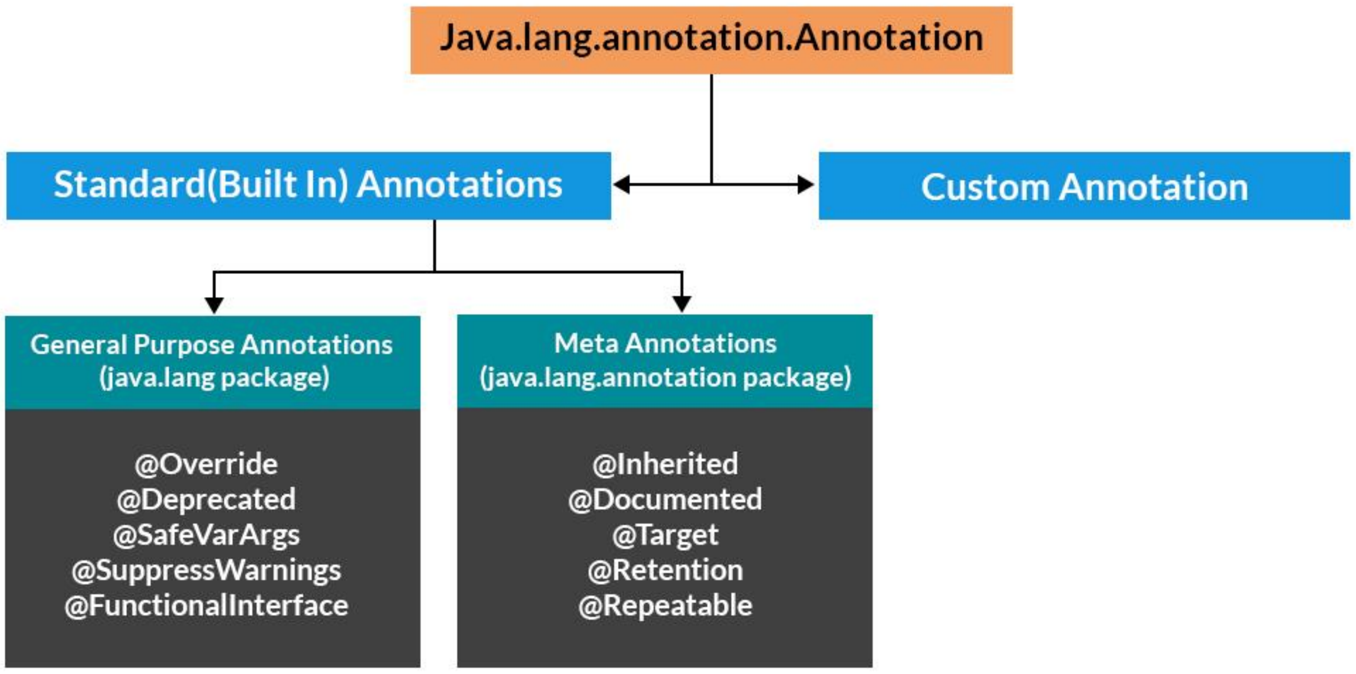
```
@Entity
 public class Users {
}
```

Annotations are used to provide supplemental information about a program.

- Annotations start with ‘@’.

- Annotations do not change the action of a compiled program.
- Annotations help to associate *metadata* (information) to the program elements i.e. instance variables, constructors, methods, classes, etc.
- Annotations are not pure comments as they can change the way a program is treated by the compiler. See below code for example.
- Annotations basically are used to provide additional information, so could be an alternative to XML and Java marker interfaces.

Hierarchy of Annotations in Java



Java.lang.annotation.Annotation
Standard(Built In) Annotations
Custom Annotation
General Purpose Annotations (java.lang package)
@Override
@Deprecated
@SafeVarArgs
@SuppressWarnings
@FunctionalInterface
Meta Annotations (java.lang.annotation package)
@Inherited
@Documented
@Target
@Retention
@Repeatable

Implementation:
Note: *This program throws compiler error because we have mentioned override, but not overridden, we have overloaded display.*

Category 1: Marker Annotations

The only purpose is to mark a declaration. These annotations contain no members and do not consist of any data. Thus, its presence as an annotation is sufficient. Since the marker interface contains no members, simply determining whether it is present or absent is sufficient. **@Override** is an example of Marker Annotation.

Example

```
@TestAnnotation()
```

Category 2: Single value Annotations

These annotations contain only one member and allow a shorthand form of specifying the value of the member. We only need to specify the value for that member when the annotation is applied and don't need to specify the name of the member. However, in order to use this shorthand, the name of the member must be a value.

Example

```
@TestAnnotation(“testing”);
```

Category 3: Full Annotations

These annotations consist of multiple data members, names, values, pairs.

Example

```
@TestAnnotation(owner=”Rahul”, value=”Class Geeks”)
```

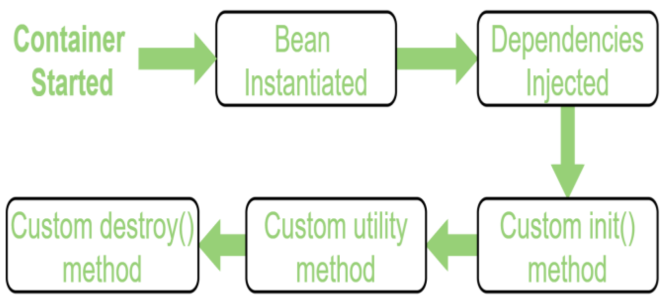
Category 4: Type Annotations

These annotations can be applied to any place where a type is being used. For example, we can annotate the return type of a method. These are declared annotated with ***@Target*** *annotation*.

Spring Bean Life Cycle and Callbacks

The lifecycle of any object means when & how it is born, how it behaves throughout its life, and when & how it dies. Similarly, the bean life cycle refers to when & how the bean is instantiated, what action it performs until it lives, and when & how it is destroyed. In this article, we will discuss the life cycle of the bean.

Bean life cycle is managed by the spring container. When we run the program then, first of all, the spring container gets started. After that, the container creates the instance of a bean as per the request, and then dependencies are injected. And finally, the bean is destroyed when the spring container is closed. Therefore, if we want to execute some code on the bean instantiation and just after closing the spring container, then we can write that code inside the custom **init()** method and the **destroy()** method.
The following image shows the process flow of the bean life cycle.



Bean Life Cycle Process Flow

Note: We can choose a custom method name instead of **init()** and **destroy()**. Here, we will use init() method to execute all its code as the spring container starts up and the bean is instantiated, and destroy() method to execute all its code on closing the container.

1. Spring bean life cycle involves initialization and destruction callbacks and Spring bean aware classes.
   2. Initialization callback methods execute after dependency injection is completed. Their purposes are to check the values that have been set in bean properties, perform any custom initialization or provide a wrapper on original bean etc. Once the initialization callbacks are completed, bean is ready to be used.

3. When IoC container is about to remove bean, destruction callback methods execute. Their purposes are to release the resources held by bean or to perform any other finalization tasks.
4. When more than one initialization and destructions callback methods have been implemented by bean, then those methods execute in certain order.

SPRING CONFIGURATION STYLE

Spring Framework provides three ways to configure beans to be used in the application.

1. **Annotation Based Configuration** - By using @Service or @Component annotations. Scope details can be provided with @Scope annotation.
2. **XML Based Configuration** - By creating Spring Configuration XML file to configure the beans. If you are using Spring MVC framework, the xml based configuration can be loaded automatically by writing some boiler plate code in web.xml file.
3. **Java Based Configuration** - Starting from Spring 3.0, we can configure Spring beans using java programs. Some important annotations used for java based configuration are @Configuration, @ComponentScan and @Bean.

1. Configuring Beans in Spring / Spring Boot

There are multiple ways to configure beans in Spring and Spring Boot:

- **Annotation-based Configuration**: Annotations such as `@Component`, `@Service`, `@Repository`, and `@Controller` are used to mark classes as Spring-managed beans. These annotations can be used to automatically detect and register beans in the Spring application context.
- **Java Configuration**: Spring provides `@Configuration` and `@Bean` annotations to define beans using Java configuration classes.
  - `@Configuration` marks a class as a configuration class.
  - `@Bean` is used to define individual beans.
- **XML Configuration (Spring XML Schema)**: Beans can be defined in XML configuration files using the `<bean>` element. While XML configuration is less common in modern Spring applications, it is still supported for legacy and specific use cases.

Let's explore each method in detail:

2. Annotation-based Configuration [for Stereotype Annotations]

Annotation-based configuration is the most common approach used in modern Spring applications. When Spring loads, Java beans are scanned in the following places:

- All *@Bean* definitions in *@Configuration* annotated classes
- If component scanning is enabled, all stereo-type annotated (such as *@Component*) classes

```
@Configuration

@ComponentScan(basePackages = "com.howtodoinjava.spring")

public class AppConfig {

}
```

When component scanning is enabled, we can define the beans using one of the following annotations as appropriate.

- `@Component`
- `@Repository`
- `@Service`
- `@Controller`
- `@RestController`

```
package com.howtodoinjava.spring.service;



@Service

public class EmployeeManager {





  public Employee create(Employee employee) {

    //...
  }
}
```

Now we can load the beans in context using *AnnotationConfigApplicationContext* as follows:

```java
//Method 1
//ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);

//Method 2
AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
ctx.register(AppConfig.class);
ctx.refresh();

EmployeeManager empManager = ctx.getBean(EmployeeManager.class);
Employee emp = empManager.create();
```

3. Java Configuration [for @Bean Annotation]

Instead of annotating the classes with Spring annotations, we can declare them as Spring bean in the configuration class:

```java
@Configuration

public class AppConfig {



    @Bean

    public EmployeeManager employeeManager() {

        return new EmployeeManager();

    }

}
```

Now we can load this bean into the application context as follows:

```java
//Method 1
//ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);

//Method 2
AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
ctx.register(AppConfig.class);
ctx.refresh();
```

```
EmployeeManager empManager = ctx.getBean(EmployeeManager.class);
Employee emp = empManager.create();
```

4. XML Configuration

XML configuration involves defining beans in XML files using the **`<bean>`** element. While less common than annotation-based and Java configuration, XML configuration is still supported in Spring.

4.1. Bean Definitions

When using XML files, we can define the beans either in a single file or we can distribute the beans in separate files for better code structure. Either way, the bean definitions will be created the same.

```
<?xml version="1.0" encoding="UTF-8"?>

<beans>

  <bean id="operations" class="com.howtodoinjava.core.demo.beans.Operations"></bean>
  <bean id="employee" class="com.howtodoinjava.core.demo.beans.Employee"></bean>
  <bean id="department" class="com.howtodoinjava.core.demo.beans.Department"></bean>

</beans>
```

If we have created multiple bean definition files, we can import the files in the current file as follows:

```
<beans>

  <import resource="employee.xml"/>
  <import resource="department.xml"/>
  <bean id="operations" class="com.howtodoinjava.spring.beans.Operations"></bean>



</beans>
```

4.2. Loading Beans into Context

To load the bean definitions files and thus initialize beans, we can pass the bean definition file name into the constructor of any one of the *ApplicationContext* implementations.

- *ClassPathXmlApplicationContext*
- *FileSystemXmlApplicationContext*
- *XmlWebApplicationContext*

Following is an example of loading the *beans.xml* file into *ClassPathXmlApplicationContext*. Note that the bean definition file is located at `'/src/main/resources/beans.xml'`.

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("classpath:beans.xml");



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

Department department = ctx.getBean(Department.class);

Operations operations = ctx.getBean(Operations.class);
```

Program output:

```
Jan 02, 2018 3:10:27 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions

INFO: Loading XML bean definitions from class path resource [beans.xml]



Jan 02, 2018 3:10:27 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions

INFO: Loading XML bean definitions from class path resource [employee.xml]
```

```
Jan 02, 2018 3:10:27 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions

INFO: Loading XML bean definitions from class path resource [department.xml]
```

5. When to Use Each Method

- **Annotation-based Configuration**: Use this approach for most cases in modern Spring / Spring Boot applications. It's concise, easy to read, and promotes convention over configuration.
- **Java Configuration**: Use Java configuration when you need more control over bean creation, or when you want to use features like conditional bean registration.
- **XML Configuration**: Use XML configuration for legacy applications or when working with frameworks that require XML configuration.

Spring Boot

In Spring Boot, choosing a build system is an important task. We recommend Maven or Gradle as they provide a good support for dependency management. Spring does not support well other build systems.

Dependency Management

Spring Boot team provides a list of dependencies to support the Spring Boot version for its every release. You do not need to provide a version for dependencies in the build configuration file. Spring Boot automatically configures the dependencies version based on the release. Remember that when you upgrade the Spring Boot version, dependencies also will upgrade automatically.

Note − If you want to specify the version for dependency, you can specify it in your configuration file. However, the Spring Boot team highly recommends that it is not needed to specify the version for dependency.

Maven Dependency

For Maven configuration, we should inherit the Spring Boot Starter parent project to manage the Spring Boot Starters dependencies. For this, simply we can inherit the starter parent in our **pom.xml** file as shown below.

```
<parent>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-parent</artifactId>
   <version>1.5.8.RELEASE</version>
</parent>
```

We should specify the version number for Spring Boot Parent Starter dependency. Then for other starter dependencies, we do not need to specify the Spring Boot version number. Observe the code given below −

```
<dependencies>
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
   </dependency>
</dependencies>
```

Gradle Dependency

We can import the Spring Boot Starters dependencies directly into **build.gradle** file. We do not need Spring Boot start Parent dependency like Maven for Gradle. Observe the code given below –

```
buildscript {
   ext {
      springBootVersion = '1.5.8.RELEASE'
   }
   repositories {
      mavenCentral()
   }
   dependencies {
      classpath("org.springframework.boot:spring-boot-gradle-
plugin:${springBootVersion}")
   }
}
```

Similarly, in Gradle, we need not specify the Spring Boot version number for dependencies. Spring Boot automatically configures the dependency based on the version.

```
dependencies {
   compile('org.springframework.boot:spring-boot-starter-
web')
}
```

In Spring Boot, choosing a build system is an important task. We recommend Maven or Gradle as they provide a good support for dependency management. Spring does not support well other build systems.

Dependency Management

Spring Boot team provides a list of dependencies to support the Spring Boot version for its every release. You do not need to provide a version for dependencies in the build configuration file. Spring Boot automatically configures the dependencies version based on the release. Remember that when you upgrade the Spring Boot version, dependencies also will upgrade automatically.

Note − If you want to specify the version for dependency, you can specify it in your configuration file. However, the Spring Boot team highly recommends that it is not needed to specify the version for dependency.

Maven Dependency

For Maven configuration, we should inherit the Spring Boot Starter parent project to manage the Spring Boot Starters dependencies.

We should specify the version number for Spring Boot Parent Starter dependency. Then for other starter dependencies, we do not need to specify the Spring Boot version number.

Gradle Dependency

We can import the Spring Boot Starters dependencies directly into **build.gradle** file. We do not need Spring Boot start Parent dependency like Maven for Gradle.

Spring Boot – Code Structure

There is no specific layout or code structure for Spring Boot Projects. However, there are some best practices followed by developers that will help us too. You can divide your project into layers like service layer, entity layer, repository layer**,**, etc. You can also divide the project into modules. For example, the parent project has two child modules. The first module is for the data layer and the second module is for the web layer. You can also divide the project into features.
Note: *Avoid Default Package*

It is. because a class is said to be in a **default package** when it does not include a **package** declaration. It is not a best practice to include a class in the default package. It is because Spring scans the classes in packages and sub-packages mentioned in the annotations like **@ComponentScan**, **@EntityScan**, **@SpringBootApplication** etc.
Note: *It is recommended to use Java's package naming conventions with a reverse domain name. For example,* ***com.gfg.demo****.*

- Main Application Class

It is recommended to place the Main Application class in the root package with annotations like **@SpringBootApplication** or **@ComponentScan** or **@EnableAutoConfiguration**. It allows the Spring to scan all classes in the root package and sub-packages. For example, if you are writing a JPA application similar to the below layout example, the MainApplicaiton.java is placed in the root package and all

Customer-related classes in the sub-package **com.gfg.demo.customer** and Order related classes in the **com.gfg.demo.order**.
Layout Structure is as follows:

Let us discuss two approaches that are typically used by most developers to structure their spring boot projects.

1. Structure by Feature
2. Structure by Layer

Structure 1: By feature
In this approach, all classes pertaining to a certain feature are placed in the same package. The structure by feature looks is shown in below example

Example

```
com

 +- gfg

     +- demo

         +- MyApplication.java

         |

         +- customer

         |   +- Customer.java

         |   +- CustomerController.java

         |   +- CustomerService.java

         |   +- CustomerRepository.java

         |

         +- order

             +- Order.java

             +- OrderController.java

             +- OrderService.java

             +- OrderRepository.java
```

The advantages of this structure is as follows:

- Find a class to be modified is easy.
- By deleting a particular sub-package, all the classes related to a certain feature can be deleted.
- Testing and Refactoring is easy.
- Features can be shipped separately.

Structure 2: By Layer
Another way to place the classes is by layer i.e; all controllers can be placed in controllers package and services under services package and all entities under domain or model etc.

Example

```
com

 +- gfg
```

```
     +- demo
         +- MyApplication.java
         |
         +- domain
         |   +- Customer.java
         |   +- Order.java
         |
         +- controllers
         |     +- OrderController.java
         |   +- CustomerController.java
         |
         +- services
         |    +- CustomerService.java
         |    +- OrderService.java
         |
         +- repositories
              +- CustomerRepository.java
              +- OrderRepository.java
```

Though the above structure looks feasible and easy to locate classes by a layer. It has few disadvantages when compared to Structure by Feature.

- Features or Modules cannot be shipped separately.
- Hard to locate a class pertaining to a certain feature.
- Code Refactoring on a certain feature is difficult since the feature classes located in every layer.
- It causes merge conflicts among developers using GitHub, BitBucket, etc. for Collaboration.

Now let us wrap up by acquiring the location of MainApplication.java**.** as in both the structures proposed above we have seen that the **MainApplication.java** is placed in the root package with **@SpringBootApplication** annotation. It s as shown below in as a sample example which is as follows:

Example

```
@SpringBootApplication
public class MyApplication {


    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }


}
```

Spring Boot – Runners

What is Spring Boot Runners?

Spring Boot provides two runner interfaces, *ApplicationRunner and CommandLineRunner*. Being **Functional Interfaces,** both the runners have a single functional method, *run()*. When we implement one of these runners, Spring Boot invokes its *run()* method after it starts the context and before the application starts.

That means we can use Spring Boot CommandLineRunner or ApplicationRunner to execute a piece of code when launching an Application or to create a **Spring Boot non-web application**.

Difference Between CommandLineRunner and ApplicationRunner

Overall, both CommandLineRunner and ApplicationRunner are similar, and we can use them to do exact same things.

CommandLineRunner (*Spring Boot Documentation: Link*)

Interface used to indicate that a bean should run when it is contained within a SpringApplication. Multiple CommandLineRunner beans can be defined within the same application context and can be ordered using the Ordered interface or @Order annotation.

ApplicationRunner (*Spring Boot Documentation: Link*)

Interface used to indicate that a bean should run when it is contained within a SpringApplication. Multiple ApplicationRunner beans can be defined within the same application context and can be ordered using the Ordered interface or @Order annotation.

The only difference between the two interfaces is the signature of the *run()* method. The ***run()* method in the *CommandLineRunner* receives the application or program argument as an array of String**.

```
void run(String... args)
```
Code language: Java (java)

However, **the *run()* method in ApplicationRunner receives the program arguments wrapped in an *ApplicationArguments* instance**.
The *ApplicationArguments* provides convenient methods to access the program arguments.

```
void run(ApplicationArguments args)
```
Code language: JavaScript (javascript)

The following sections will teach **how to use both *ApplicationRunner* and *CommandLineRunner* to execute specific code at a Spring Boot Application startup**.

LOGGER

Logging in Spring Boot plays a vital role in Spring Boot applications for recording information, actions, and events within the app. It is also used for monitoring the performance of an application, understanding the behaviour of the application, and recognizing the issues within the application. Spring Boot offers flexible logging capabilities by providing various logging frameworks and also provides ways to manage and configure the logs.

Why to use Spring Boot – Logging?

A good logging infrastructure is necessary for any software project as it not only helps in understanding what's going on with the application but also to trace any unusual incident or error present in the project. This article covers several ways in which logging can be enabled in a spring boot project through easy and simple configurations. Let’s first do the initial setup to explore each option in more depth.

Elements of Logging Framework

- Logger: It captures the messages.
- Formatter: It formats the messages which are captured by loggers.
- Handler: It prints messages on the console, stores them in a file or sends an email, etc.

Java provides several logging frameworks, some of which are:

1. *Logback Configuration logging*
2. *Log4j2 Configuration logging*

Introduction to RESTful Web Services

RESTful Web Services

Last Updated : 22 Dec, 2022

-

REST or Representational State Transfer is an architectural style that can be applied to web services to create and enhance properties like performance, scalability, and modifiability. RESTful web services are generally highly scalable, light, and maintainable and are used to create APIs for web-based applications. It exposes API from an application in a secure and stateless manner to the client. The protocol for REST is HTTP. In this architecture style, clients and servers use a standardized interface and protocol to exchange representation of resources.

REST emerged as the predominant Web service design model just a couple of years after its launch, measured by the number of Web services that use it. Owing to its more straightforward style, it has mostly displaced SOAP and WSDL-based interface design.

REST became popular due to the following reasons:

1. It allows web applications built using different programming languages to communicate with each other. Also, web applications may reside in different environments, like on Windows, or for example, Linux.
2. Mobile devices have become more popular than desktops. Using REST, you don't need to worry about the underlying layer for the device. Therefore, it saves the amount of effort it would take to code applications on mobiles to talk with normal web applications.
3. Modern applications have to be made compatible with the Cloud. As Cloud-based architectures work using the REST principle, it makes sense for web services to be programmed using the REST service-based architecture.

RESTful Architecture:

1. **Division of State and Functionality:** State and functionality are divided into distributed resources. This is because every resource has to be accessible via normal HTTP commands. That means a user should be able to issue the GET request to get a file, issue the POST or PUT request to put a file on the server, or issue the DELETE request to delete a file from the server.
2. **Stateless, Layered, Caching-Support, Client/Server Architecture:** A type of architecture where the web browser acts as the client, and the web server acts as the server hosting the application, is called a client/server architecture. The state of the application should not be maintained by REST. The architecture should also be layered, meaning that there can be intermediate servers between the client and the end server. It should also be able to implement a well-managed caching mechanism.

Principles of RESTful applications:

1. **URI Resource Identification:** A RESTful web service should have a set of resources that can be used to select targets of interactions with clients. These resources can be identified by URI (Uniform Resource Identifiers). The URIs provide a global addressing space and help with service discovery.
2. **Uniform Interface:** Resources should have a uniform or fixed set of operations, such as PUT, GET, POST, and DELETE operations. This is a key principle that differentiates between a REST web service and a non-REST web service.
3. **Self-Descriptive Messages:** As resources are decoupled from their representation, content can be accessed through a large number of formats like HTML, PDF, JPEG, XML, plain text, JSON, etc. The metadata of the resource can be used for various purposes like control caching, detecting transmission errors, finding the appropriate representation format, and performing authentication or access control.
4. **Use of Hyperlinks for State Interactions:** In REST, interactions with a resource are stateless, that is, request messages are self-contained. So explicit state transfer concept is used to provide stateful interactions. URI rewriting, cookies, and form fields can be used to implement the exchange of state. A state can also be embedded in response messages and can be used to point to valid future states of interaction.

Advantages of RESTful web services:

1. **Speed:** As there is no strict specification, RESTful web services are faster as compared to SOAP. It also consumes fewer resources and bandwidth.
2. **Compatible with SOAP:** RESTful web services are compatible with SOAP, which can be used as the implementation.
3. **Language and Platform Independency:** RESTful web services can be written in any programming language and can be used on any platform.
4. **Supports Various Data Formats:** It permits the use of several data formats like HTML, XML, Plain Text, JSON, etc.

Example :

Here is an example of a simple RESTful service that allows a client to create, read, update, and delete (CRUD) a resource :

- Javascript

```
// GET /resource/123

// Returns the state of the resource with ID 123

app.get('/resource/:id', function(req, res) {

  var id = req.params.id;

  var resource = findResourceById(id);
  res.json(resource);

});



// POST /resource

// Creates a new resource with the state specified in the request body

app.post('/resource', function(req, res) {

  var resource = req.body;

  var id = createResource(resource);

  res.json({ id: id });

});



// PUT /resource/123

// Updates the state of the resource with ID 123 with the state
specified in the request body

app.put('/resource/:id', function(req, res) {

  var id = req.params.id;

  var resource = req.body;

  updateResource(id, resource);

  res.sendStatus(200);

});



// DELETE /resource/123

// Deletes the resource with ID 123
```

```
app.delete('/resource/:id', function(req, res) {

  var id = req.params.id;

  deleteResource(id);

  res.sendStatus(200);

});
```

In this example, the service uses the GET, POST, PUT, and DELETE HTTP methods to implement the CRUD operations on a resource. The service uses the app.get(), app.post(), app.put(), and app.delete() methods to register the appropriate handler functions for each operation. The req and res objects represent the request and response, respectively, and are used to access information about the request and send a response to the client.

Spring – REST Controller

Spring Boot is built on the top of the spring and contains all the features of spring. And is becoming a favorite of developers these days because of its rapid production-ready environment which enables the developers to directly focus on the logic instead of struggling with the configuration and setup. Spring Boot is a microservice-based framework and making a production-ready application in it takes very little time. In this article, we will discuss what is REST controller is in the spring boot. There are mainly two controllers are used in the spring, controller and the second one is RestController with the help of **@controller** and **@restcontroller** annotations. The main difference between the @restcontroller and the @controller is that the @restcontroller combination of the @controller and @ResponseBody annotation.
RestController: *RestController is used for making restful web services with the help of the @RestController annotation. This annotation is used at the class level and allows the class to handle the requests made by the client. Let's understand @RestController annotation using an example. The RestController allows to handle all REST APIs such as GET, POST, Delete, PUT requests.*
Spring Initializr is a web-based tool using which we can easily generate the structure of the Spring Boot project. It also provides various different features for the projects expressed in a metadata model. This model allows us to configure the list of dependencies that are supported by JVM. Here, we will create the structure of an application using a spring initializer and then use an IDE to create a sample GET route. Therefore

Spring @RequestMapping Annotation with Example

One of the most important annotations in spring is the **@RequestMapping Annotation** which is used to map HTTP requests to handler methods of MVC and REST controllers. In Spring MVC applications, the DispatcherServlet (Front Controller) is responsible for routing incoming HTTP requests to handler methods of controllers. When configuring Spring MVC, you need to specify the mappings between the requests and handler methods. To configure the mapping of web requests, we use the **@RequestMapping** annotation. The @RequestMapping annotation can be applied to class-level and/or method-level in a controller. The class-level annotation maps a specific request path or pattern onto a controller. You can then apply additional method-level annotations to make mappings more specific to handler methods. So let's understand @RequestMapping Annotation at Method-level and Class level by examples.

Spring Boot is the most popular framework of Java for building enterprise-level web applications and back-ends. Spring Boot has a handful of features that support quicker and more efficient web app development. Some of them are Auto-configuration, Embedded Server, opinionated defaults, and Annotation Support. In this article, we'll be exploring the core annotation of **Spring Boot – @RequestMapping** which is part of the set of annotations that Spring Boot employs for defining URL endpoints and REST APIs.

@RequestMapping

This annotation is a versatile and flexible annotation that can be used with a controller (class) as well as the methods to map specific web requests with the handler methods and controllers. This annotation is part of a larger set of annotations provided by Spring Framework to define URL endpoints and simplify the development of Spring Boot applications.

It has the following features:

- Define several different endpoints to access a specific resource.
- Build REST APIs to serve web requests.
- Simplify the web development process by simply defining an annotation that offers a set of functionalities for handling requests.
- Define multiple endpoints in a single @RequestMapping annotation.

Step-By-Step Implementation of @RequestMapping annotation

For this article, we'll be using the following tools:

- Java 8 or higher
- Java IDE like Eclipse, IntelliJ, and VS code (We'll be using IntelliJ)
- POSTMAN for testing Request Mappings
- Dependency: Spring Web Starter

Step-1: Create a starter file and extract it

Go to Spring Initializr and create a starter file having a single dependency – **Spring Web.** Download and extract it into your local folder and open it in your favorite IDE.

pom.xml File:

- XML

```xml
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <parent>

        <groupId>org.springframework.boot</groupId>

        <artifactId>spring-boot-starter-parent</artifactId>

        <version>3.2.0</version>

        <relativePath/> <!-- lookup parent from repository -->

    </parent>

    <groupId>com.GeeksForGeeks</groupId>

    <artifactId>RequestMappingExample</artifactId>

    <version>0.0.1-SNAPSHOT</version>

    <name>RequestMappingExample</name>

    <description>Request Mapping Example</description>

    <properties>

        <java.version>17</java.version>

    </properties>

    <dependencies>

        <dependency>
```

```
            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-web</artifactId>

        </dependency>



        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-test</artifactId>

            <scope>test</scope>

        </dependency>

    </dependencies>



    <build>

        <plugins>

            <plugin>

                <groupId>org.springframework.boot</groupId>

                <artifactId>spring-boot-maven-plugin</artifactId>

            </plugin>

        </plugins>

    </build>



</project>
```

Step-2: Define a Controller

- Create a new package for containing all the controllers that we will be adding in our application under the **src/main/java/package_name/Controllers**

- Add a new **TestController** inside **Controllers** Package for defining Request Mappings. Your final directory structure would look something like this :

Step-4: Define URL Templates inside the controller

Annotate the Controller with @Controller to signify that this class is a controller that has some URL templates defined inside it and @RequestMapping on the controller as well as the methods to specify which URL path will give what output.

Below is the code Implementation of Controller:

- Java

```
import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.ResponseBody;



import java.util.ArrayList;

import java.util.List;



// Controller class

@Controller

@ResponseBody

@RequestMapping("/test")

public class TestController {



    // URL Path - 1

    // Returns a String

    @RequestMapping("/hello")

    public String sayHello() {
        System.out.println("dsvsdvdsvs");

        return "Hello Geek!";

    }



    // URL Path - 2

    // Returns a String

    @RequestMapping("/sport")

    public String doSomeSport() {

        return "Run 5 kilometers today!";

    }



    // URL Path - 3

    // Returns a List

    @RequestMapping("/today/tasks")

    public List<String> todaysTasks() {



        List<String> myTasks = new ArrayList<String>();

        myTasks.add("Write 5 articles on GeeksforGeeks Today");

        myTasks.add("Run 5 kilometers");

        myTasks.add("Do Laundry");



        return myTasks;

    }
```

```
}
```

Explanation of the above Program:

We've defined multiple end points using @RequestMapping which can be accessed at following URLs :

- http://localhost:8080/test/hello
- http://localhost:8080/test/sport
- http://localhost:8080/test/today/tasks

Annotations used:

- **@Controller**: This annotation implicitly marks the class as a component making it eligible for component scanning while signifying that this class invokes business logic, handles incoming web requests and returns customized responses.
- **@ResponseBody**: This annotation is used on both class level as well as on method level to indicate that the return value generated by this class or method doesn't need to be resolved to a view page like HTML or JSP, rather it should be directly serialized into HTTP response body. And this serialization from Java Object to JSON is done by a technology called – Jackson Project that performs Jackson Data Binding under the hood.

Our request method can return any type of data – POJOs (Plain Old Java Objects) and this data will be serialized into JSON by the Jackson Project.

Note: *@ResponseBody can be ignored if you're using @RestController instead of @Controller.*

Outputs for every endpoint on POSTMAN:

Endpoint – 1: http://localhost:8080/test/hello

Endpoint – 2: http://localhost:8080/test/sport

Endpoint – 3: http://localhost:8080/test/today/tasks

Notice that we're selecting the type of request as GET in our POSTMAN this is because all of our methods are only returning some data, not making any changes in a database (as we're not connected to any database).

And to be more specific, the actual use of @RequestMapping is to define the start of URL and the request handler method will perform some operation and depending on the type of operation a method performs, we will annotate it with GET, PUT, POST or DELETE. Let's look at some of the most commonly used type of annotation we can use with a specific web request in brief:

- **@GetMapping:** This annotation is associated with the requests that are requesting a resource without making any changes in the database.
- **@PostMapping:** This annotation is associated with the requests that are trying to add new data in the database. Ex: Adding a new student record that contains all the information regarding a student like student_id, name, enrollment_number, branch etc.
- **@PutMapping:** This annotation is associated with the update requests that wants to update or change some already existing data in our database.
- **@DeleteMapping:** This annotation is associated with deleting persistent objects (already existing data) from our database.

Difference Between @RequestBody and @ResponseBody Annotation in Spring

Last Updated : 17 Dec, 2023

To achieve the functionality of handling request data and response data in Spring MVC, @RequestBody and @ResponseBody annotations are used. So, in this article, we will go dive into the **difference between @RequestBody and @ResponseBody annotations** with an example.

@RequestBody

- @RequestBody is mainly used with CRUD Operations to read the request body.

Example:

```
@PostMapping("/addStudent")
public void AddStudent(@RequestBody Student student) {
  // body
}
```

@ResponseBody

- @ResponseBody is typically used with GET methods to write the response body content.

Example:

```
@GetMapping("/getStudent")
@ResponseBody
public Student getStudent()
{
    return student;
}
```

Difference between @RequestBody and @ResponseBody

Paramaters	@RequestBody	@ResponseBody
Purpose	Applicable for the incoming request data.	Applicable for the outgoing response data.
Method body	Used with POST, PUT, PATCH methods to read the request body.	Used with GET methods to write the response body.
Return value	Typically void or a simple type	Typically, a complex object representing the response data
Object	The deserialized object is passed as a method parameter.	The serialized object is returned from the method.
Payloads	Required to read JSON/XML request payloads.	Required to write JSON/XML response payloads.

Example of @RequestBody and @ResponseBody Annotation:

Step 1: Set up a new Spring MVC project

- Create a new Maven project in your preferred IDE (e.g., IntelliJ or Eclipse or Spring Tool Suite) and add the following dependencies.
    - Spring web
    - Spring Data JPA
    - MySQL Driver

- XML

PathVariable

Java language is one of the most popular languages among all programming languages. There are several advantages of using the java programming language, whether for security purposes or building large distribution projects. One of the advantages of using JAVA is that Java tries to connect every concept in the language to the real world with the help of the concepts of classes, inheritance, polymorphism, etc.

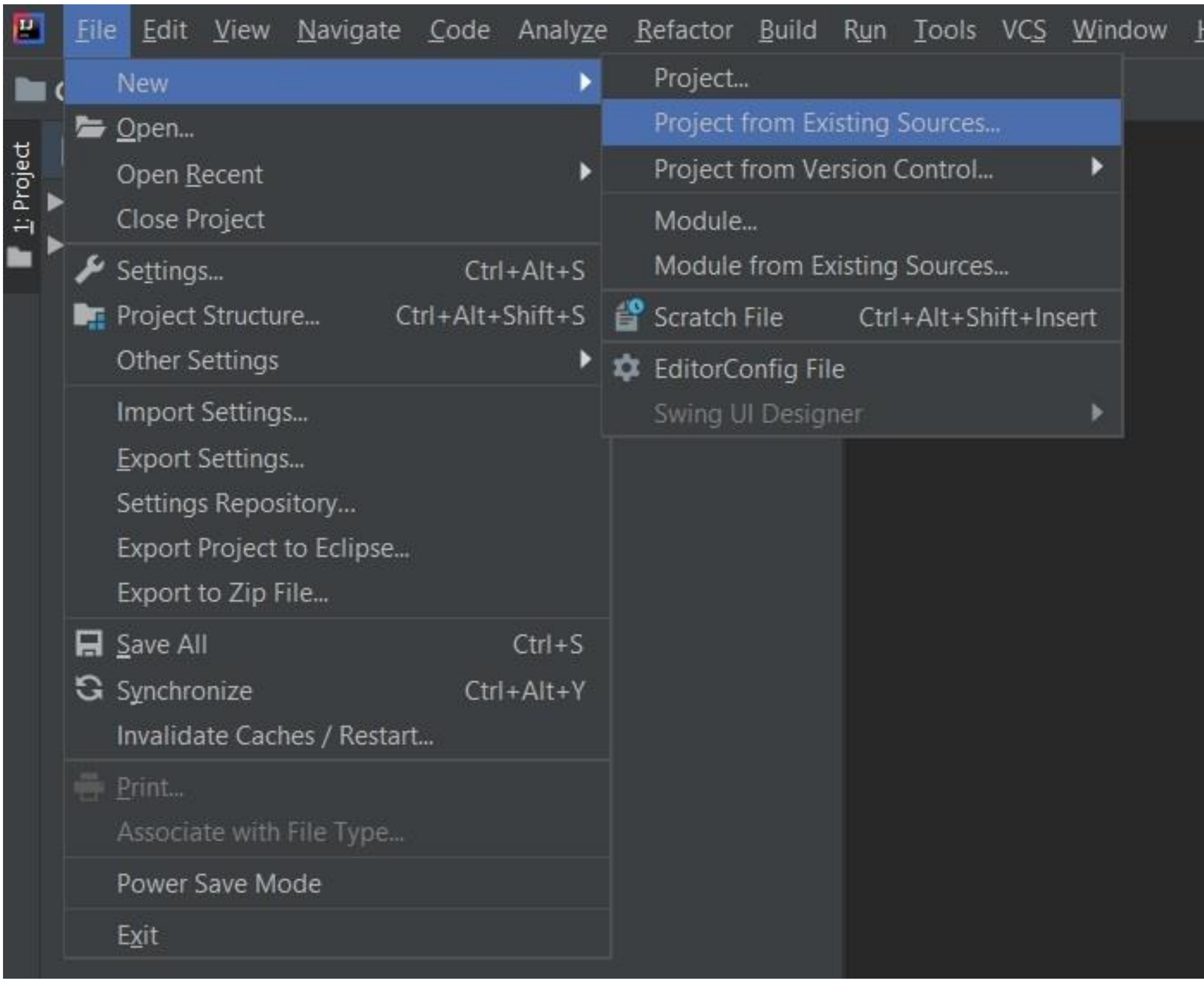
There are several other concepts present in java that increase the user-friendly interaction between the java code and the programmer such as generic, access specifiers, annotations in java, etc these features add an extra property to the class as well method of the java program. In this article, we will discuss what is path variable is in the spring boot.
Path variable in the spring boot represents different kinds of parameters in the incoming request with the help of @pathvariable annotation.
Java language is one of the most popular languages among all programming languages. There are several advantages of using the java programming language, whether for security purposes or building large distribution projects. One of the advantages of using JAVA is that Java tries to connect every concept in the language to the real world with the help of the concepts of classes, inheritance, polymorphism, etc.

There are several other concepts present in java that increase the user-friendly interaction between the java code and the programmer such as generic, access specifiers, annotations in java, etc these features add an extra property to the class as well method of the java program. In this article, we will discuss what is path variable is in the spring boot.
Path variable in the spring boot represents different kinds of parameters in the incoming request with the help of @pathvariable annotation.
Note: *First we need to establish the spring application in our project.*
Spring Initializr is a web-based tool using which we can easily generate the structure of the Spring Boot project. It also provides various different features for the projects expressed in a metadata model. This model allows us to configure the list of dependencies that are supported by JVM. Here, we will create the structure of an application using a spring initializer and then use an IDE to create a sample GET route. Therefore, to do this, the following steps are followed:
Step 1: Go to Spring Initializr
Step 2: Fill in the details as per the requirements. For this application:

```
Project: Maven
```

```
Language: Java
Spring Boot: 2.2.8
Packaging: JAR
Java: 8
Dependencies: Spring Web
```

Step 3: Click on Generate which will download the starter project.
Step 4: Extract the zip file. Now open a suitable IDE and then go to File->New->Project from existing sources->Spring-boot-app and select pom.xml. Click on import changes on prompt and wait for the project to sync as pictorially depicted below as follows:

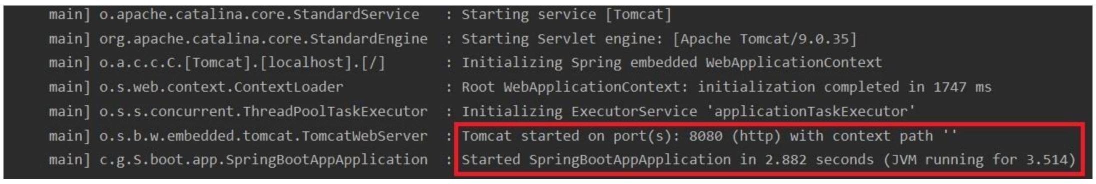


Note: *In the Import Project for Maven window, make sure you choose the same version of JDK which you selected while creating the project.*
Step 5: Go to src->main->java->com.gfg.Spring.boot.app, create a java class with the name Controller and add the annotation @RestController. Now create a GET API as shown below:
Example 1: Controller.java

```
@RestController


// Class

public class Controller {
    @GetMapping("/hello/{name}/{age}")

    public void insert(@PathVariable("name") String name,
                       @PathVariable("age") int age) {

        // Print and display name and age
        System.out.println(name);
        System.out.println(age);
    }
}
```

This application is now ready to run.

Step 6: Run the SpringBootAppApplication class and wait for the Tomcat



```
main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
main] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.35]
main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
main] o.s.web.context.ContextLoader            : Root WebApplicationContext: initialization completed in 1747 ms
main] o.s.s.concurrent.ThreadPoolTaskExecutor  : Initializing ExecutorService 'applicationTaskExecutor'
main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
main] c.g.S.boot.app.SpringBootAppApplication  : Started SpringBootAppApplication in 2.882 seconds (JVM running for 3.514)
```

server to start.

Note: *The default port of the Tomcat server is 8080 and can be changed in the application.properties file.*

Step 7: Lastly now go to the browser and enter the URL localhost:8080. Observe the output and now do the same for localhost:8080/hello/Aayush/23

Output:

```
Aayush

23
```

RequestParam

The @RequestParam annotation is used to extract data from the query parameters in the request URL. Query parameters are the key-value pairs that appear after the ? in a URL. Let’s consider an example where you have a REST API endpoint for searching users based on a query parameter:

```
@RestController

@RequestMapping("/users")
```

```
public class UserController {



    @GetMapping("/search")

    public ResponseEntity<List<User>> searchUsers(@RequestParam("name")
String name) {

        // Implementation to search users based on the provided name

        // ...

        return ResponseEntity.ok(users);

    }

}
```

In the above code, the @RequestParam annotation is used to extract the name parameter from the query parameters. The name parameter is specified as an argument of the @RequestParam annotation. When a request is made to /users/search?name=John, the value John will be passed to the searchUsers method as the name parameter. You can then use this parameter to perform a search operation based on the provided name.

You can also specify optional parameters by setting the required attribute of @RequestParam to false. Additionally, you can provide default values using the defaultValue attribute. Here's an example that demonstrates how to specify optional parameters using the @RequestParam annotation in Spring Boot and provide default values using the defaultValue attribute:

RestTemplate APIs:

- **getForObject** - Retrieves a representation via GET.
- **getForEntity** - Retrieves a ResponseEntity (that is, status, headers, and body) by using GET.
- **headForHeaders** - Retrieves all headers for a resource by using HEAD.
- **postForLocation** - Creates a new resource by using POST and returns the Location header from the response.
- **postForObject** - Creates a new resource by using POST and returns the representation from the response.
- **postForEntity** - Creates a new resource by using POST and returns the representation from the response.
- **put** - Creates or updates a resource by using PUT.
- patchForObject - Updates a resource by using PATCH and returns the representation from the response. Note that the JDK HttpURLConnection does not support the PATCH, but Apache HttpComponents and others do.
- **delete** - Deletes the resources at the specified URI by using DELETE.
- **optionsForAllow** - Retrieves allowed HTTP methods for a resource by using ALLOW.
- **exchange** - A more generalized (and less opinionated) version of the preceding methods that provides extra flexibility when needed. It accepts a RequestEntity (including HTTP method, URL, headers, and body as input) and returns a ResponseEntity.
- **execute** - The most generalized way to perform a request, with full control over request preparation and response extraction through callback interfaces.

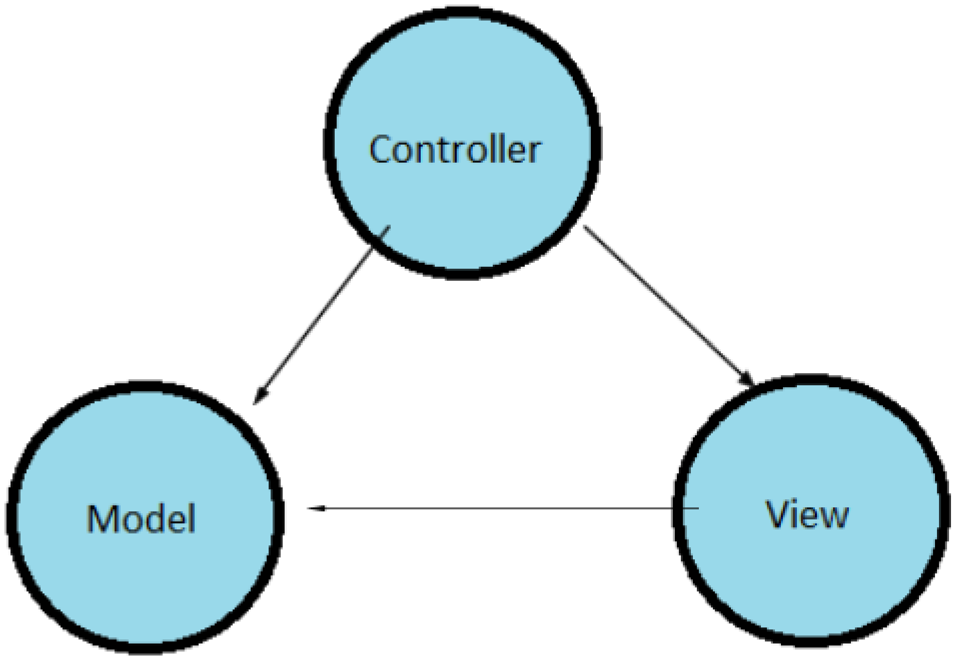
Let's create a *SpringRestClient* class and add the following content to it:

```
import java.util.Arrays;
import java.util.HashMap;
import java.util.Map;

import org.springframework.http.HttpEntity;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpMethod;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.client.RestTemplate;

public class SpringRestClient {

        private static final String GET_EMPLOYEES_ENDPOINT_URL =
"http://localhost:8080/api/v1/employees";
        private static final String GET_EMPLOYEE_ENDPOINT_URL =
"http://localhost:8080/api/v1/employees/{id}";
        private static final String CREATE_EMPLOYEE_ENDPOINT_URL =
"http://localhost:8080/api/v1/employees";
        private static final String UPDATE_EMPLOYEE_ENDPOINT_URL =
"http://localhost:8080/api/v1/employees/{id}";
        private static final String DELETE_EMPLOYEE_ENDPOINT_URL =
"http://localhost:8080/api/v1/employees/{id}";
        private static RestTemplate restTemplate = new RestTemplate();

        public static void main(String[] args) {
                SpringRestClient springRestClient = new SpringRestClient();
                // Step1: first create a new employee
                springRestClient.createEmployee();

                // Step 2: get new created employee from step1
                springRestClient.getEmployeeById();

                // Step3: get all employees
                springRestClient.getEmployees();

                // Step4: Update employee with id = 1
                springRestClient.updateEmployee();

                // Step5: Delete employee with id = 1
                springRestClient.deleteEmployee();
        }

        private void getEmployees() {

                HttpHeaders headers = new HttpHeaders();
                headers.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
                HttpEntity<String> entity = new HttpEntity<String>("parameters",
headers);

                ResponseEntity<String> result =
restTemplate.exchange(GET_EMPLOYEES_ENDPOINT_URL, HttpMethod.GET, entity,
                                String.class);

                System.out.println(result);
        }

        private void getEmployeeById() {

                Map<String, String> params = new HashMap<String, String>();
                params.put("id", "1");

                RestTemplate restTemplate = new RestTemplate();
                Employee result = restTemplate.getForObject(GET_EMPLOYEE_ENDPOINT_URL,
Employee.class, params);

                System.out.println(result);
        }

        private void createEmployee() {

                Employee newEmployee = new Employee("admin", "admin", "admin@gmail.com");

                RestTemplate restTemplate = new RestTemplate();
                Employee result =
restTemplate.postForObject(CREATE_EMPLOYEE_ENDPOINT_URL, newEmployee, Employee.class);

                System.out.println(result);
        }

        private void updateEmployee() {
                Map<String, String> params = new HashMap<String, String>();
                params.put("id", "1");
                Employee updatedEmployee = new Employee("admin123", "admin123",
"admin123@gmail.com");
                RestTemplate restTemplate = new RestTemplate();
                restTemplate.put(UPDATE_EMPLOYEE_ENDPOINT_URL, updatedEmployee, params);
        }

        private void deleteEmployee() {
                Map<String, String> params = new HashMap<String, String>();
                params.put("id", "1");
                RestTemplate restTemplate = new RestTemplate();
                restTemplate.delete(DELETE_EMPLOYEE_ENDPOINT_URL, params);
        }
}
```

Introduction

We'll be building a dashboard that displays the statistics about the Covid-19. Now there are a lot of blogs and tutorials about what is Spring and why is it preferred in the Java world for building REST APIs and Web Applications, so this article will focus on how to build a web application and deploy it on Heroku.

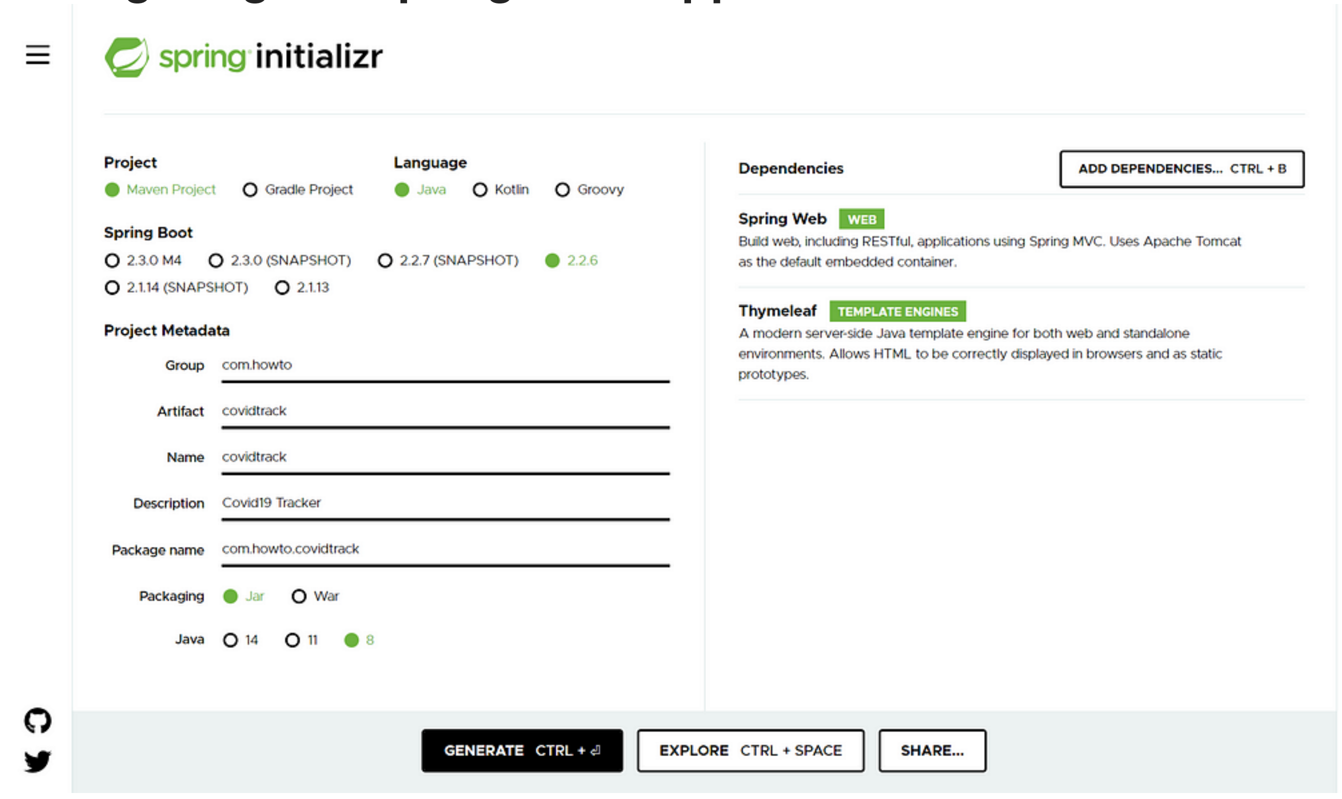
We will use the concept of MVC, i.e., Model-View-Controller. MVC is an architecture that separates various components of an application like the Input Logic, Business Logic, and UI Logic. The views and the models don't interact with each other. The controller receives the request from the view and gets the required details from the model and transfers it to the view.



The API we'll be using for the data is Covid19India API. The API has the daily data of Covid19 cases in India and also gives the updated list of cases in each state.

Let's get started!

Configuring the Spring Boot Application



Project
[✓ Maven Project] Gradle Project

Language
[✓ Java] Kotlin Groovy

Spring Boot
2.3.0 M4 2.3.0 (SNAPSHOT) 2.2.7 (SNAPSHOT) [✓ 2.2.6] 2.1.14 (SNAPSHOT) 2.1.13

Project Metadata

Group	com.howto
Artifact	covidtrack
Name	covidtrack
Description	Covid19 Tracker
Package name	com.howto.covidtrack
Packaging	[✓ Jar] War
Java	14 11 [✓ 8]

Dependencies

Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Thymeleaf TEMPLATE ENGINES
A modern server-side Java template engine for both web and standalone environments. Allows HTML to be correctly displayed in browsers and as static prototypes.

We can get the basic configuration files and the boilerplate from **Spring Initializer**. We name our application and select the appropriate **java** version. We'll be using Java 11 in this application. Next, we add few dependencies which provide the core features. Spring Web provides us with all the required libraries and classes for creating RESTful applications and for the server for our application.

Thymeleaf is a template engine which we'll be using to communicate from the backend to the HTML. We hit the generate button and we'll have a zip file downloaded. We can open the folder in Intellij.

Our first step is to create an HTML file in /src/main/resources/templates folder called “index.html”. This HTML file will be the landing and home page of our application.