**INTRODUCTION**

**What is Spring?**

This lesson gives a quick overview of Spring as a dependency injection framework.

**We'll cover the following**

* [Tight coupling](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qA0AvkLGgEG#Tight-coupling)
* [Loose coupling](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qA0AvkLGgEG#Loose-coupling)

The Spring framework is an open-source Java application framework, which is based on two key principles: **dependency injection** and **Inversion of Control**. Spring has the ability to autowire the dependency at run time, which allows the developer to write *loosely coupled* code.

Spring framework uses metadata in the form of xml file or java annotations to create objects and identifies dependencies, thereby producing a ready-to-use application.

Spring creates a ready-to-use application

A typical web application is divided into three layers: web, business, and the data layer. These layers have objects that collaborate with each other to make the application work. These collaborations are called *dependencies*. A typical application has a lot of classes and dependencies.

**Tight coupling**

*Tightly coupled* code involves creating an instance of the dependency inside the class. As an example, suppose we have an application that recommends movies to watch. The application uses content-based filtering that employs item-to-item similarity as well as user preferences. The class MovieRecommender is directly instantiating an object of ContentBasedFilter, which makes ContentBasedFilter a dependency of MovieRecommender.

 public class MovieRecommender {  
    ContentBasedFilter filter = new ContentBasedFilter();  
    //...  
 }

 public class ContentBasedFilter {  
    //...  
 }

Problems can arise when we want to use a different option for the dependency. Suppose we did not get good movie recommendations from the content-based filter and want to switch to a collaborative filter which takes into account the choices of users who have watched similar movies. This entails changing the code of MovieRecommender, which would be a disadvantage of using tightly coupled code.

**Loose coupling**

A better way would be to implement an interface. This will remove the direct instantiation of the ContentBasedFilter, and instead, ask for the type of filter as an argument to the constructor.

interface Filter {  
    //method declarations  
}

public class ContentBasedFilter implements Filter {  
    //implement interface methods  
}

public class MovieRecommender {  
  
    Filter filter;  
  
    public MovieRecommender(Filter filter) {  
        this.filter = filter;  
    }  
  
    //...  
}

This way MovieRecommender is not dependent on a specific type of filter and can be used with both a content-based filter and a collaborative filter. The above code snippet is an example of loosely coupled code. Loose coupling has a number of advantages.

public static void main(String[] args) {  
  
   MovieRecommender recommender = new MovieRecommender(new ContentBasedFilter());  
    //...  
}

Here, we have created an object of ContentBasedFilter class implementing the Filter interface and an object of MovieRecommender class. We have injected the ContentBasedFilter object into the MovieRecommender object. The Spring framework writes the above code on its own. *Spring creates objects and populates dependencies*. As a programmer, you only have to tell which objects it has to create and what the dependencies of each object are.

Spring takes control of populating the dependencies and injecting the ContentBasedFilter object into the MovieRecommender object. This is in contrast to the approach shown in the first code snippet where MovieRecommender instantiated the ContentBasedFilter object itself. Spring inverts the control by taking responsibility for populating the dependency. This is referred to as *Inversion of Control (IoC)*.

To summarize, Spring is a dependency injection framework that promotes loosely coupled code.

**History**

Walk through the history of major developments in the enterprise application development and releases of the Spring framework.

**We'll cover the following**

* [The rise of Spring](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7ANnW24NOjG#The-rise-of-Spring)
* [Development timeline](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7ANnW24NOjG#Development-timeline)

Spring is a framework for enterprise development. It is similar to J2EE but is way more popular because of its helper classes, which make programming a breeze.

**The rise of Spring**

J2EE has been around since 1999, but it earned a bad reputation because of its use of Enterprise Java Beans (EJB), which were complicated to use and deteriorated the application’s performance. Other issues with J2EE included the need to write extensive plumbing code for error handling and JDBC connectivity, and the lack of unit testing support. The applications were heavyweight because all the Java EE features were supported and had to be configured and all dependencies had to be hard-coded as well.

In 2002, Rod Johnson wrote a book outlining a framework that simplified EE development. It was initially called Interface21 before being released as an open source framework. When Spring made its appearance in 2003, programmers were using J2EE without Java Beans. Spring gained immense popularity in a short amount of time and captured most of the J2EE market share. The EJB functionality was improved in Java EE5 and Java EE6, with the focus being on ease of use. The current version, Java EE8, is similar to Spring, and both frameworks can be used to achieve the same results.

Spring and Java EE timeline

**Development timeline**

Spring has gained a lot of momentum since the launch of Spring 1.0 in 2004. By the time Spring 2.0 was released in 2006, Spring had surpassed 1 million downloads! Spring 2.0 removed complexities from XML config files. The next version, Spring 2.5, introduced annotation configurations. Spring 3.0, released in 2009, set the minimum Java requirement to Java 5 and provided built-in REST support. Spring 3.2 introduced Java configuration and got rid of web.xml. In 2012, the Spring Mobile and Spring Android projects were released. In 2013, Spring 4.0 came around. This was the same year Spring Boot 1.0 was introduced to remove complexities associated with creating a web application. Spring Boot configures all the required Spring components and sets up dependencies in Gradle/Maven along with configuring the required beans using either xml, annotations, or Java code. The current version of the framework, Spring 5.0 came out in 2017. It sets the minimum Java requirement to Java 8.

Spring has simplified application development and its ease of use has led to widespread adoption of the framework. Spring Boot enables users to create a project with all required dependencies automatically wired in.

**Terminology**

Learn the different terms used in the world of Spring.

**We'll cover the following**

* [Beans](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B8pAzvx7DXX#Beans)
* [Autowiring](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B8pAzvx7DXX#Autowiring)
* [Dependency injection](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B8pAzvx7DXX#Dependency-injection)
* [Inversion of Control](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B8pAzvx7DXX#Inversion-of-Control)
* [IoC container](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B8pAzvx7DXX#IoC-container)
* [Bean factory](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B8pAzvx7DXX#Bean-factory)
* [Application context](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B8pAzvx7DXX#Application-context)

**Beans**

Beans are the objects of classes that are managed by Spring. Traditionally, objects used to create their own dependencies, but Spring manages all the dependencies of an object and instantiates the object after injecting the required dependencies. The **@Component** annotation is the most common method of defining beans.

@Component  
public class Vehicle {  
  
}

**Autowiring**

The process of identifying a dependency, looking for a match, and then populating the dependency is called autowiring. The **@Autowired** annotation tells Spring to find and inject a collaborating bean into another. If more than one bean of the same type is available, Spring throws an error. In the following scenario, two beans of type Operator are detected by Spring:

@Component  
class Arithmetic(){  
    @Autowired  
    private Operator operator;  
    //...  
}  
  
@Component  
class Addition implements Operator {  
  
}  
  
@Component  
class Subtraction implements Operator {  
  
}

Spring will not know which bean to inject in the Arithmetic bean unless the developer explicitly specifies it.

**Dependency injection**

Dependency injection is the process by which Spring looks up the beans that are needed for a particular bean to function and injects them as a dependency. Spring can perform dependency injection by using constructor or by using a setter method.

**Inversion of Control**

Traditionally, the class which needed the dependency created an instance of the dependency. The class decided when to create the dependency and how to create it. For example, Engine class is a dependency of Vehicle class, which creates its object:

class Vehicle{  
    
    private Engine engine = new Engine();  
    //...  
}

Spring takes this responsibility from the class and creates the object itself. The developer simply mentions the dependency and the framework takes care of the rest.

class Vehicle{  
   
    private Engine engine;  
    //...  
}

Thus, control moves from the component that needs the dependency to the framework. The framework takes the responsibility for finding out the dependencies of a component, ensuring their availability and injecting them in the component. This process is called Inversion of Control.

Traditional approach vs Inversion of Control

**IoC container**

An IoC container is a framework that provides the *Inversion of Control* functionality.

The IoC container manages the beans. For the above mentioned example, it creates an instance of the Engine class, then creates an instance of Vehicle class, and then injects the Engine object as a dependency into the Vehicle object.

class Vehicle {  
    private Engine engine;  
    //...     
}

IoC container is a generic term. It is not framework specific. Spring offers two implementations of the IoC container:

1. Bean factory
2. Application context

Implementations of IoC

Both of them are interfaces that have different implementations available. Application context is the typical IoC container in the context of Spring. Spring recommends using it unless there is a memory concern, like in a mobile device. If available memory is low, bean factory should be used.

**Bean factory**

The basic version of the Spring IoC container is bean factory. It is the legacy IoC container and provides basic management for beans and wiring of dependencies. In Spring, bean factory still exists to provide backward compatibility.

**Application context**

Application context adds more features to the bean factory that are typically needed by an enterprise application. It is the most important part of the Spring framework. All the core logic of Spring happens here. It includes basic management of beans and wiring of dependencies as provided by the bean factory. Additional features in application context include Spring AOP features, internationalization, web application context, etc.

# Spring Architecture

This lesson sheds light on the modular architecture of Spring and also discusses popular Spring projects.

**We'll cover the following**

* [Spring modules](https://www.educative.io/courses/guide-spring-5-spring-boot-2/g7z68gP1jr3#Spring-modules)
  + [Data access/ integration](https://www.educative.io/courses/guide-spring-5-spring-boot-2/g7z68gP1jr3#Data-access/-integration)
  + [Web (MVC/remoting)](https://www.educative.io/courses/guide-spring-5-spring-boot-2/g7z68gP1jr3#Web-(MVC/remoting))
  + [Test](https://www.educative.io/courses/guide-spring-5-spring-boot-2/g7z68gP1jr3#Test)
  + [AOP](https://www.educative.io/courses/guide-spring-5-spring-boot-2/g7z68gP1jr3#AOP)
* [Spring projects](https://www.educative.io/courses/guide-spring-5-spring-boot-2/g7z68gP1jr3#Spring-projects)

Spring is not one big framework. It is broken down into modules. This can be seen in the Maven Dependencies folder, where there are a lot of JAR files instead of just one big JAR.

JAR files in the Maven Dependencies folder

Spring is built in a modular way and this enables some modules to be used without using the whole framework. It also makes integration with other frameworks easy. The developer can choose which module to use and discard ones that are not required.

## Spring modules

The modules of Spring architecture, grouped together in layers, are shown below:

The Core Container contains the following modules: Beans, Core, Context, and Spring Expression Language (SpEL). These modules provide fundamental functionality of the Spring framework, like Inversion of Control (IoC), dependency injection, internationalization as well as support for querying the object at run time.

### Data access/ integration

Spring has very good integration with data and integration layers, and provides support to interact with databases. It contains modules like JDBC, ORM, OXM, JMS, and Transactions.

* The **JDBC** (Java Database Connectivity) module allows the data layer to interact with databases to get data or store data, or to interact with other systems without the need of cumbersome JDBC coding. Spring JDBC is very straightforward as compared to plain JDBC and makes the code very short.
* The **ORM** (Object Relational Mapping) module provides support to integrate with ORM frameworks including Hibernate and JPA.
* The **JMS** (Java Messaging Service) module talks to other applications through the queue to produce and consume messages.
* The **OXM** (object-XML mapping) module makes the object-to-XML transformation easy by providing useful features.
* The transaction management module provides support for successful rollback in case a transaction fails.

### Web (MVC/remoting)

It contains the Web, Servlets, Portlets, and Sockets modules to support the creation of a web application. Spring offers a web framework of its own called Spring MVC.

### Test

The Test module handles the cross cutting concern of unit testing. The Spring Test framework supports testing with JUnit, TestNG, as well as creating mock objects for testing the code in isolation.

### AOP

The AOP module provides Aspect Oriented Programming functionality like method interception and pointcuts as well as security and logging features. Spring has its own module called Spring AOP that offers basic, aspect-oriented programming functionality. Advanced AOP functionality can be implemented through integration with AspectJ. AOP features cross cutting concerns from business logic.

## Spring projects

Spring also provides solutions to different enterprise application problems through Spring projects. Some of them are discussed below:

Spring projects

**Spring Boot** is used to develop micro services. It makes developing applications easy through features like startup projects, auto configuration, and actuator. Spring Boot has gained massive popularity since it was first released in 2014.

**Spring Cloud** allows the development of cloud native applications that can be dynamically configured and deployed. It provides functionality for handling common patterns in distributed systems.

**Spring Data** provides consistent access to SQL and NoSQL databases.

**Spring Integration** implements the patterns outlined by the book Enterprise Application Integration Patterns. It allows enterprise applications to be connected easily through messaging and declarative adapters.

**Spring Batch** provides functionality to handle large volumes of data like ability to restart, ability to read from and write to different systems, chunk processing, parallel processing, and transaction management.

**Spring Security** provides security solutions for different applications be it a web application or a REST service. It also provides authentication and authorization features.

**Spring Session** manages session information and makes it easier to share session data between services in the cloud regardless of the platform/container. It also supports multiple sessions in a single browser instance.

**Spring Mobile** offers device detection and progressive rendering options that make mobile web application development easy.

**Spring Android** facilitates the development of Android applications.

**Reasons for Sustained Popularity**

Let's look at some factors of Spring that led to its widespread adoption and sustained popularity.

**We'll cover the following**

* [Flexibility and integration with other frameworks](https://www.educative.io/courses/guide-spring-5-spring-boot-2/mypp9z3ANR3#Flexibility-and-integration-with-other-frameworks)
* [Removes plumbing code](https://www.educative.io/courses/guide-spring-5-spring-boot-2/mypp9z3ANR3#Removes-plumbing-code)
* [Promotes testable code](https://www.educative.io/courses/guide-spring-5-spring-boot-2/mypp9z3ANR3#Promotes-testable-code)
* [Staying up to date](https://www.educative.io/courses/guide-spring-5-spring-boot-2/mypp9z3ANR3#Staying-up-to-date)

**Flexibility and integration with other frameworks**

Spring has a very flexible architecture. Spring modules are not dependent on one another and offer a developer the freedom to pick and choose according to the requirements of the application. Spring projects are designed with very specific purposes in mind.

Spring offers integration with a large number of frameworks. For example, even though Spring offers its own MVC framework, SpringMVC, it also offers integration with other MVC frameworks. Using Spring does not decrease the developer’s options.

**Removes plumbing code**

Plumbing code not only makes programming longer but also reduces the readability of code. For example, in JDBC programming, a lot of code is required for simple functionality. Connection establishment and exception handling span many lines of code.

Spring removes plumbing code and lets the programmer focus on the application logic. The amount of code written in Spring is negligible. No exception handling code is required because Spring makes all its exceptions unchecked.

**Promotes testable code**

Spring framework enables writing testable code. It offers good integration with JUnit and Mockito frameworks, which lets us write unit tests quickly and easily. The core feature of Spring is dependency injection and if it is used properly, writing unit tests for the code becomes very easy.

**Staying up to date**

Spring is able to stay current and adapt to changes in development. For example, microservices and cloud services have evolved in the last decade. Spring has come up with projects to keep up with the trend, like Spring Boot, which helps with designing microservices.

**Spring Basics**

**Creating a Project**

Learn a fast way to create and run a Spring project using Spring Boot.

Building a Spring application from scratch can be hard. The developer needs to decide which Maven dependencies to use, set up the configuration for XML or Java, install Tomcat or another server, etc. All these things are needed for building the infrastructure of the application. This involves a lot of configuration in XML or Java, which is difficult and susceptible to errors.

Spring Boot offers a quick-fire way to create a Spring project. It makes getting started with the actual application development easy by skipping the manual configuration part. The highlight of Spring Boot is its auto-configuration feature whereby it automatically includes all the dependencies of a project based on property files and JAR classpaths. Spring Boot is basically the Spring framework along with embedded servers. Spring boot removes the need for XML configuration.

Spring Boot removes the need of manual configuration and comes with an embedded server

📝**Note:** Given the simplicity and ease that Spring Boot provides, we will use it to create our first Spring project.

**Spring Initializr** is one way to create a Spring Boot project, where we can simply select the dependencies and create a basic project structure with a Maven or Gradle build specification. This project is available for download in the form of a zip file to be used in a variety of IDEs like Eclipse or IntelliJ etc.

1. Go to start.spring.io and provide some basic information about the project. We will choose **Maven Project**, select **Java** as the language, and go for the latest version of Spring Boot. At the time of writing this lesson, it is 2.4.4. Any version which is greater than Spring Boot 2.0 will work for the examples that we will go through. It is better to avoid SNAPSHOT versions as they are alpha or beta versions.

Next, the Initializr form asks for project metadata. The group ID and artifact ID are basically the package name and class name. We will use the following:

**GroupId**: io.datajek.spring.basics

**ArtifactId**: movie-recommender-system

Spring Initializr, by default, creates Spring as one of the dependencies of the project, so we do not need to explicitly specify any dependency. Later in this course, we will create projects with dependencies like Web, AOP, JDBC, JPA, etc.

Creating a Spring Boot project

* When the **Generate** button on the form is clicked, the Initializer creates a zip file that is downloaded by the browser. Unzip this file and place it in a folder on the hard drive.
* To import this project in Eclipse, choose **File => Import => Existing Maven Projects**. Search for Maven in the search bar if the option isn’t visible. Browse to the folder on the hard drive where the unzipped Spring Boot project is placed. The **pom.xml** file can be seen. Select the file and click **Finish** to import the project.

Importing Spring Boot project in Eclipse

The first time around, the import will take a little while as all the dependencies needed to set up the project are being downloaded. When the import process finishes, the following hierarchy can be seen:

* src/main/java where the Java code will be written. Right now, it contains the project file **MovieRecommenderSystemApplication.java**
* src/main/resources where the application properties are written.
* src/test/java where the tests will be written.

Project directory structure

The **pom.xml** file contains the project metadata information and lists the dependencies.

The **Maven Dependencies** folder contains the jar files of all the dependencies. Spring has automatically been added as a dependency.spring-boot-starter, spring-context, spring-beans, and spring-core can also be seen among other dependencies autowired by the Spring Initializer.

Maven Dependencies folder

* Based on the data we provided to the Spring Initializr, the **Maven Dependencies folder** directory in our project contains the **io.datajek.spring.basics.movierecommendersystem** package. This package has the MovieRecommenderSystemApplication class containing the main method. We run this class to execute the application.

To run the project, right click the **MovieRecommenderSystemApplication.java** file and run it as **Java Application**.

Running Spring Boot project in Eclipse

This will launch a simple Spring context. The program successfully runs and prints some text on the console.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

package io.datajek.spring.basics.movierecommendersystem.demo;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class MovieRecommenderSystemApplication {

    public static void main(String[] args) {

        SpringApplication.run(MovieRecommenderSystemApplication.class, args);

    }

}





Run

Save

Reset.

**Dependency**

This lesson explains the concept of dependency by setting up a simple example using a Spring Boot application.

**We'll cover the following**

* [Tight coupling](https://www.educative.io/courses/guide-spring-5-spring-boot-2/R8pJ4nYWLOK#Tight-coupling)

In this lesson, we will build a movie recommender system and add some logic to it.

A recommender system is a system that filters some entities based on the user’s history. Recommender systems also rank these items based on user preferences. The system works by taking an input and then finding items similar to that input.

In this lesson, we will write the basic code for a movie recommender application. The application will take a movie and recommend other movies similar to it. There are various ways in which recommendations can be found. One method is *content-based filtering* in which item-to-item similarity is used as a basis for finding matches. So, for a movie like Finding Dory, the system will find movies of the same genre, like Happy Feet, Ice Age, Shark Tale, etc.

Content-based filtering

1. We will create sub-packages inside the **io.datajek.spring.basics.movierecommendersystem** package to keep the files for every lesson separate. This is an optional step. Right click on the base package and click **New** => **Package**. Then provide the name as **lesson1** and click **Finish** to create the sub-package.

The package for each lesson contains the MovieRecommenderSystemApplication class containing the main method. This file can simply be copied from the **io.datajek.spring.basics.movierecommendersystem** package to the newly created package.

1. We will create a class named RecommenderImplementation in the **lesson1** sub-package.

Creating a class

The RecommenderImplementation class will have a method called recommendMovies, which will find similar movies using a content-based filter and then return the results.

The recommendMovies method takes a movie as input so the input parameter is of type String and returns a list of similar movies. Hence, the return type is String[]. Inside the method, we will use a filter to find similar movies and return the results.

public class RecommenderImplementation {  
  
    public String[] recommendMovies (String movie) {  
        //use content based filter to find similar movies  
        //return the results  
        return new String[] {"M1", "M2", "M3"};  
    }  
}

We need to hard code the results returned by this method to avoid compile-time errors. The hard coded results will be removed in the step 5.

Now, we will write the logic of the recommendMovies method in the RecommenderImplementation class. As mentioned earlier, we will use a technique called *content-based filtering* to find movie recommendations. We can implement this technique as a separate class to keep the recommendMovies method independent of the filter implementation.

We will create a new class ContentBasedFilter that has a method getRecommendations implementing the logic of the content-based filter. The method returns a list of movies that are relevant to the input, taking into account the user’s watch history as well as movies that are similar to the input. We will not make it complex at this step and ignore input arguments like the number of movies to recommend, user’s watch history, and the user-movie matrix of ratings.

Our method will have just one input parameter, the movie name, of String type. All the steps in finding the relevant movies are performed in this method.

We have hardcoded the results. The getRecommendations method returns three movies similar to the movie, “*Finding Dory*”.

public class ContentBasedFilter {  
  
    public String[] getRecommendations(String movie) {  
        //logic of content based filter  
        return new String[] {"Happy Feet", "Ice Age", "Shark Tale"};  
    }  
}

Now, we can use an object of the ContentBasedFilter class in the RecommenderImplementation class as follows:

public class RecommenderImplementation {  
  
    public String[] recommendMovies(String movie) {  
        //use content based filter to find similar movies  
        ContentBasedFilter filter = new ContentBasedFilter();  
        String[] results = filter.getRecommendations(movie);  
        //return the results  
        return results;  
    }  
}

Since the getRecommendations method is returning a String[], we can remove our hard-coded results and replace them with the actual ones returned by the method.

In the MovieRecommenderSystemApplication class, we will create an object of RecommenderImplementation class and use it to find movie recommendations for the movie, **Finding Dory**, as follows:

public class MovieRecommenderSystemApplication {  
  
    public static void main(String[] args) {  
        RecommenderImplementation recommender = new RecommenderImplementation();      
        String[] result = recommender.recommendMovies("Finding Dory");  
        System.out.println(Arrays.toString(result));  
    }  
}

To make the output readable, we have used the use the toString method and after importing java.util.Arrays.

The code in the widget below, when executed, returns a list of movies hard-coded in the ContentBasedFilter class.

MovieRecommenderSystemApplication.java

RecommenderImplementation.java

ContentBasedFilter.java

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

package io.datajek.spring.basics.movierecommendersystem.lesson1;

import java.util.Arrays;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class MovieRecommenderSystemApplication {

    public static void main(String[] args) {

        //SpringApplication.run(MovieRecommenderSystemApplication.class, args);

        //create object of RecommenderImplementation class

        RecommenderImplementation recommender = new RecommenderImplementation();

        //call method to get recommendations

        String[] result = recommender.recommendMovies("Finding Dory");

        //display results

        System.out.println(Arrays.toString(result));

    }

}





Run

Save

Reset

**Tight coupling**

As can be seen, the ContentBasedFilter class is a dependency of the RecommenderImplementation class. The RecommenderImplementation class needs an object of ContentBasedFilter class to perform its task. This is an example of *tight coupling*.

Tight coupling of classes

If we want to use another filter in place of the content-based filter, we will need to change the code in the RecommenderImplementation class.

Consider a scenario, where we want to use one type of filter in one situation and another type of filter in another situation. Tight coupling makes this difficult to achieve.

In the simple example above, we created two classes which work together, thus creating a dependency. In a typical enterprise application, there are a large number of objects which work together to provide some end result to the user. This results in a lot of dependencies. Spring is a dependency injection framework that makes the process of managing these dependencies easy

**Decoupling Components**

Let's change the tightly coupled code to loosely coupled.

**We'll cover the following**

* [Filter interface](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7XQn9NkyyWr#Filter-interface)
* [Loose coupling](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7XQn9NkyyWr#Loose-coupling)

Right now, the RecommenderImplementation class is hard coded to use the ContentBasedFilter class. If we need to change the way our application recommends movies, we will need to change the code of the RecommenderImplementation class.

Say, we want to switch from *ContentBased* filter to *Collaborative* filter and take into account the preferences of users having a similar watch history.

Collaborative filtering

1. We have created a sub-package called **lesson2** inside the **io.datajek.spring.basics.movierecommendersystem** package for the code example shown in this lesson.

The package contains the **MovieRecommenderSystemApplication.java**, **RecommenderImplementation.java**, and **ContentBasedFilter.java** files from the previous lesson.

1. Changing the type of filter from content based to collaborative would call for a change in the code of the RecommenderImplementation class.

First, let’s create a class CollaborativeFilter, which, like the ContentBasedFilter class, has one method getRecommendations as follows:

public class CollaborativeFilter {  
   
    public String[] getRecommendations(String movie) {  
        //logic of collaborative filter  
        return new String[] { };  
     }  
 }

This method recommends a list of movies for “Finding Dory” based on the logic of collaborative filter.

1. If we want to switch to the new filter created in the previous step, we will have to change the code in the RecommenderImplementation class as follows:

public String[] recommendMovies(String movie) {  
    CollaborativeFilter filter = new CollaborativeFilter();  
    String[] results =   
filter.getRecommendations("Finding Dory");  
    return results;  
}

Here, we have created an object of the CollaborativeFilter class instead of the ContentBasedFilter class. Every time we want to change the filter implementation, we will have to change the code in the recommendMovies method.

**Filter interface**

One way to make the code loosely coupled is by using an interface called Filter. An interface contains abstract methods whose implementation is left to the classes using it.

We can create an interface by using the wizard just like we created a class in the last lesson. Alternatively, we can simply extend the class definition and Eclipse IDE will automatically prompt to create an interface:

The Filter interface will have only one method definition.

public interface Filter {  
    public String[]   
    getRecommendations(String movie);  
}

Both ContentBasedFilter and CollaborativeFilter now implement the Filter interface.

public class ContentBasedFilter implements Filter{  
    //...  
}

public class CollaborativeFilter implements Filter{  
    //...  
}

**Loose coupling**

Loose coupling can be achieved by making the RecommenderImplementation class use the interface instead of one of its implementations. We will create a constructor for the RecommenderImplementation class to initialize the Filter.

public class RecommenderImplementation {  
  
    //use filter interface to select filter  
    private Filter filter;  
              
    public RecommenderImplementation(Filter filter) {  
        super();  
        this.filter = filter;  
    }  
  
    //use a filter to find recommendations  
    public String [] recommendMovies (String movie) {  
        //...  
    }  
}

The method getRecommendations now belongs to the interface. To check which implementation of the interface is being used to get movie recommendations, we can print the name of the filter as follows:

public String [] recommendMovies (String movie) {  
          
    //print the name of interface implementation being used  
    System.out.println("Name of the filter in use: " + filter + "\n");  
  
    String[] results = filter.getRecommendations("Finding Dory");  
      
    return results;  
}

By using the interface instead of an actual implementation, we can dynamically choose which algorithm to use. Our code has now become *loosely coupled*. In the MovieRecommenderSystemApplication file, when we create a RecommenderImplementation object, we can pass the name of the filter to use:

RecommenderImplementation recommender = new RecommenderImplementation(new ContentBasedFilter());

When we run the application, it prints the name of the filter along with the results.

MovieRecommenderSystemApplication.java

CollaborativeFilter.java

ContentBasedFilter.java

RecommenderImplementation.java

Filter.java

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

package io.datajek.spring.basics.movierecommendersystem.lesson2;

import java.util.Arrays;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class MovieRecommenderSystemApplication {

    public static void main(String[] args) {

        //SpringApplication.run(MovieRecommenderSystemApplication.class, args);

        //passing name of the filter as constructor argument

        RecommenderImplementation recommender = new RecommenderImplementation(new ContentBasedFilter());

        //call method to get recommendations

        String[] result = recommender.recommendMovies("Finding Dory");

        //display results

        System.out.println(Arrays.toString(result));

    }

}





Run

Save

Reset

Let’s now change the code to use the CollaborativeFilter and see the output change.

RecommenderImplementation recommender = new RecommenderImplementation(new CollaborativeFilter());

In this lesson, we made RecommenderImplementation class independent of the filter implementation. The RecommenderImplementation now calls methods of the Filter interface.

`Filter` interface removes tight coupling

Now **Filter** is a dependency of RecommenderImplementation. We still have to create an object of RecommenderImplementation and an object of Filter and pass the objects to the constructor.

**Managing Beans and Dependencies**

Learn how to use annotations to direct Spring to manage beans and autowire dependencies.

**We'll cover the following**

* [@Component](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMN7P6BlEGO#@Component)
* [@Autowired](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMN7P6BlEGO#@Autowired)
* [@ComponentScan](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMN7P6BlEGO#@ComponentScan)
* [@SpringBootApplication](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMN7P6BlEGO#@SpringBootApplication)

So far, we have created objects of RecommenderImplementation class and two classes implementing the Filter interface. We are binding the objects together in the constructor. Our code is now *loosely coupled* as we are passing the name of the filter to be used as a constructor argument.

Spring automates the above process of creating objects and binding them together. It takes the responsibility of creating instances of classes and binding instances based on their dependencies. The instances or objects that Spring manages are called **beans**. To manage objects and dependencies, Spring requires information about three things:

* Beans
* Dependencies
* Location of beans

1. For the code example shown in this lesson, we have created a sub-package called **lesson3** inside the package **io.datajek.spring.basics.movierecommendersystem**.

The package contains **MovieRecommenderSystemApplication.java**, **RecommenderImplementation.java**, **ContentBasedFilter.java**, and **CollaborativeFilter.java** files from the previous lesson.

**@Component**

* If we want Spring to create and manage objects, we can do so by adding the @Component annotation at the beginning of the class and importing **org.springframework.stereotype.Component**.

For now, we want Spring to manage objects of RecommenderImplementation and ContentBasedFilter class only, so we will add the @Component annotation at two places in the code:

import org.springframework.stereotype.Component;  
  
@Component  
public class RecommenderImplementation {  
    //...  
}

import org.springframework.stereotype.Component;  
  
@Component  
public class ContentBasedFilter implements Filter {  
    //...  
}

The Spring container will have two beans, one of type RecommenderImplementation and the other of type ContentBasedFilter.

Beans in the Spring container

**@Autowired**

* The second thing Spring needs to know is the dependencies of each object. The @Autowired annotation is used for this purpose and we need to import **org.springframework.beans.factory.annotation.Autowired** to be able to use this annotation.

In our application, the ContentBasedFilter class (which implements the Filter interface) is a dependency of the RecommenderImplementation class.

import org.springframework.stereotype.Component;  
import org.springframework.beans.factory.annotation.Autowired;  
  
@Component  
public class RecommenderImplementation {  
  
    @Autowired  
    private Filter filter;  
    // ...  
}

The @Autowired annotation tells Spring that RecommenderImplementation needs an object of type Filter. In other words, Filter is a dependency of RecommenderImplementation.

Identifying bean dependencies

**@ComponentScan**

* The third thing that Spring requires from the developer, is the location of the beans so that it can find them and autowire the dependencies. The @ComponentScan annotation is used for this purpose. This annotation can be used with or without arguments. It tells Spring to scan a specific package and all of its sub-packages. In our case, all the files that contain beans are in the same package, io.datajek.springbasics, so we want Spring to do a component scan on this package.

Since we are using Spring Boot, it uses the @SpringBootApplication annotation on the **MovieRecommenderSystemApplication** class. This annotation is equivalent to the following three annotations:

* @Configuration, which declares a class as the source for bean definitions

@EnableAutoConfiguration, which allows the application to add beans using classpath definitions

* @ComponentScan, which directs Spring to search for components in the path specified

`@SpringBootApplication` combines three annotations

Because of @SpringBootApplication annotation, we do not need to use @ComponentScan annotation in our code.

**@SpringBootApplication**

@SpringBootApplication tells Spring to scan all the files in the package where the class with this annotation is present. It also scans any sub-packages of the package where it is placed.

When we use the @Component, @Autowired, and @SpringBootApplication annotations, the following line in our code becomes redundant as it is automatically done by Spring:

RecommenderImplementation recommender = new RecommenderImplementation(new ContentBasedFilter());

* The beans that Spring creates are managed by the *Application Context*. We can get information about a bean from the Application Context. The run method returns the ApplicationContext, which can be assigned to a variable appContext. Then the getBean method of ApplicationContext can be used to get the bean of a particular class. We will create a local variable recommender and assign the bean to it as follows:

public static void main(String[] args) {  
      
    //ApplicationContext manages the beans and dependencies  
    ApplicationContext appContext = SpringApplication.run(          
                                             MovieRecommenderSystemApplication.class, args);  
  
    //use ApplicationContext to find which filter is being used  
    RecommenderImplementation recommender = appContext.getBean(  
                                             RecommenderImplementation.class);    
      
    //call method to get recommendations  
    String[] result = recommender.recommendMovies("Finding Dory");  
      
    //display results   
    System.out.println(Arrays.toString(result));  
}

Instead of us having to create an instance of the RecommenderImplementation class, Spring Application Context creates the beans. We can simply pick it up from there and use it to execute the RecommendMovies method.

This might look complex to a beginner, but consider for a moment an application that has hundreds of beans, each having a number of dependencies. The fact that we do not have to explicitly create beans and manually wire in the dependencies makes the job of a developer very easy.

When we run this application, the output shows that the bean being used is **ContentBasedFilter**. If the @Component annotation is used on the **CollaborativeFilter** class instead of the **ContentBasedFilter** class, the output will change accordingly:

MovieRecommenderSystemApplication.java

CollaborativeFilter.java

ContentBasedFilter.java

Filter.java

RecommenderImplementation.java

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

package io.datajek.spring.basics.movierecommendersystem.lesson3;

import java.util.Arrays;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ApplicationContext;

@SpringBootApplication

public class MovieRecommenderSystemApplication {

    public static void main(String[] args) {

        //ApplicationContext manages the beans and dependencies

        ApplicationContext appContext = SpringApplication.run(MovieRecommenderSystemApplication.class, args);

        //use ApplicationContext to find which filter is being used

        RecommenderImplementation recommender = appContext.getBean(RecommenderImplementation.class);

        //call method to get recommendations

        String[] result = recommender.recommendMovies("Finding Dory");

        //display results

        System.out.println(Arrays.toString(result));

    }

}





Run

Save

Reset

1. To understand what goes on in the background, we will change the logging level to debug. This can be done by adding the following to the application.properties file in src/main/resources:

Logging.level.org.springframework = debug

When run, the next code widget will show the log of all the actions that are being performed in the background. A summary of the actions is reproduced below:

* *Loading source class…*

The package is being searched. Spring starts with a component scan to find anything with @Component as well as other annotations.

* *Identified candidate component class…*

Spring identifies two candidates which have the @Component annotation as we only used it in two places in our code.

* *Creating shared instance of singleton bean ‘movieRecommenderSystemApplication’*
* *Creating shared instance of singleton bean ‘contentBasedFilter’*

Spring starts creating instances of the beans. It creates beans that do not have any dependency first.

* *Creating shared instance of singleton bean ‘recommenderImplementation’*

*Autowiring by type from bean name ‘recommenderImplementation’ via constructor to bean named ‘contentBasedFilter’*

Now Spring can autowire the dependency using the constructor that we have provided and creates the RecommenderImplementation bean.

* To better understand these annotations, play around with the code below and see what error messages Spring throws when some of the annotations are missing. The error message can be found at the end of the log.

If we remove @Component from the ContentBasedFilter class, Spring will throw an error when trying to autowire the dependency saying it required a bean of type **Filter** that could not be found.

If we remove @Component from the RecommenderImplementation class as well, we will get an error when trying to execute the getBean method as no beans exist.

If we add @Component to the CollaborativeFilter class, Spring will not know which bean of Filter type to autowire. It says, “expected single matching bean but found two”.

MovieRecommenderSystemApplication.java

application.properties

CollaborativeFilter.java

ContentBasedFilter.java

Filter.java

RecommenderImplementation.java

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

package io.datajek.spring.basics.movierecommendersystem.lesson3;

import java.util.Arrays;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ApplicationContext;

@SpringBootApplication

public class MovieRecommenderSystemApplication {

    public static void main(String[] args) {

        //ApplicationContext manages the beans and dependencies

        ApplicationContext appContext = SpringApplication.run(MovieRecommenderSystemApplication.class, args);

        //use ApplicationContext to find which filter is being used

        RecommenderImplementation recommender = appContext.getBean(RecommenderImplementation.class);

        //call method to get recommendations

        String[] result = recommender.recommendMovies("Finding Dory");

        //display results

        System.out.println(Arrays.toString(result));

    }

}





Run

Save

Reset

**Autowiring By Type — @Primary**

This lesson demonstrates how Spring dynamically autowires a dependency in case it finds more than one component of the same type.

**We'll cover the following**

* [NoUniqueBeanDefinitionException](https://www.educative.io/courses/guide-spring-5-spring-boot-2/myoBYoM4qE0#NoUniqueBeanDefinitionException)
* [@Primary annotation](https://www.educative.io/courses/guide-spring-5-spring-boot-2/myoBYoM4qE0#@Primary-annotation)

In the last lesson, we saw Spring manage two beans of the RecommenderImplementation and ContentBasedFilter classes for us. In this lesson, we will add another bean and see how Spring can dynamically choose a bean if it finds two matches of the same type.

1. For the code example shown in this lesson, we have created a sub-package called **lesson4** inside the package **io.datajek.spring.basics.movierecommendersystem**.

The package contains **MovieRecommenderSystemApplication.java**, **RecommenderImplementation.java**, **ContentBasedFilter.java**, and **CollaborativeFilter.java** files from the previous lesson.

**NoUniqueBeanDefinitionException**

1. We will add the @Component annotation on the CollaborativeFilter class to declare it a bean. Now both implementations of the Filter interface are beans. Previously, when Spring searched for a dependency to be autowired in the RecommenderImplementation object, it only found one bean of matching type. Now when we run the application, it fails to start.

The NoUniqueBeanDefinitionException occurs. The error message says: Required a single bean but two were found.

MovieRecommenderSystemApplication.java

Filter.java

CollaborativeFilter.java

RecommenderImplementation.java

ContentBasedFilter.java

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

package io.datajek.spring.basics.movierecommendersystem.lesson4;

import org.springframework.stereotype.Component;

@Component

public class ContentBasedFilter implements Filter{

    //getRecommendations takes a movie as input and returns a list of similar movies

    public String[] getRecommendations(String movie) {

        //implement logic of content based filter

        //return movie recommendations

        return new String[] {"Happy Feet", "Ice Age", "Shark Tale"};

    }

}





Run

Save

Reset

**@Primary annotation**

1. One way Spring can choose between two beans of the same type is by giving one bean priority over the other. The @Primary annotation is used for making a component the default choice when multiple beans of the same type are found.

Let’s say we want the collaborative filter to take precedence. We will add the @Primary annotation on the CollaborativeFilter class and import **org.springframework.context.annotation.Primary**. When we run the application now, it uses CollaborativeFilter as expected.

Bean with `@Primary` annotation is injected

MovieRecommenderSystemApplication.java

Filter.java

CollaborativeFilter.java

RecommenderImplementation.java

ContentBasedFilter.java

1

2

3

4

5

6

7

8

9

10

11

12

13

14

package io.datajek.spring.basics.movierecommendersystem.lesson4;

import org.springframework.stereotype.Component;

import org.springframework.context.annotation.Primary;

@Component

@Primary

public class CollaborativeFilter implements Filter{

    public String[] getRecommendations(String movie) {

        //logic of collaborative filter

        return new String[] {"Finding Nemo", "Ice Age", "Toy Story"};

    }

}





Run

Save

Reset

Using @Primary is called *autowiring by type*. We are looking for instances of type Filter.

1. If we make both beans primary by adding the @Primary annotation to both implementations of the Filter interface, we will get an error. This happens because Spring doesn’t know which one to inject in the RecommenderImplementation object. The error message states “more than one ‘primary’ bean found among candidates".

**Autowiring By Name**

Let's look at another autowiring approach known as autowiring by name and see which approach has higher priority; by name or by type.

In the last lesson, we looked at the autowiring by type approach where priority was given to the collaborative filter using the @Primary annotation.

Another approach is autowiring by name where we specify the bean that is to be used by name. In this approach, while creating an object, the dependency is injected by matching the name of the reference variable to the bean name. The developer has to ensure that the variable name is the same as its bean name.

1. For the code example shown in this lesson, we have created a sub-package called **lesson5** inside the package **io.datajek.spring.basics.movierecommendersystem**.

The package contains **MovieRecommenderSystemApplication.java**, **RecommenderImplementation.java**, **Filter.java**, **ContentBasedFilter.java**, and **CollaborativeFilter.java** files from the previous lesson.

1. We will begin by omitting the @Primary annotation from the CollaborativeFilter class. Now, to let Spring know which bean to use, we will change the variable name in the RecommenderImplementation class to match the bean name as follows:

public class RecommenderImplementation {  
  
    @Autowired  
    private Filter contentBasedFilter;  
   
    public String [] recommendMovies (String movie) {         
        System.out.println("\nName of the filter in use: " + contentBasedFilter + "\n");  
        String[] results = contentBasedFilter.getRecommendations("Finding Dory");  
        return results;  
   }  
}

Now when the application is run, it chooses the ContentBasedFilter bean for autowiring. When Spring finds two beans of the same type (Filter), it determines that the bean to inject is the one whose name matches the bean with the @Component annotation. In other words, the variable name (contentBasedFilter) matches the bean name (ContentBasedFilter).

public class RecommenderImplementation {  
    @Autowired  
    private Filter contentBasedFilter;  
    //...  
}

@Component  
public class ContentBasedFilter implements Filter{  
    //...  
}

Bean with matching name is injected

The code is shown below:

MovieRecommenderSystemApplication.java

Filter.java

CollaborativeFilter.java

RecommenderImplementation.java

ContentBasedFilter.java

1

2

3

4

5

6

7

8

9

10

11

12

13

package io.datajek.spring.basics.movierecommendersystem.lesson5;

import org.springframework.stereotype.Component;

@Component

public class CollaborativeFilter implements Filter{

    public String[] getRecommendations(String movie) {

        //logic of collaborative filter

        return new String[] {"Finding Nemo", "Ice Age", "Toy Story"};

    }

}





Run

Save

Reset

1. As an exercise, let’s see what happens if the bean name and variable names are different. Let’s change the name of the variable to filter.

When the application is run, autowiring does not take place and, as expected, we get the *NoUniqueBeanDefinitionException*.

1. We have seen two autowiring approaches so far. To see which autowiring approach takes precedence, we will use the @Primary annotation on ContentBasedFilter class and use autowiring by name by changing the name of the variable of type Filter in RecommenderImplementation class to collaborativeFilter.

The application chooses the ContentBasedFilter bean, showing that @Primary has a higher priority.

This is because @Autowired annotation tries to resolve dependency by type first. If it fails to resolve a conflict and finds more than one bean of the same type then it tries to resolve it by name.

The autowiring by name approach is advantageous when we want to use one bean in one situation and another bean in some other situation. Using @Primary will always give preference to one bean, which is impractical if we want to use different beans in different scenarios. Autowiring by name ensures that if we have some other component which wants to use another type of bean, it can request Spring by using a different variable name.

**Autowiring — @Qualifier**

Let's examine the @Qualifier annotation for autowiring and compare it to @Primary.

**We'll cover the following**

* [@Qualifier annotation](https://www.educative.io/courses/guide-spring-5-spring-boot-2/gxwV2Km7yj3#@Qualifier-annotation)
* [Comparison with @Primary](https://www.educative.io/courses/guide-spring-5-spring-boot-2/gxwV2Km7yj3#Comparison-with-@Primary)

**@Qualifier annotation**

Like @Primary, the @Qualifier annotation gives priority to one bean over the other if two beans of the same type are found. The bean whose name is specified in the @Qualifier annotation qualifies to be injected as a dependency. The @Qualifier annotation can be used in a scenario when we have multiple objects of the same type and autowiring by name cannot be used because the variable name doesn’t match any bean name.

1. For the code example shown in this lesson, we have created a sub-package called **lesson6** inside the package **io.datajek.spring.basics.movierecommendersystem**.

The package contains **MovieRecommenderSystemApplication.java**, **RecommenderImplementation.java**, **Filter.java**, **ContentBasedFilter.java**, and **CollaborativeFilter.java** files from the previous lesson.

1. Say, we want to use the name **CBF** for ContentBasedFilter. We can either specify it in the @Component annotation or use @Qualifier annotation on the class. Both approaches, shown below, yield the same result:

@Component("CBF")  
public  class ContentBasedFilter implements Filter{  
    //...  
}

@Component  
@Qualifier("CBF")  
public  class ContentBasedFilter implements Filter{  
    //...  
}

Now, we can use the @Qualifier annotation in the RecommenderImplementation class where the dependency is injected to indicate which bean to use.

public class RecommenderImplementation {  
  
    @Autowired  
    @Qualifier("CBF")  
    private Filter filter;  
   
    public String [] recommendMovies (String movie) {         
        //...  
    }  
}

The name of the Filter implememtation used with the @Qualifier annotation (in this case, **CBF**) has to match the name used with the @Component (or @Qualifier) annotation on the class.

Bean with same `@Qualifier` value is injected

When the application is run, the ContentBasedFilter bean qualifies to be autowired.

1. We can use the name **CF** for CollaborativeFilter class as follows:

@Component  
@Qualifier("CF")  
public  class CollaborativeFilter implements Filter{  
    //...  
}

Another way to give the name **CF** to the CollaborativeFilter bean is:

@Component("CF")  
public  class CollaborativeFilter implements Filter{  
    //...  
}

In the RecommenderImplementation class, we can inject CollaborativeFilter bean by using @Qualifier annotation and specifying the name **CF**.

@Autowired  
@Qualifier("CF")  
private  Filter filter;

Depending upon which filter is required in a given scenario, we can change the @Qualifier annotation in the RecommenderImplementation class.

MovieRecommenderSystemApplication.java

Filter.java

CollaborativeFilter.java

RecommenderImplementation.java

ContentBasedFilter.java

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

package io.datajek.spring.basics.movierecommendersystem.lesson6;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.beans.factory.annotation.Qualifier;

import org.springframework.stereotype.Component;

@Component

public class RecommenderImplementation {

    @Autowired

    @Qualifier("CF")

    private Filter filter;

    //use a filter to find recommendations

    public String [] recommendMovies (String movie) {

        //print the name of interface implementation being used

        System.out.println("\nName of the filter in use: " + filter + "\n");

        String[] results = filter.getRecommendations("Finding Dory");

        return results;

    }

}





Run

Save

Reset

**Comparison with @Primary**

1. The @Qualifier annotation takes precedence over the @Primary annotation. To show this, let’s add the @Primary annotation to the ContentBasedFilter class and run the application.

MovieRecommenderSystemApplication.java

Filter.java

CollaborativeFilter.java

RecommenderImplementation.java

ContentBasedFilter.java

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

package io.datajek.spring.basics.movierecommendersystem.lesson6;

import org.springframework.context.annotation.Primary;

import org.springframework.beans.factory.annotation.Qualifier;

import org.springframework.stereotype.Component;

@Component

@Primary

@Qualifier("CBF")

public class ContentBasedFilter implements Filter{

    //getRecommendations takes a movie as input and returns a list of similar movies

    public String[] getRecommendations(String movie) {

        //implement logic of content based filter

        //return movie recommendations

        return new String[] {"Happy Feet", "Ice Age", "Shark Tale"};

    }

}





Run

Save

Reset

When the application is run, the CollaborativeFilter bean gets autowired. This is because @Primary is the default setting, while @Qualifier is specific. @Primary defines the default selection when no other information is available. It tells Spring to use the bean marked as primary as its first choice if it encounters more than one bean of the same type. On the other hand, @Qualifier tells Spring to use a specific bean if it finds multiple beans of matching type.

@Primary annotation should be used if there is one clear favorite to be used in a majority of situations. In some cases, one algorithm might be more efficient or more important than the rest and is declared as the primary choice. The bean with @Primary gets chosen unless another bean is required, which can be specified with @Qualifier. The bean with @Qualifier is only used to request an “alternate” bean in case the primary choice is not required.

@Autowired annotation resolves dependencies by type. If there are more than one beans of the same type, a conflict arises. We have seen three different approaches to resolve conflicts. They can be resolved using the @Primary annotation, renaming the variable to match the name of the class, or by using the @Qualifier annotation.

**Constructor and Setter Injection**

Learn how beans can be wired in using constructor and setter injection.

**We'll cover the following**

* [Constructor injection](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JE0y2ZEGq1v#Constructor-injection)
* [Setter injection](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JE0y2ZEGq1v#Setter-injection)
* [Field injection](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JE0y2ZEGq1v#Field-injection)

Spring framework identifies dependencies and wires them in. Spring framework gives the developer control over how beans are wired in. There are a variety of options to choose from. We will focus on *constructor injection* and \_setter injection.

Dependency injection

1. For the code example shown in this lesson, we have created a sub-package called **lesson7** inside the package **io.datajek.spring.basics.movierecommendersystem**.

The package contains **MovieRecommenderSystemApplication.java**, **RecommenderImplementation.java**, **Filter.java**, **ContentBasedFilter.java**, and **CollaborativeFilter.java** files from the previous lesson.

1. To show the two different ways of dependency injection, we will create a copy of the RecommenderImplementation class and call it RecommenderImplementation2. One will be used to show constructor injection while the other will demonstrate setter injection.

We will use the @Autowired annotation at different places in the code to direct Spring which injection type to use.

**Constructor injection**

1. Autowiring the dependency using a constructor is called *constructor injection*. We will create a constructor in the RecommenderImplementation class that initializes the filter to be used for finding movie recommendations as follows:

public RecommenderImplementation( Filter filter) {  
    this.filter = filter;  
    System.out.println("Constructor invoked...");  
}

Since we have two implementations of the Filter interface, we need to specify which one to use. If we use the @Primary annotation, then Spring will use the primary bean as the default choice. However, if we want different beans to be used in different scenarios, then the @Qualfier annotation with the bean name can be used to give a hint to Spring about which bean to inject. Since, we have not specified bean names with the @Component annotation, their default names (contentBasedFilter and collaborativeFilter) will be used.

*Default bean name is the class name with the first letter in lowercase.*

1. To use constructor for injecting dependencies, we can move the @Autowired annotation to the constructor. We will also use the @Qualifier annotation to inject the bean of CollaborativeFilter type. The @Qualifier annotation cannot be used on the constructor (as it results in an error message: *“The annotation @Qualifier is disallowed for this location”*), rather, it should be used in the argument list right in front of the property that we want to be autowired as shown:

@Autowired  
public RecommenderImplementation(@Qualifier("collaborativeFilter") Filter filter) {  
    this.filter = filter;  
    System.out.println("Constructor invoked...");  
}

It should also be noted that the use of the @Autowired annotation is optional when using constructors.

1. In the main method, we will call the getBean method to show that constructor injection takes place and CollaborativeFilter bean is injected as follows:

public static void main(String[] args) {  
      
    ApplicationContext appContext = SpringApplication.run(  
                                    MovieRecommenderSystemApplication.class, args);  
  
    //RecommenderImplementation injects dependency using constructor  
    System.out.println("Calling getBean() on RecommenderImplementation");  
    RecommenderImplementation recommender = appContext.getBean(  
                                                  RecommenderImplementation.class);   
  
    String[] result = recommender.recommendMovies("Finding Dory");  
    System.out.println(Arrays.toString(result));  
}

When the application is run, Spring injects the CollaborativeFilter bean in the RecommenderImplementation class using the constructor. The “Constructor invoked…” message is printed on the console.

**Setter injection**

1. Another way to wire in a dependency is by using a setter method. We will create a setter method in the RecommenderImplementation2 class called setFilter as follows:

public void setFilter(Filter filter) {  
    this.filter = filter;  
    System.out.println("Setter method invoked..");  
}

1. We can guide Spring to use the setter method by using the @Autowired annotation before the method. We will also use the @Qualifier annotation to instruct Spring to use the ContentBasedFilter bean as follows:

@Autowired  
@Qualifier("contentBasedFilter")  
public void setFilter(Filter filter) {  
    //...  
}

1. In the main method, we will call the getBean method to show that setter injection takes place and ContentBasedFilter bean is injected as follows:

public static void main(String[] args) {  
      
    ApplicationContext appContext = SpringApplication.run(  
                                    MovieRecommenderSystemApplication.class, args);  
  
    //...  
    
    //RecommenderImplementation2 injects dependency using setter method  
    System.out.println("Calling getBean() on RecommenderImplementation2");  
    RecommenderImplementation2 recommender2 = appContext.getBean(  
                                                 RecommenderImplementation2.class);   
  
    result = recommender2.recommendMovies("Finding Dory");  
    System.out.println(Arrays.toString(result));  
}

When the application is run, Spring injects the ContentBasedFilter bean in the RecommenderImplementation2 class using the setFilter method. The “Setter invoked…” message is also displayed on the console.

**Field injection**

1. We have seen two dependency injection methods above but Spring was already performing dependency injection without a constructor or setter method in the RecommenderImplementation class. We have been using @Autowired annotation directly on the Filter field. This is called *field injection*.

public class RecommenderImplementation {  
    @Autowired  
    private Filter filter;  
      
    //...      
}

Using field injection keeps the code simple and readable, but it is unsafe because Spring can set private fields of the objects. Testing also becomes inconvenient because we need a way to perform dependency injection for testing. Yet another disadvantage is that a developer may add a lot of optional dependencies which can make the application complex. If there was a constructor, then each additional dependency would result in increasing the number of arguments of the constructor.

Both constructor and setter injection result in the same outcome. However, there are some differences. Setter injection is more readable as it specifies the name of the dependency as the method name but the number of setter methods increases with each increasing dependency increasing the boiler plate code. Setter injection is used to avoid the *BeanCurrentlyInCreationException* raised in case of a circular dependency, because unlike constructor injection where dependencies are injected at the time when context is loaded, setter injection injects dependencies when they are needed.

Constructor injection ensures that all dependencies are injected because an object cannot be constructed until all its dependencies are available. It also ensures immutability as the state of the bean cannot be modified after creation.

MovieRecommenderSystemApplication.java

Filter.java

ContentBasedFilter.java

CollaborativeFilter.java

RecommenderImplementation2.java

RecommenderImplementation.java

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

package io.datajek.spring.basics.movierecommendersystem.lesson7;

import java.util.Arrays;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ApplicationContext;

@SpringBootApplication

public class MovieRecommenderSystemApplication {

    public static void main(String[] args) {

        //ApplicationContext manages the beans and dependencies

        ApplicationContext appContext = SpringApplication.run(MovieRecommenderSystemApplication.class, args);

        //RecommenderImplementation injects dependency using constructor

        System.out.println("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

        System.out.println("Calling getBean() on RecommenderImplementation");

        RecommenderImplementation recommender = appContext.getBean(RecommenderImplementation.class);

        String[] result = recommender.recommendMovies("Finding Dory");

        System.out.println(Arrays.toString(result));

        //RecommenderImplementation2 injects dependency using setter method

        System.out.println("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

        System.out.println("Calling getBean() on RecommenderImplementation2");

        RecommenderImplementation2 recommender2 = appContext.getBean(RecommenderImplementation2.class);

        result = recommender2.recommendMovies("Finding Dory");

        System.out.println(Arrays.toString(result));

    }





Run

Save

Reset

# Spring In-depth

**Bean Scope**

Learn about the differences between singleton and prototype bean scopes.

**We'll cover the following**

* [Types of bean scopes](https://www.educative.io/courses/guide-spring-5-spring-boot-2/xl1xxQlxvqz#Types-of-bean-scopes)
* [Singleton scope](https://www.educative.io/courses/guide-spring-5-spring-boot-2/xl1xxQlxvqz#Singleton-scope)
* [Prototype scope](https://www.educative.io/courses/guide-spring-5-spring-boot-2/xl1xxQlxvqz#Prototype-scope)
* [Spring vs. Gang of Four singleton](https://www.educative.io/courses/guide-spring-5-spring-boot-2/xl1xxQlxvqz#Spring-vs.-Gang-of-Four-singleton)

The Spring container manages beans. The term bean scope refers to the lifecycle and the visibility of beans. It tells how long the bean lives, how many instances of the bean are created, and how the bean is shared.

**Types of bean scopes**

There are six types of scopes: singleton, prototype, request, session, application, and websocket.

Types of bean scopes

The singleton and prototype scopes can be used in any application while the last four scopes are only available for a web application. In this lesson, we will focus on singleton and prototype bean scopes only.

**Singleton scope**

The default scope of a bean is **singleton**, in which only one instance of the bean is created and cached in memory. Multiple requests for the bean return a shared reference to the same bean. In contrast, **prototype** scope results in the creation of new beans whenever a request for the bean is made to the application context.

In our movie recommendation system example, we have two implementations of the Filter interface, namely ContentBasedFilter and CollaborativeFilter. We will use them to show the differences between singleton and prototype bean scope.

For the code example shown in this lesson, we have created a sub-package called **lesson8** inside the package **io.datajek.spring.basics.movierecommendersystem**.

The package contains **MovieRecommenderSystemApplication.java**, **Filter.java**, **ContentBasedFilter.java**, and **CollaborativeFilter.java** files.

1. Application context manages the beans and we can retrieve a bean using the getBean() method. If we request the application context for the ContentBasedFilter bean three times as shown, we get the same bean:

public static void main(String[] args) {  
    //ApplicationContext manages the beans and dependencies  
    ApplicationContext appContext = SpringApplication.run(  
                                       MovieRecommenderSystemApplication.class, args);  
  
    //Retrieve singleton bean from application context thrice  
    ContentBasedFilter cbf1 = appContext.getBean(ContentBasedFilter.class);   
    ContentBasedFilter cbf2 = appContext.getBean(ContentBasedFilter.class);   
    ContentBasedFilter cbf3 = appContext.getBean(ContentBasedFilter.class);   
          
    System.out.println(cbf1);  
    System.out.println(cbf2);  
    System.out.println(cbf3);  
}

MovieRecommenderSystemApplication.java

Filter.java

CollaborativeFilter.java

ContentBasedFilter.java

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

package io.datajek.spring.basics.movierecommendersystem.lesson8;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ApplicationContext;

@SpringBootApplication

public class MovieRecommenderSystemApplication {

    public static void main(String[] args) {

        //ApplicationContext manages the beans and dependencies

        ApplicationContext appContext = SpringApplication.run(MovieRecommenderSystemApplication.class, args);

        //Retrieve singleton bean from application context thrice

        ContentBasedFilter cbf1 = appContext.getBean(ContentBasedFilter.class);

        ContentBasedFilter cbf2 = appContext.getBean(ContentBasedFilter.class);

        ContentBasedFilter cbf3= appContext.getBean(ContentBasedFilter.class);

        System.out.println(cbf1);

        System.out.println(cbf2);

        System.out.println(cbf3);

    }

}





Run

Save

Reset

As can be verified from the output, all beans are the same. The application context did not create a new bean when we requested it the second and third time. Rather, it returned the reference to the bean already created. Pictorially, it can be shown as follows:

Singleton scope

Singleton bean scope is the default scope. It is used to minimize the number of objects created. Beans are created when the context is loaded and cached in memory. All requests for a bean are returned with the same memory address. This type of scope is best suited for cases where stateless beans are required. On the contrary, prototype bean scope is used when we need to maintain the state of the beans.

**Prototype scope**

1. Now we will change the scope of the CollaborativeFilter bean from singleton to prototype. For this, we will use the @Scope annotation and import org.springframework.context.annotation.Scope. We can specify the scope in the two ways shown below. Option 2 is the preferred approach.

//Option 1  
@Scope("prototype")  
  
//Option 2   
@Scope(ConfigurableBeanFactory.SCOPE\_PROTOTYPE)

Just like the previous step, we will ask the application context for the CollaborativeFilter bean three times and output the results as follows:

public static void main(String[] args) {  
    //ApplicationContext manages the beans and dependencies  
    ApplicationContext appContext = SpringApplication.run(  
                                    MovieRecommenderSystemApplication.class, args);  
  
    //...  
  
    //Retrieve prototype bean from application context thrice  
    CollaborativeFilter cf1 = appContext.getBean(CollaborativeFilter.class);      
    CollaborativeFilter cf2 = appContext.getBean(CollaborativeFilter.class);      
    CollaborativeFilter cf3 = appContext.getBean(CollaborativeFilter.class);  
      
    System.out.println(cf1);  
    System.out.println(cf2);  
    System.out.println(cf3);  
}

This time the application context will return three different objects. It will create a new object every time we invoke the getBean() method.

Prototype scope

MovieRecommenderSystemApplication.java

Filter.java

CollaborativeFilter.java

ContentBasedFilter.java

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

package io.datajek.spring.basics.movierecommendersystem.lesson8;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ApplicationContext;

@SpringBootApplication

public class MovieRecommenderSystemApplication {

    public static void main(String[] args) {

        //ApplicationContext manages the beans and dependencies

        ApplicationContext appContext = SpringApplication.run(MovieRecommenderSystemApplication.class, args);

        //Retrieve prototype bean from application context thrice

        CollaborativeFilter cf1 = appContext.getBean(CollaborativeFilter.class);

        CollaborativeFilter cf2 = appContext.getBean(CollaborativeFilter.class);

        CollaborativeFilter cf3 = appContext.getBean(CollaborativeFilter.class);

        System.out.println(cf1);

        System.out.println(cf2);

        System.out.println(cf3);

    }

}





Run

Save

Reset

Spring creates a singleton bean even before we ask for it while a prototype bean is not created till we request Spring for the bean. In the code widget below, we will print a message in the ContentBasedFilter and CollaborativeFilter constructors and comment everything in the main method. When the application is run, the output shows that Spring has created a ContentBasedFilter bean but the CollaborativeFilter bean has not yet been created.

MovieRecommenderSystemApplication.java

Filter.java

CollaborativeFilter.java

ContentBasedFilter.java

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

package io.datajek.spring.basics.movierecommendersystem.lesson8;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ApplicationContext;

@SpringBootApplication

public class MovieRecommenderSystemApplication {

    public static void main(String[] args) {

        //ApplicationContext manages the beans and dependencies

        ApplicationContext appContext = SpringApplication.run(MovieRecommenderSystemApplication.class, args);

        //Retrieve singleton bean from application context thrice

    //  ContentBasedFilter cbf1 = appContext.getBean(ContentBasedFilter.class);

    //  ContentBasedFilter cbf2 = appContext.getBean(ContentBasedFilter.class);

    //  ContentBasedFilter cbf3= appContext.getBean(ContentBasedFilter.class);

        //Retrieve prototype bean from application context thrice

    //  CollaborativeFilter cf1 = appContext.getBean(CollaborativeFilter.class);

    //  CollaborativeFilter cf2 = appContext.getBean(CollaborativeFilter.class);

    //  CollaborativeFilter cf3 = appContext.getBean(CollaborativeFilter.class);

    }

}





Run

Save

Reset

If the code creating multiple objects of both classes is un-commented (lines 16-18 and 21–23), it will be seen that the singleton bean constructor is called only once while the prototype bean constructor is called three times.

**Spring vs. Gang of Four singleton**

It is important to note that there is a difference between the Spring singleton and the Gang of Four (GoF) singleton design patterns. The singleton design pattern as specified by the GoF means one bean per JVM. However, in Spring it means one bean per application context. By the GoF definition, even if there were more than one application contexts running on the same JVM, there would still be only one instance of the singleton class.

**Mixing Bean Scope**

See what happens when the dependency of a singleton bean has prototype scope and learn one of the several possible ways to get the correct output.

**We'll cover the following**

* [Singleton bean with prototype dependency](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVvpGvK3BkR#Singleton-bean-with-prototype-dependency)
* [Proxy](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVvpGvK3BkR#Proxy)
* [@Lookup](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVvpGvK3BkR#@Lookup)

**Singleton bean with prototype dependency**

In this lesson, we will discuss an interesting problem of mixing bean scopes. Sometimes, a bean has singleton scope but its dependency has prototype scope. An example is the content-based filter which recommends movies based on item-to-item similarity. Our basic implementation of the content-based filter compares different movies and assigns a similarity score. Hence, Movie is a dependency of the ContentBasedFilter class.

The ContentBasedFilter bean has singleton scope because we need only one instance of the filter. However, the Movie bean has prototype scope because we need more than one objects of this class.

Singleton bean with prototype dependency

1. For the code example shown in this lesson, we have created a sub-package called **lesson9** inside the package **io.datajek.spring.basics.movierecommendersystem**.

The package contains **MovieRecommenderSystemApplication.java**, **Filter.java**, and **ContentBasedFilter.java** files from the previous lesson.

1. We will create a new class called Movie in the **lesson9** package. This class has variables for storing movie name, genre and producer, as well as a static member, instances, to maintain the global count of instances created. The variable instances is incremented in the constructor. The scope of the Movie class is prototype because we want multiple objects of this class in the ContentBasedFilter class.

@Component @Scope(ConfigurableBeanFactory.SCOPE\_PROTOTYPE)  
public class Movie {  
    //for keeping track of instances created  
    private static int instances = 0;  
  
    private int id;  
    private String name;  
    private String genre;    
    private String producer;   
   
    public Movie() {  
        instances++;  
        System.out.println("Movie constructor called");  
     }  
  
    public static int getInstances() {  
        return Movie.instances;  
    }  
  
    //...  
}

1. We will modify the ContentBasedFilter class to create a variable instances for keeping track of the number of objects created. This variable is incremented in the constructor. We will also create a dependency on the Movie class using the @Autowired annotation. Lastly, we will add a getter method for the dependency (getMovie) as shown below:

@Component  
public class ContentBasedFilter   
implements Filter {  
    //for keeping track of instances created  
    private static int instances= 0;  
   
    @Autowired  
    private Movie movie;  
  
    public ContentBasedFilter() {  
        instances++;        
        System.out.println("ContentBasedFilter constructor called");  
    }  
  
    public Movie getMovie() {  
        return movie;  
    }  
  
    public static int getInstances(){  
        return ContentBasedFilter.instances;  
    }  
}

1. In the main method, we will get the ContentBasedFilter bean from the application context and then use it to retrieve the Movie bean thrice.

public static void main(String[] args) {  
    ApplicationContext appContext = SpringApplication.run(  
                                          MovieRecommenderSystemApplication.class, args);  
  
    //Retrieve singleton bean from application context  
    ContentBasedFilter filter = appContext.getBean(ContentBasedFilter.class);     
    System.out.println("\nContentBasedFilter bean with singleton scope");  
    System.out.println(filter);  
      
    //Retrieve prototype bean from the singleton bean thrice  
    Movie movie1 = filter.getMovie();     
    Movie movie2 = filter.getMovie();     
    Movie movie3 = filter.getMovie();  
      
    System.out.println("\nMovie bean with prototype scope");  
    System.out.println(movie1);  
    System.out.println(movie2);  
    System.out.println(movie3);  
  
    //Print number of instances of each bean  
    System.out.println("\nContentBasedFilter instances created: "+  
                                                       ContentBasedFilter.getInstances());  
    System.out.println("Movie instances created: "+ Movie.getInstances());  
}

We expect one instance of ContentBasedFilter bean and three instances of the prototype Movie bean but the output is different, as can be seen by running the following code:

MovieRecommenderSystemApplication.java

Filter.java

Movie.java

ContentBasedFilter.java

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

package io.datajek.spring.basics.movierecommendersystem.lesson9;

import org.springframework.beans.factory.config.ConfigurableBeanFactory;

import org.springframework.context.annotation.Scope;

import org.springframework.stereotype.Component;

@Component

@Scope(ConfigurableBeanFactory.SCOPE\_PROTOTYPE)

public class Movie {

    private static int instances = 0;

    private int id;

    private String name;

    private String genre;

    private String producer;

    public Movie() {

        instances++;

        System.out.println("Movie constructor called");

    }

    public static int getInstances(){

        return Movie.instances;

    }

    public double movieSimilarity(int movie1, int movie2) {

        double similarity = 0.0;

        //if genres are same add 0.3 to similarity

        //if producers are same add 0.5 to similarity





Run

Save

Reset

The output of the above code widget shows that the same Movie bean is returned every time. Moreover, the number of instances of the prototype bean created is one instead of three. A singleton bean is created when the context is loaded. The Movie constructor was called by Spring when it was creating the ContentBasedFilter bean. The prototype bean is injected into the singleton bean at the time of creation of the singleton bean when the container initializes it. This explains the following messages in the output:

ContentBasedFilter constructor called

Movie constructor called

*When a prototype bean is injected into a singleton bean, it loses its prototype behavior and acts as a singleton.*

The same instance of the bean is returned by the application context every time it is requested using the getMovie method.

Prototype bean acting as a singleton bean is wired in at the time of container initialization

**Proxy**

1. Right now, Spring cannot inject the prototype bean into the singleton bean after it has been created. This problem can be solved in a number of ways. One of them is by using a **proxy**. We declare the bean with prototype scope as a proxy using the proxyMode element inside the @Scope annotation.

@Scope(value=ConfigurableBeanFactory.SCOPE\_PROTOTYPE, proxyMode=ScopedProxyMode.TARGET\_CLASS)

The prototype bean doesn’t get autowired into the singleton bean at the time of its creation. Instead, a proxy or placeholder object is autowired. The proxy adds a level of indirection. When the developer requests the prototype bean from Spring, a proxy is created and is returned by the application context. The proxy mode allows Spring container to inject a new object into the singleton bean when a method on the proxy object is called.

Proxy injected in place of prototype bean

After making this change in the code, a proxy Movie object is created and we get a new Movie bean when the ContentBasedFilter bean calls the getMovie() method on the proxy object. The proxy resolves the Movie instance and calls getMovie() on the resolved instance.

MovieRecommenderSystemApplication.java

Filter.java

Movie.java

ContentBasedFilter.java

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

package io.datajek.spring.basics.movierecommendersystem.lesson9;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ApplicationContext;

@SpringBootApplication

public class MovieRecommenderSystemApplication {

    public static void main(String[] args) {

        //ApplicationContext manages the beans and dependencies

        ApplicationContext appContext = SpringApplication.run(MovieRecommenderSystemApplication.class, args);

        //Retrieve and print singleton bean from application context

        ContentBasedFilter filter = appContext.getBean(ContentBasedFilter.class);

        System.out.println("\nContentBasedFilter bean with singleton scope");

        System.out.println(filter);

        //Retrieve and print prototype bean from the singleton bean twice

        Movie movie1 = filter.getMovie();

        Movie movie2 = filter.getMovie();

        Movie movie3 = filter.getMovie();

        System.out.println("\nMovie bean with prototype scope");

        System.out.println(movie1);

        System.out.println(movie2);

        System.out.println(movie3);

        //Print number of instances of each bean

        System.out.println("\nContentBasedFilter instances created: "+ ContentBasedFilter.getInstances());





Run

Save

Reset

As can be seen from the output, the singleton bean constructor is called when the ContentBasedFilter object is initialized, but the Movie constructor isn’t called at that time. The Movie constructor is called whenever the proxy object gets used (as in the println statements in line 26-28).

**@Lookup**

1. Another method is by using the @Lookup annotation on the getMovie() method. This annotation tells Spring to return an instance of Movie type. It is essentially the same as beanFactory.getBean(Movie.class).

One thing to consider is that singleton scope minimizes the number of objects created so the scope should only be changed where necessary. If there are more objects, there will be an impact on the memory used as well as on garbage collection.

# @ComponentScan

In this lesson, we will talk about how Spring searches for beans found in different packages.

**We'll cover the following**

* [@SpringBootApplication](https://www.educative.io/courses/guide-spring-5-spring-boot-2/mygjY6MDV30#@SpringBootApplication)
* [@ComponentScan for specific package](https://www.educative.io/courses/guide-spring-5-spring-boot-2/mygjY6MDV30#@ComponentScan-for-specific-package)
* [Include and exclude filters](https://www.educative.io/courses/guide-spring-5-spring-boot-2/mygjY6MDV30#Include-and-exclude-filters)
  + [Filter types](https://www.educative.io/courses/guide-spring-5-spring-boot-2/mygjY6MDV30#Filter-types)

Spring does a component scan to search for the beans that it manages. In a Spring application, the @ComponentScan annotation without any argument tells Spring to scan the current package as well as any sub-packages that might exist. Spring detects all classes marked with the @Component, @Repository, @Service, and @Controller annotations during component scan.

## @SpringBootApplication

In a Spring application, @ComponentScan is used along with the @Configuration annotation. In a Spring Boot application, component scan happens implicitly. The @SpringBootApplication annotation is a combination of three annotations:

* @Configuration
* @EnableAutoConfiguration
* @ComponentScan

'SpringBootApplication' combines three annotations

@SpringBootApplication by default, searches the package where it is present, as well as all the sub-packages. If a bean is present in a package other than the base package or its sub-packages, it will not be found. If we want Spring to find beans defined in other packages, we need to use the @ComponentScan annotation and provide the path of the package where we want Spring to look for the beans.

1. For the code example shown in this lesson, we have created a sub-package called **lesson10** inside the package **io.datajek.spring.basics.movierecommendersystem**. The package contains **Filter.java** and **CollaborativeFilter.java** files copied from **lesson8** package.

We will also use the **lesson9** sub-package from the previous lesson containing **MovieRecommenderSystemApplication.java**, **Filter.java**, **ContentBasedFilter.java**, and **Movie.java** files.

1. To demonstrate that Spring cannot find beans in other packages, we will use the MovieRecommenderSystemApplication class in the **lesson9** sub-package to access the ContentBasedFilter and CollaborativeFilter beans. The ContentBasedFilter bean is in the same package as the application file, but the CollaborativeFilter bean is present in a different package, namely: **io.datajek.spring.basics.movierecommendersystem.lesson10**.

CollaborativeFilter bean is not in the package having @SpringBootApplication annotation.

The MovieRecommenderSystemApplication class has the @SpringBootApplication annotation which means that **io.datajek.spring.basics.movierecommendersystem.lesson9** and its sub-packages will be scanned for beans. The containsBean method, which returns a Boolean value, is used to check if the bean is found during component scanning.

package io.datajek.spring.basics.movierecommendersystem.lesson9;  
  
@SpringBootApplication  
public class MovieRecommenderSystemApplication {  
  
    public static void main(String[] args) {  
        ApplicationContext appContext = SpringApplication.run(  
                                              MovieRecommenderSystemApplication.class, args);  
  
        System.out.println("ContentBasedFilter bean found = " +   
                                              appContext.containsBean("contentBasedFilter"));  
        System.out.println("CollaborativeFilter bean found = " +   
                                              appContext.containsBean("collaborativeFilter"));  
    }  
}

When the application is run, the ContentBasedFilter bean (from **lesson9** package) is found but the CollaborativeFilter bean (from **lesson10** package) is not found. This is because Spring scans the **io.datajek.spring.basics.movierecommendersystem.lesson9** package where the @SpringBootApplication annotation is used.

## @ComponentScan for specific package

1. We need to guide Spring to search the **io.datajek.spring.basics.movierecommendersystem.lesson10** package, where the CollaborativeFilter bean is present. The basePackages argument lists all the package names that are scanned during component scanning. We will use the @ComponentScan annotation with basePackages argument as follows:

@ComponentScan(basePackages = "io.datajek.spring.basics.movierecommendersystem.lesson10")

This time Spring detects the CollaborativeFilter bean but since we explicitly specified the package to be searched, it only looked for beans in that package only, and thus the ContentBasedFilter bean was not found.

ContentBasedFilter bean is not found during component scan.

If **lesson9** package is included in the basePackages list, as shown below, both beans will be found.

@ComponentScan(basePackages={"io.datajek.spring.basics.movierecommendersystem.lesson9",  
                             "io.datajek.spring.basics.movierecommendersystem.lesson10"})

## Include and exclude filters

1. @ComponentScan can be used to include or exclude certain packages from being scanned. Include filters are used to include certain classes in component scan. Exclude filters are used to stop Spring from auto-detecting classes in component scan.

### Filter types

There are different types of filters that make use of stereotype annotations, interfaces, regular expressions, and AspectJ expressions. Spring also allows the creation of custom filters, e.g., find only those beans whose names are a certain length. FilterType can have the following values:

* FilterType.ANNOTATION
* FilterType.ASPECTJ
* FilterType.ASSIGNABLE\_TYPE
* FilterType.REGEX
* FilterType.CUSTOM

One way to direct Spring to detect both ContentBasedFilter and CollaborativeFilter beans, is to use the include filter of type **REGEX** and provide the path of the package where the ContentBasedFilter bean is present.

@ComponentScan(basePackages = "io.datajek.spring.basics.movierecommendersystem.lesson10")  
@ComponentScan(includeFilters = @ComponentScan.Filter (  
                    type= FilterType.REGEX,   
                    pattern="io.datajek.spring.basics.movierecommendersystem.lesson9.\*"))

Our REGEX pattern evaluates to all beans declared with @Component annotation in the lesson9 package. Now when the application is run, beans from two different packages are successfully detected.

Both ContentBasedFilter and CollaborativeFilter beans are found in component scan.

Use the code widget below to run the application without @ComponentScan annotation and see the results. Then run with the @ComponentScan annotation providing the path to **lesson10** package, and observe the results. Lastly, run @ComponentScan with the includeFilters to see how it affects the output.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

package io.datajek.spring.basics.movierecommendersystem.lesson9;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.FilterType;

@SpringBootApplication

//@ComponentScan(basePackages="io.datajek.spring.basics.movierecommendersystem.lesson10")

//@ComponentScan(includeFilters = @ComponentScan.Filter (type= FilterType.REGEX, pattern="io.datajek.spring.basics.movierecommendersystem.lesson9.\*"))

public class MovieRecommenderSystemApplication {

    public static void main(String[] args) {

        //ApplicationContext manages the beans and dependencies

        ApplicationContext appContext = SpringApplication.run(MovieRecommenderSystemApplication.class, args);

        //check if beans are found

        System.out.println("CollaborativeFilter bean found = " + appContext.containsBean("collaborativeFilter"));

        System.out.println("ContentBasedFilter bean found = " + appContext.containsBean("contentBasedFilter"));

    }

}





Run

Save

Reset

**Bean Lifecycle: @PostConstruct, @ PreDestroy**

This lesson sheds light on bean lifecycle methods annotated with @PostConstruct and @PreDestroy.

**We'll cover the following**

* [@PostConstruct](https://www.educative.io/courses/guide-spring-5-spring-boot-2/Y5zwjNYBJy9#@PostConstruct)
* [@PreDestroy](https://www.educative.io/courses/guide-spring-5-spring-boot-2/Y5zwjNYBJy9#@PreDestroy)

Spring manages the entire lifecycle of beans from the time they are created till the time they are destroyed. It provides post-initialization and pre-destruction callback methods on the beans. The developer can tap into these callbacks and write custom initialization and cleanup code.

1. For the code example shown in this lesson, we have created a sub-package called **lesson11** inside the package **io.datajek.spring.basics.movierecommendersystem**.

The package contains **MovieRecommenderSystemApplication.java**, **Filter.java**, **ContentBasedFilter.java**, and **RecommenderImplementation.java** files.

1. In this lesson, we will introduce the slf4j logger to log the sequence of events in our application and use it instead of System.out.println. Loggers offer flexibility, as well as better output, that includes timestamp, name of the thread executing the code, and the name of the class. This information comes in handy when debugging applications. We will declare a logger and import the relevant files as follows:

import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
  
@Component  
public class RecommenderImplementation {  
  
    private Logger logger = LoggerFactory.getLogger(this.getClass());  
    //...  
}

Similarly, we will declare a logger in the ContentBasedFilter class as well.

1. Next, we will create a setter method in RecommenderImplementation to inject the Filter dependency and display a logging message as follows:

@Component  
public class RecommenderImplementation {  
  
    //...  
    private Filter filter;  
   
    @Autowired  
    public void setFilter(Filter filter) {  
     logger.info("In RecommenderImplementation setter method..dependency injection");  
     this.filter = filter;  
    }  
    //...  
}

**@PostConstruct**

1. When Spring creates a bean, the first thing it does, is to autowire the dependencies. If the developer wants to perform a task after the dependencies have been populated, it can be done by calling a method annotated with the @PostConstruct annotation. A method with this annotation works like the init method. The @PostConstruct annotation tells Spring to call the method for us once the object has been created. The method can have any name and its return type is always void. After the bean is created, we can initialize the contents of the bean, load data, establish a database connection, or connect to a web server. The post construct method is only called after all the dependencies have been populated.

Bean creation

1. We will create a method called postConstruct in the RecommenderImplementation class and use the @PostConstruct annotation on it. This method can have any name.

import javax.annotation.PostConstruct;  
  
@Component  
public class RecommenderImplementation {  
  
    //...  
  
    @PostConstruct  
    public void postConstruct() {  
        //initialization code goes here  
        logger.info("In RecommenderImplementation postConstruct method");  
    }  
  
    //...  
}

Similarly, for the ContentBasedFilter class, the constructor and post construct method are shown below:

@Component  
public class ContentBasedFilter implements Filter{  
    //...  
    public ContentBasedFilter() {    
        super();  
        logger.info("In ContentBasedFilter constructor method");          
    }  
   
    @PostConstruct  
    private void postConstruct() {  
        //load movies into cache  
        logger.info("In ContentBasedFilter postConstruct method");  
    }  
   
    //...  
}

1. In the main method, we will create a RecommenderImplementation bean and observe the order in which the constructor and post construct methods of the bean and its dependency are called.

public static void main(String[] args) {  
  
    ApplicationContext appContext = SpringApplication.run(  
                                          MovieRecommenderSystemApplication.class, args);  
      
    //use ApplicationContext to get recommender object  
    RecommenderImplementation recommender = appContext.getBean(  
                                                        RecommenderImplementation.class);     
  
    System.out.println(recommender);  
}

The logger messages give an insight into the sequence of events after the container is initialized. In order to create the RecommenderImplementation bean, the ContentBasedFilter bean is needed. The constructor of the ContentBasedFilter class is called. After the constructor method, the PostConstruct method is called. When the bean has been created, it is injected into the RecommenderImplementation bean as can be seen from the logger output of the setter method. After the dependency has been injected into the RecommenderImplementation bean, its PostConstruct method is called. Now the bean is ready for use and is returned by the getBean() method after which the bean name is printed.

Bean creation - sequence of events

The logger output is shown below:

**@PreDestroy**

1. The callback method that is executed just before the bean is destroyed is annotated using @PreDestroy. The method having this annotation is called when the bean is in the process of being removed from the container. All cleanup stuff can be performed in this method. A method with the @PreDestroy annotation can be used to release resources or close a database connection.

Bean destruction

1. We will write our custom destroy method in both the RecommenderImplementation and ContentBasedFilter classes as follows:

import javax.annotation.PreDestroy;  
  
@Component  
public class RecommenderImplementation {  
    //...  
  
    @PreDestroy  
    public void preDestroy() {  
        //cleanup code  
        logger.info("In RecommenderImplementation preDestroy method");  
    }  
}

@Component  
public class ContentBasedFilter   
implements Filter{  
  
    //...  
  
    @PreDestroy  
    private void preDestroy() {  
        //clear movies from cache  
        logger.info("In ContentBasedFilter preDestroy method");  
    }  
}

The log shows that when the RecommenderIplementation bean is in the process of being destroyed, its PreDestroy method is called followed by the ContentBasedFilter bean’s PreDestroy method.

MovieRecommenderSystemApplication.java

RecommenderImplementation.java

Filter.java

ContentBasedFilter.java

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

import java.util.Arrays;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ApplicationContext;

@SpringBootApplication

public class MovieRecommenderSystemApplication {

    public static void main(String[] args) {

        //ApplicationContext manages the beans and dependencies

        ApplicationContext appContext = SpringApplication.run(MovieRecommenderSystemApplication.class, args);

        //use ApplicationContext to get recommender object

        RecommenderImplementation recommender = appContext.getBean(RecommenderImplementation.class);

        System.out.println();

        System.out.println();

        System.out.println(recommender);

        //call method to get recommendations

        String[] result = recommender.recommendMovies("Finding Dory");

        //display results

        System.out.println();

        System.out.println();

        System.out.println(Arrays.toString(result));

    }

}





Run

Save

Reset

**Bean Lifecycle: Prototype Scoped Beans**

This lesson shows that Spring does not manage the complete lifecycle of prototype scoped beans.

**We'll cover the following**

* [Lifecycle of prototype beans](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JYEpQOAAYZ9#Lifecycle-of-prototype-beans)
* [Post-construct method](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JYEpQOAAYZ9#Post-construct-method)
* [Pre-destroy method](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JYEpQOAAYZ9#Pre-destroy-method)

**Lifecycle of prototype beans**

Spring manages the entire lifecycle of singleton beans but it does not completely manage the lifecycle of prototype beans. This is because there might be a large number of prototype instances and the container can become overwhelmed keeping track of them.

The Spring container creates the prototype beans and hands them over when requested. Thereafter, it is the responsibility of the application to destroy the bean and free up any resources that it has acquired.

1. For the code example shown in this lesson, we will use the **lesson11** sub-package created in the last lesson and add a class with prototype scope.
2. We will use the **Movie** class from the [bean scope](https://www.educative.io/pageeditor/5352985413550080/4785082259734528/6032092216426496) lesson to show that the post initialization method is called but the pre-destruction method is not called for prototype beans. This class has prototype scope as shown:

@Component  
@Scope(value = ConfigurableBeanFactory.SCOPE\_PROTOTYPE,   
       proxyMode = ScopedProxyMode.TARGET\_CLASS)  
public class Movie {  
  
    private Logger logger =   
    LoggerFactory.getLogger(this.getClass());  
   
    private int id;  
    private String name;  
    private String genre;    
    private String producer;   
    //...  
}

**Post-construct method**

1. We will add a constructor and post construct method to this class as follows:

public Movie() {  
    super();  
    logger.info("In Movie constructor method");       
}  
  
@PostConstruct  
private void postConstruct() {  
    //initialization code  
    logger.info("In Movie postConstruct method");  
}

**Pre-destroy method**

1. The Movie class will have a pre-destroy method, preDestroy as follows:

@PreDestroy  
private void preDestroy() {  
    //cleanup code  
    logger.info("In Movie preDestroy method");  
}

1. In the main method, we will retrieve a singleton RecommenderImplementation and two prototype Movie beans from the application context.

public static void main(String[] args) {  
  
    ApplicationContext appContext = SpringApplication.run(  
                                          MovieRecommenderSystemApplication.class, args);  
  
    //Retrieving singleton bean from application context  
    RecommenderImplementation recommender = appContext.getBean(  
                                                        RecommenderImplementation.class);     
    System.out.println(recommender);  
  
    //Retrieving prototype bean from application context twice  
    Movie m1 = appContext.getBean(Movie.class);  
    System.out.println(m1);  
  
    Movie m2 = appContext.getBean(Movie.class);  
    System.out.println(m2);  
  
}

1. When the application is run, we can see that the constructor and post construct methods of the singleton RecommenderImplementation bean (and its dependency, ContentBasedFilter bean) are called when the bean is created, before the application starts.

The prototype bean is not created beforehand and the constructor and post construct methods for the Movie bean are only called when we request the application context for the Movie bean.

When the application terminates, the PreDestroy method is called for the singleton RecommenderImplementation bean (and its dependency ContentBasedFilter bean) but not for the prototype scoped Movie bean.

MovieRecommenderSystemApplication.java

Movie.java

RecommenderImplementation.java

Filter.java

ContentBasedFilter.java

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ApplicationContext;

@SpringBootApplication

public class MovieRecommenderSystemApplication {

    public static void main(String[] args) {

        //ApplicationContext manages the beans and dependencies

        ApplicationContext appContext = SpringApplication.run(MovieRecommenderSystemApplication.class, args);

        System.out.println();

        //Retrieving singleton bean from application context

        RecommenderImplementation recommender = appContext.getBean(RecommenderImplementation.class);

        System.out.println(recommender);

        //Retrieving prototype bean from application context twice

        System.out.println();

        Movie m1 = appContext.getBean(Movie.class);

        System.out.println(m1);

        System.out.println();

        Movie m2 = appContext.getBean(Movie.class);

        System.out.println(m2);

        System.out.println();

    }

}





Run

Save

Reset

**Contexts and Dependency Injection Framework**

Learn how to replace Spring annotations with CDI annotations.

**We'll cover the following**

* [@Named](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qV2A6Vn5rxr#@Named)
* [@Inject](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qV2A6Vn5rxr#@Inject)
* [Other CDI annotations](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qV2A6Vn5rxr#Other-CDI-annotations)

Contexts and Dependency Injection (CDI) is an interface that standardizes dependency injection for Java EE. It defines different annotations for dependency injection like @Named, @Inject, @Scope, @Singleton, etc. Different CDI implementation frameworks provide support and functionality for these annotations.

@Named is used to define a bean and @Inject is used for autowiring one bean into another. Spring supports most of the annotations defined by CDI.

1. For the code example shown in this lesson, we have created a sub-package called **lesson12** inside the package **io.datajek.spring.basics.movierecommendersystem**.

The package contains **MovieRecommenderSystemApplication.java**, **Filter.java**, **ContentBasedFilter.java**, **CollaborativeFilter.java**, and **RecommenderImplementation.java** files.

1. To be able to use CDI annotations in our Spring application, we need to add a dependency in the pom.xml file as follows:

<dependency>  
    <groupId>javax.inject</groupId>  
    <artifactId>javax.inject</artifactId>  
    <version>1</version>  
</dependency>

The dependency is automatically downloaded when the pom.xml file is saved and is visible in the **Maven Dependencies** folder as javax.inject-1.jar. This jar lists the annotations defined by CDI. Now, we can use these annotations in our application.

**@Named**

1. Spring framework provides an implementation for some CDI annotations. In Spring framework, a bean is declared using the @Component annotation. However, it also supports the @Named annotation. This means we can replace the @Component from the RecommenderImplementation, ContentBasedFilter, and CollaborativeFilter classes and use @Named to declare components.

import javax.inject.Named;  
  
@Named  
public class RecommenderImplementation {  
    //...  
}

@Named  
public class ContentBasedFilter {  
    //...  
}

@Named  
public class CollaborativeFilter {  
    //...  
}

**@Inject**

1. Spring has the @Autowired annotation for dependency injection but it also supports the equivalent CDI annotation, @Inject. So, the Filter dependency in RecommenderImplementation class declared using @Autowired can be declared using @Inject.

import javax.inject.Inject;  
  
@Named  
public class RecommenderImplementation {  
    @Inject  
    @Qualifier("CF")  
    private Filter filter;  
    //...  
}

1. In the main method, we will print the beans to verify if the RecommenderImplementation bean is getting detected and the Filter dependency is getting autowired successfully.

public static void main(String[] args) {  
    //...  
    RecommenderImplementation recommender = appContext.getBean(  
                                                       RecommenderImplementation.class);  
    System.out.println(recommender);  
    System.out.println(recommender.getFilter());  
    //...     
}

When the application is run, the output is the same as when @Component and @Autowired annotations were used.

**Other CDI annotations**

1. Other annotations provided by CDI are @Qualifier, @Scope, and @Singleton. The @Qualifier annotation is used to break ties if two beans of the same type qualify to be injected in a dependency.

@Scope is used to set the scope of the bean, similar to the @Scope annotation in Spring. The @Singleton annotation is used to explicitly set the scope to singleton in CDI annotation. In Spring, we can specify singleton scope using the @Scope annotation.

Both Spring and CDI annotations provide the same functionality. The only difference is that if the application is migrated to another framework, the CDI annotations can still be used, whereas Spring annotations are specific to the Spring framework. Thus, CDI annotations are often preferred because CDI is a Java EE standard.

MovieRecommenderSystemApplication.java

RecommenderImplementation.java

Filter.java

ContentBasedFilter.java

CollaborativeFilter.java

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

package io.datajek.spring.basics.movierecommendersystem.lesson12;

import java.util.Arrays;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ApplicationContext;

@SpringBootApplication

public class MovieRecommenderSystemApplication {

    public static void main(String[] args) {

        //ApplicationContext manages the beans and dependencies

        ApplicationContext appContext = SpringApplication.run(MovieRecommenderSystemApplication.class, args);

        //use ApplicationContext to find which filter is being used

        RecommenderImplementation recommender = appContext.getBean(RecommenderImplementation.class);

        System.out.println(recommender);

        System.out.println(recommender.getFilter());

        //call method to get recommendations

        String[] result = recommender.recommendMovies("Finding Dory");

        //display results

        System.out.println(Arrays.toString(result));

    }

}





Run

Save

Reset

**Spring Application Configuration**

Learn how to configure a Spring application.

**We'll cover the following**

* [spring-core dependency](https://www.educative.io/courses/guide-spring-5-spring-boot-2/39WvjGDqwz9#spring-core-dependency)
* [spring-context dependency](https://www.educative.io/courses/guide-spring-5-spring-boot-2/39WvjGDqwz9#spring-context-dependency)
* [@Configuration](https://www.educative.io/courses/guide-spring-5-spring-boot-2/39WvjGDqwz9#@Configuration)
* [AnnotationConfigApplicationContext](https://www.educative.io/courses/guide-spring-5-spring-boot-2/39WvjGDqwz9#AnnotationConfigApplicationContext)
* [@ComponentScan](https://www.educative.io/courses/guide-spring-5-spring-boot-2/39WvjGDqwz9#@ComponentScan)
* [Closing the application context](https://www.educative.io/courses/guide-spring-5-spring-boot-2/39WvjGDqwz9#Closing-the-application-context)

We created the Movie Recommender System using Spring Boot. Let’s look at how it can be run using the core features of Spring.

We will remove the dependencies autoconfigured by Spring Boot and replace them with the dependencies needed for a Spring project.

1. For the code example shown in this lesson, we have created a sub-package called **lesson13** inside the package **io.datajek.spring.basics.movierecommendersystem**.

The package contains **MovieRecommenderSystemApplication.java**, **Filter.java**, **ContentBasedFilter.java**, **CollaborativeFilter.java**, and **RecommenderImplementation.java** files.

**spring-core dependency**

1. spring-core provides the fundamental features of Spring framework like dependency injection and Inversion of Control.

Since we created our application using Spring Initializr, it has the spring-boot-starter dependency in the pom.xml file. This dependency brings in Spring Boot functionality. In this lesson, we will replace it with spring-core.

<!-- remove the following dependency:  
<dependency>  
<groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot- starter</artifactId>  
</dependency>     
-->  
  
<!-- replace it with this dependency: -->  
<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-core</artifactId>  
</dependency>

spring-core defines the bean factory and forms the base of the Spring framework.

**spring-context dependency**

1. To be able to use ApplicationContext, we need to add another dependency called spring-context as follows:

<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-context</artifactId>  
</dependency>

By removing the spring-boot-starter dependency, the @SpringBootApplication annotation will no longer be available when we run the Java application. However, this annotation can be used in test scope because we still have the spring-boot-starter-test dependency in **pom.xml**.

**@Configuration**

1. @SpringBootApplication cannot be used in the Java application file anymore as we have replaced the spring-boot-starter dependency with spring-core in step 1. This annotation defined the application configuration in Spring Boot. In the Java realm, we use @Configuration and import the **org.springframework.context.annotation.Configuration** jar.

**AnnotationConfigApplicationContext**

1. If we run the application now, the following compilation error is encountered:

Unresolved compilation problem: SpringApplication cannot be resolved.

The SpringApplication class creates the application context. It belongs to the **org.springframework-boot** package. When using the spring-core, the application context is created using AnnotationConfigApplicationContext class as follows:

//ApplicationContext manages the beans and dependencies  
  
/\*Change this:  
ApplicationContext appContext =                
     SpringApplication.run(MovieRecommenderSystemApplication.class, args);  
\*/  
  
//to this:  
AnnotationConfigApplicationContext appContext =   
        new AnnotationConfigApplicationContext(MovieRecommenderSystemApplication.class);

**@ComponentScan**

1. If we try to run the application now, it throws the NoSuchBeanDefinition exception, which means that the application context is unable to locate beans declared using @Component. We need to help it in component scanning by providing the @ComponentScan annotation on the MovieRecommenderSystemApplication class.

Look at the following image:

@SpringBootApplication combines three annotations

The @SpringBootApplication annotation performs multiple tasks. Now that it has been removed, we need to explicitly use the @ComponentScan annotation to guide Spring to the package which contains the beans.

After making these changes, we are able to run the same application without using Spring Boot!

**Closing the application context**

1. Spring Boot automatically closes the application context in the end, but now we need to close the context as follows:

//close the application context  
appContext.close();

Another way is to use a try catch block around the statement creating appContext. In this way, any error will result in the context being automatically closed.

try( AnnotationConfigApplicationContext appContext =   
  new AnnotationConfigApplicationContext(MovieRecommenderSystemApplication.class)) {  
    //...  
}

The changes that we made in this lesson that enabled us to run a Spring Boot application as a basic Spring application are as follows:

* + Removing the spring-boot starter dependency and replacing it with spring-core and spring-context.
  + Replacing @SpringBootApplication with @Configuration and @ComponentScan.
  + Replacing the SpringApplication class with the AnnotationConfigApplicationContext class.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

package io.datajek.spring.basics.movierecommendersystem.lesson13;

import java.util.Arrays;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

//@SpringBootApplication

@Configuration

@ComponentScan

public class MovieRecommenderSystemApplication {

    public static void main(String[] args) {

        //ApplicationContext manages the beans and dependencies

        //ApplicationContext appContext = SpringApplication.run(MovieRecommenderSystemApplication.class, args);

        AnnotationConfigApplicationContext appContext = new AnnotationConfigApplicationContext(MovieRecommenderSystemApplication.class);

        //use ApplicationContext to find which filter is being used

        RecommenderImplementation recommender = appContext.getBean(RecommenderImplementation.class);

        //call method to get recommendations

        String[] result = recommender.recommendMovies("Finding Dory");

        //display results

        System.out.println(Arrays.toString(result));

        appContext.close();

    }





Run

Save

Reset

# XML Application Configuration

This lesson shows how to replace Spring annotations with XML configuration.

**We'll cover the following**

* [XML configuration file](https://www.educative.io/courses/guide-spring-5-spring-boot-2/3YPDyZVv4Kn#XML-configuration-file)
* [<bean> tag](https://www.educative.io/courses/guide-spring-5-spring-boot-2/3YPDyZVv4Kn#%3Cbean%3E-tag)
* [ClassPathXmlApplicationContext](https://www.educative.io/courses/guide-spring-5-spring-boot-2/3YPDyZVv4Kn#ClassPathXmlApplicationContext)
* [Dependency injection](https://www.educative.io/courses/guide-spring-5-spring-boot-2/3YPDyZVv4Kn#Dependency-injection)
  + [Using <property> tag](https://www.educative.io/courses/guide-spring-5-spring-boot-2/3YPDyZVv4Kn#Using-%3Cproperty%3E-tag)
  + [Using constructor injection](https://www.educative.io/courses/guide-spring-5-spring-boot-2/3YPDyZVv4Kn#Using-constructor-injection)
* [Closing the context](https://www.educative.io/courses/guide-spring-5-spring-boot-2/3YPDyZVv4Kn#Closing-the-context)

In this lesson, we will show how to configure a Spring application using XML. We will remove annotations from our application altogether and use XML to define beans and dependencies.

1. For the code example shown in this lesson, we have created a sub-package called **lesson14** inside the package **io.datajek.spring.basics.movierecommendersystem**.

The package contains **MovieRecommenderSystemApplication.java**, **Filter.java**, **ContentBasedFilter.java**, **CollaborativeFilter.java**, and **RecommenderImplementation.java** files copied from the previous lesson.

1. We will remove all annotations from the code. This includes the @Configuration and @ComponentScan annotations from the MovieRecommenderSystemApplication class and @Component, @Qualifier, and @Autowired annotations from RecommenderImplementation, ContentBasedFilter, and CollaborativeFilter classes.

## XML configuration file

1. The first step is creating an XML file that contains the bean definitions. Spring will read this file and know which beans to create and manage. We will create an XML file in **src/main/resources** and call it **appContext.xml**.

First, we need to provide some metadata for validating the tags which will be used in this file. The metadata defines the schema location of the tags, as shown:

<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                      
       http://www.springframework.org/schema/beans/spring-beans.xsd">  
  
<!-- bean definitions -->  
</beans>

## <bean> tag

1. Next, we will define the beans inside this metadata using the <bean> </bean> tag. For every bean, we need to specify the fully-qualified class name as well as a reference id. The fully-qualified class name is the class name along with its package name. We used @Component at three places in our application. Now we will declare three beans as follows:

<bean id="contentBasedFilter"   
class="io.datajek.spring.basics.movierecommendersystem.lesson14.ContentBasedFilter">  
</bean>  
  
<bean id="collaborativeFilter"   
class="io.datajek.spring.basics.movierecommendersystem.lesson14.CollaborativeFilter">   
</bean>  
  
<bean id="recommenderImplementation"       
class="io.datajek.spring.basics.movierecommendersystem.lesson14.RecommenderImplementation">  
</bean>

The IOC container will read the **appContext.xml** file and create objects of the classes mentioned in it. It will call the constructor of the class to create the object by giving it the name that we specified as the **id**. Hence, the following line:

<bean id="contentBasedFilter"   
class="io.datajek.spring.basics.movierecommendersystem.lesson14.ContentBasedFilter">     
</bean >

translates to:

ContentBasedFilter contentBasedFilter = new ContentBasedFilter();

After reading **appContext.xml** file, the IOC container creates the following beans:

## ClassPathXmlApplicationContext

1. Inside the main method, the application context will be created using ClassPathXmlApplicationContext by providing the name of the XML config file as an argument:

ClassPathXmlApplicationContext appContext =   
                               new ClassPathXmlApplicationContext("appContext.xml");

Once the context is loaded, we can check which beans are present using the getBeanDefinitionNames method:

//check the beans which have been loaded  
System.out.println("\nBeans loaded:");  
System.out.println(Arrays.toString(appContext.getBeanDefinitionNames()));

We can also retrieve a bean using the getBeans method by providing the bean id (defined in the **appContext.xml** file) along with the class name as follows:

//retrieve bean from the application context  
RecommenderImplementation recommender = appContext.getBean("recommenderImplementation",   
                                                            RecommenderImplementation.class);

## Dependency injection

1. In our application, Filter is a dependency of RecommenderImplementation. In XML, a dependency can be defined in multiple ways:

### Using <property> tag

We can create getter and setter methods for the filter dependency as follows:

public class RecommenderImplementation {  
   
    private Filter filter;  
  
    public Filter getFilter() {  
        return filter;  
    }  
  
    public void setFilter(Filter filter)   
    {  
        this.filter = filter;  
    }  
    //...  
}

To define this dependency, we will modify the bean tag created in step 2 and use <property> tag for the dependency as follows:

<bean id="recommenderImplementation"       
 class="io.datajek.spring.basics.movierecommendersystem.lesson14.RecommenderImplementation">  
 <property name="filter" ref="collaborativeFilter"/>   
</bean>

The setter method setFilter will be used to set the filter dependency in the recommenderImplementation object by passing the reference of the collaborativeFilter object. We can change the ref property to use the contentBasedFilter bean as well.

A no-arg constructor is needed for the above mentioned dependency injection to work, if the RecommenderImplementation class has a constructor for setting filter dependency.

### Using constructor injection

Instead of using the setter method, we can define a constructor in the RecommenderImplementation class as follows:

public class RecommenderImplementation {  
   
    private Filter filter;  
  
    public RecommenderImplementation(Filter filter) {  
        this.filter = filter;  
    }  
    //...  
}

To use the constructor method defined above, we will use the <constructor-arg> tag in the bean tag created in step 2 as follows:

<bean id="recommenderImplementation"       
 class="io.datajek.spring.basics.movierecommendersystem.lesson14.RecommenderImplementation">  
 <constructor-arg ref="collaborativeFilter"/>   
</bean>

This tells Spring to pass a bean called collaborativeFilter as a constructor argument when creating the recommenderImplementation object. We can change the name of the argument to autowire contentBasedFilter as well.

1. After dependency injection, we can run the application to find movie recommendations in the main method:

//retrieve bean from the application context  
RecommenderImplementation recommender = appContext.getBean("recommenderImplementation",   
                                                            RecommenderImplementation.class);  
//call method to get recommendations  
String[] result = recommender.recommendMovies("Finding Dory");  
           
//display results  
System.out.println(Arrays.toString(result));

From the output, it can be seen that the dependency has been autowired.

## Closing the context

1. The last step is to close the context.

appContext.close();

We are now ready to run the application using XML configuration.

MovieRecommenderSystemApplication.java

RecommenderImplementation.java

appContext.xml

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

<?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

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

<!-- bean definitions -->

      <bean id="contentBasedFilter"

            class="io.datajek.spring.basics.movierecommendersystem.lesson14.ContentBasedFilter">

      </bean>

      <bean id="collaborativeFilter"

            class="io.datajek.spring.basics.movierecommendersystem.lesson14.CollaborativeFilter">

      </bean>

      <bean id="recommenderImplementation"

            class="io.datajek.spring.basics.movierecommendersystem.lesson14.RecommenderImplementation">

            <!--<property name="filter" ref="collaborativeFilter"/> -->

            <constructor-arg ref="collaborativeFilter"/>

      </bean>

</beans>





Run

Save

Reset

**XML Configuration with Java Annotations**

Learn how to use a combination of XML context and Java annotations.

**We'll cover the following**

* [<context:component-scan> tag](https://www.educative.io/courses/guide-spring-5-spring-boot-2/gx9oXYkZQ79#%3Ccontext:component-scan%3E-tag)
* [<context:annotation-config> tag](https://www.educative.io/courses/guide-spring-5-spring-boot-2/gx9oXYkZQ79#%3Ccontext:annotation-config%3E-tag)

In the last lesson, we removed all Java annotations from our application and used the **appContext.xml** file to define beans and inject the dependency. However, if we want to detect beans defined by the @Component annotation and inject the dependencies using @Autowired annotation while using XML context, we can do that too.

In large projects, declaring a lot of beans using the <bean> tag is cumbersome, so annotation-based dependency injection was introduced in Spring 2.5. This enabled automatic detection of beans having the @Component annotation. The <context:component-scan> tag is used to turn this feature on.

1. For the code example shown in this lesson, we use the **lesson14** package from the previous lesson.

**<context:component-scan> tag**

1. Right now, we have declared three beans in **appContext.xml**. Suppose we want to declare the **ContentBasedFilter** and CollaborativeFilter beans using the @Component annotation instead of defining them using the <bean> tag in **appContext.xml**.

@Component  
public class ContentBasedFilter implements Filter {  
 //..  
}

@Component  
public class CollaborativeFilter implements Filter {  
 //..  
}

Just annotating the classes with @Component isn’t enough for Spring to detect them as beans. We need to trigger a component scan. In XML context, the <context:component-scan> tag is used to activate component scanning. To be able to use this tag, we will define a new schema and provide a shortcut name for it as context in **appContext.xml**.

By default, any tag that is used without any namespace (like <bean>) belongs to the default schema, which is <http://www.springframework.org/schema/beans>.

In the code shown below, line 3 defines the context namespace and lines 6 and 7 provide the schema location of the namespace.

1.<beans xmlns="http://www.springframework.org/schema/beans"  
2. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
3. xmlns:context="http://www.springframework.org/schema/context"  
4. xsi:schemaLocation="http://www.springframework.org/schema/beans  
5. http://www.springframework.org/schema/beans/spring-beans.xsd  
6. http://www.springframework.org/schema/context  
7. http://www.springframework.org/schema/context/spring-context.xsd">

The above code will replace the existing metadata information in **appContext.xml**.

1. Now, we can use the <context:component-scan> tag and remove the <bean> entries for the classes having @Component annotation from the configuration file.

<!-- enable component scan -->  
<context:component-scan  
 base-package="io.datajek.spring.basics.movierecommendersystem.lesson14" />  
  
<bean id="recommenderImpl"   
 class="io.datajek.spring.basics.movierecommendersystem.lesson14.RecommenderImplementation">  
 <property name="filter" ref="collaborativeFilter"/>  
</bean>

Here, we are directing Spring to scan the **io.datajek.spring.basics.movierecommendersystem.lesson14** package. By default, Spring will scan everything marked with @Component as well as the @Controller, @Repository, and @Service annotations.

1. In the main method, we will print the list of beans and also check if autowiring is taking place. The recommenderImpl bean is declared in **appContext.xml** while its dependency is declared using @Component annotation.

public static void main(String[] args) {  
      
    ClassPathXmlApplicationContext appContext = new ClassPathXmlApplicationContext(  
                                                                    "appContext.xml");  
    //check the beans which have been loaded  
    System.out.println("\nBeans loaded:");  
    System.out.println(Arrays.toString(appContext.getBeanDefinitionNames()));  
  
    //retrieve bean from the application context  
    RecommenderImplementation recommender =   
    appContext.getBean("recommenderImpl", RecommenderImplementation.class);  
  
    //print dependency  
    System.out.println("\nDependency: " + recommender.getFilter());  
    System.out.println();  
   
    appContext.close();  
}

When the application is run, all beans defined using the <bean> tag and @Component annotation are detected. The bean names are displayed using the getBeanDefinitionNames() method. As can be seen from the output, the list of beans is quite long. This is because the output shows all the beans in the IOC container. Autowiring is also taking place as expected.

1. Just like XML context detected the @Component annotation, it can also detect the @Autowired annotation. Right now, we are using <property> tag for injecting collaborativeFilter bean in recommenderImpl. We can remove it and instead use the @Autowired annotation in the RecommenderImplementation class as follows:

public class RecommenderImplementation {  
   
    @Autowired  
    @Qualifier("contentBasedFilter")  
    private Filter filter;  
    //...  
}

We are using the @Qualifier annotation to break the tie between two beans of Filter type.

After removing dependency injection, the updated bean definition in **appContext.xml** is:

<bean id="recommenderImpl"   
 class="io.datajek.spring.basics.movierecommendersystem.lesson14.RecommenderImplementation"/>

When the application is run, the @Autowired annotation is detected by the XML context and contentBasedFilter bean gets autowrired.

MovieRecommenderSystemApplication.java

RecommenderImplementation.java

appContext.xml

ContentBasedFilter.java

CollaborativeFilter.java

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

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

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

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

    xmlns:context="http://www.springframework.org/schema/context"

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

                     http://www.springframework.org/schema/beans/spring-beans.xsd

                     http://www.springframework.org/schema/context

                     http://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan

        base-package="io.datajek.spring.basics.movierecommendersystem.lesson14" />

    <bean id="recommenderImpl"

        class="io.datajek.spring.basics.movierecommendersystem.lesson14.RecommenderImplementation" />

     <!--    <property name="filter" ref="collaborativeFilter"/>

    </bean> -->

</beans>





Run

Save

Reset

**<context:annotation-config> tag**

1. Like the <context:component-scan> tag, the <context:annotation-config> tag can also detect dependency injection annotations, e.g., @Autowired and @Qualifier. But it cannot detect beans annotated with @Component and other stereotype annotations.

If we are using the <context:annotation-config> tag, then we need to declare beans in the XML config file.

<!-- enable detection of dependency injection annotations -->   
<context:annotation-config />  
  
<!-- bean definitions -->  
<bean id="filter"     
class="io.datajek.spring.basics.movierecommendersystem.lesson14.ContentBasedFilter" />   
  
<bean id="filter1"   
class="io.datajek.spring.basics.movierecommendersystem.lesson14.CollaborativeFilter"/>  
      
<bean id="recommenderImpl"  
class="io.datajek.spring.basics.movierecommendersystem.lesson14.RecommenderImplementation"/>

1. Now that we have enabled the detection of dependency injection annotation in the **appContext.xml**, we can use the @Autowired annotation in RecommenderImplementation class as follows:

public class RecommenderImplementation {  
   
    @Autowired  
    private Filter filter;  
  
    //getter setter  
    public Filter getFilter() {  
        return filter;  
    }  
  
    public void setFilter(Filter filter) {  
        this.filter = filter;  
    }  
  
    //...  
}

Here, autowiring is taking place by name as we have declared a bean named filter in **appContext.xml**.

When the application is run, dependency injection takes place due to the @Autowired annotation.

In this lesson, we briefly touched upon ways to use XML context with Java annotations. The <context:annotation-config> tag is redundant in the presence of <context:component-scan> tag as the latter can detect both bean and dependency injection annotations.

MovieRecommenderSystemApplication.java

RecommenderImplementation.java

appContext.xml

ContentBasedFilter.java

CollaborativeFilter.java

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

package io.datajek.spring.basics.movierecommendersystem.lesson14;

import java.util.Arrays;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MovieRecommenderSystemApplication {

    public static void main(String[] args) {

        // ApplicationContext manages the beans and dependencies

        ClassPathXmlApplicationContext appContext = new ClassPathXmlApplicationContext("appContext.xml");

        //check the beans which have been loaded

        System.out.println("\nBeans loaded:");

        System.out.println(Arrays.toString(appContext.getBeanDefinitionNames()));

        //retrieve bean from the application context

        RecommenderImplementation recommender =

                   appContext.getBean("recommenderImpl", RecommenderImplementation.class);

        //print dependency

        System.out.println("\nDependency: " + recommender.getFilter());

        System.out.println();

        appContext.close();

    }

}





Run

Save

Reset

**Stereotype Annotations**

Let's explore the different annotations that can be used in place of the generic @Component annotation and their usage.

**We'll cover the following**

* [@Controller](https://www.educative.io/courses/guide-spring-5-spring-boot-2/mEXOQQOXz9R#@Controller)
* [@Service](https://www.educative.io/courses/guide-spring-5-spring-boot-2/mEXOQQOXz9R#@Service)
* [@Repository](https://www.educative.io/courses/guide-spring-5-spring-boot-2/mEXOQQOXz9R#@Repository)

Beans can be declared using the @Component annotation and the <bean> tag. But there are other ways to define beans. We will look at some of them in this lesson.

Beans can be declared using the @Bean annotation in a configuration class or by using the @Controller, @Service, and @Repository annotations. These annotations are used at different layers of an enterprise application. A typical application has the following layers:

The web or UI layer interacts with the client program, the service layer provides an abstraction between the web and data access layer as well as taking care of the business logic, and the data layer interacts with a database or an external interface. @Component is a generic annotation. It can be used in any layer, if the developer is unsure about where the bean belongs. The other three annotations, @Controller, @Service, and @Repository, are specific to layers.

Stereotype annotations

**@Controller**

@Controller is used to define a controller in the web layer. Spring scans a class with @Controller to find methods that are mapped to different HTTP requests. It makes sure that the right view is rendered to the user. @RestController is a specialized form of @Controller.

**@Service**

@Service is used in the business layer for objects that define the business logic. It marks a class as a service provider.

**@Repository**

@Respository is used in the data layer to encapsulate storage, retrieval, and search in a typical database. This annotation can also be used for other external sources of data.

Create a new sub-package **lesson-16** and copy all files from the **lesson-11** sub-package. Here, the RecommenderImplementation class contains the business logic. The filter algorithm contained in the ContentBasedFilter class also falls in the category of business logic. The Movie class is responsible for loading movie data from a movie repository as well as comparing movies to find similarity so it represents the data access layer. Note that, we haven’t written the logic of retrieving movie data yet. All the above mentioned classes contain the @Component annotation.

We can replace the @Component annotation with stereotype annotations. Instead of @Component, we can use the @Service annotation for classes with business logic i.e., RecommenderImplementation and ContentBasedFilter. Similarly, if we had the CollaborativeFilter class in the package, we could annotate it with @Service too.

For the Movie class, we can replace the @Component annotation with @Repository annotation as it belongs to the data access layer.

In the code below, we have removed the post-construct and pre-destroy method calls from the classes. The main method in the MovieRecommenderSystemApplication class simply retrieves the RecommenderImplementation bean from the application context, calls the recommendMovies method on it and displays the list of movies returned.

It can be seen that there is no change in terms of output when the stereotype annotations replace the @Component annotation.

MovieRecommenderSystemApplication.java

RecommenderImplementation.java

Filter.java

Movie.java

ContentBasedFilter.java

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

package io.datajek.spring.basics.movierecommendersystem.lesson16;

import java.util.Arrays;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.FilterType;

@SpringBootApplication

public class MovieRecommenderSystemApplication {

    public static void main(String[] args) {

        //ApplicationContext manages the beans and dependencies

        ApplicationContext appContext = SpringApplication.run(MovieRecommenderSystemApplication.class, args);

        //retrieve bean from the application context

        RecommenderImplementation recommender = appContext.getBean(RecommenderImplementation.class);

        //call method to get recommendations

        String[] result = recommender.recommendMovies("Finding Dory");

        //display results

        System.out.println(Arrays.toString(result));

    }

}





Run

Save

Reset

Right now, there is no web layer in our application, but we can have a RecommendationController class marked with @Controller that calls the classes in the business layer. Likewise, we should also have a User class marked with @Repository to hold the user watch history and movie preference data for our recommender system.

Movie recommender system

The advantage of having annotations specific to every layer instead of the generic @Component is in Aspect Oriented Programming (AOP). We can identify the annotations and add functionality to specific annotations. Similarly, we can classify components in different categories and apply a specific logic to each category. For example, if we want to log everything that is coming from the business layer, we can identify objects with the @Service annotation using AOP and log all the content.

Spring provides a default exception translation facility for JDBC exceptions if the @Repository annotation is used. This feature cannot be used on beans annotated with @Component. When using a persistence framework like Hibernate, exceptions thrown in a class with the @Repository annotation are caught and automatically translated into Spring’s DataAccessException class.

Likewise, the request mapping feature is enabled only when using the @Controller annotation. The DispatcherServlet automatically looks for @RequestMapping for classes marked with the @Controller annotation only.

The @Controller, @Service, and @Repository annotations are similar to @Component annotation with respect to bean creation and dependency injection, except that they provide specialized functionality.

**Using an External Property File**

Learn how to use an external property file in your Java application.

**We'll cover the following**

* [application-properties file](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JYVvWgKYAwv#application-properties-file)
* [@Value](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JYVvWgKYAwv#@Value)
* [@PropertySource](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JYVvWgKYAwv#@PropertySource)

Applications have a lot of configuration and keeping it separate from the code leads to clarity. If the property file is inside the jar when the application is built, it can’t be changed later without having to un-compress the jar. Spring provides a way to change configuration while the application jar remains intact.

The application configuration is different in different environments. Local machines are used in development, then the application moves to a test environment, and afterwards a production server is used. To use different configurations in different environments, a property file is used to externalize the values for each environment. For example, the database connection in the data layer might be different in different environments. The developer can create an application with default properties and deploy it. Then, if the values need to be changed, an external property file will do the trick. The values from the external property file outside the jar overwrite the values inside the jar.

**application-properties file**

The application-properties file is a text file that defines the key-value pair for a property. The name of the property follows a convention where the class name is used with the property name to disambiguate properties with the same name in different classes.

1. For the code example shown in this lesson, we have created a sub-package called **lesson15** inside the package **io.datajek.spring.basics.movierecommendersystem**.

The package contains **MovieRecommenderSystemApplication.java**, **Filter.java**, **ContentBasedFilter.java**, **CollaborativeFilter.java**, and **RecommenderImplementation.java** files copied from **lesson6** sub-package.

1. Our movie recommender system has been using a hard coded input string, “Finding Dory”, for the movie. We can create a variable favoriteMovie in RecommenderImplementation class for the input string as follows:

String favoriteMovie = "Finding Dory";  
  
public String getFavoriteMovie() {  
    return favoriteMovie;  
}  
  
//...

We will dynamically fetch the value of favoriteMovie from a property file. To make sure that the value has been read from the file, we have created a getter method getFavoriteMovie that returns the value read.

1. We will create a file called app.properties in **src/main/resources** and define a value for favoriteMovie in it. Every line in this file defines a value for a property key using the equality (=) sign. As per convention, the fully-qualified name of the variable should be used.

recommender.implementation.favoriteMovie = Finding Dory

**@Value**

1. We can now dynamically fetch the favoriteMovie value from the file using the @Value annotation. This annotation can be used with the constructor or setter method. It can also be used directly on the variable using the following syntax:

@Value("${recommender.implementation.favoriteMovie}")  
String favoriteMovie;

If the property value is not found in app.properties file, Spring throws the IllegalArgumentException. One way to avoid this exception is to provide a default value for the property:

@Value("${recommender.implementation.favoriteMovie: Finding Dory}")  
String favoriteMovie;

Now, if the favoriteMovie property is commented out in the app.properties file using the # symbol, its default value will be read.

**@PropertySource**

1. In the MovieRecommenderSystemApplication method, we will mention the name of the file from where to fetch the value using the @PropertySource annotation. By default, Spring loads the property file from the classpath. Since our file is in **src/main/resources**, we will use classpath:app.properties. The getFavoriteMovie method will print the value read from the property file.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

@SpringBootApplication

 @PropertySource("classpath:app.properties")

public class MovieRecommenderSystemApplication {

    public static void main(String[] args) {

        ApplicationContext appContext = SpringApplication.run

          MovieRecommenderSystemApplication.class, args);

        RecommenderImplementation recommender = appContext.getBean

          (RecommenderImplementation.class);

        String favoriteMovie = recommender.getFavoriteMovie();

        System.out.println(favoriteMovie);

    }

}





If the code given below is executed, our application will run as usual after reading the value from the property file. This gives us the freedom to change the value of favoriteMovie in the app.properties file without having to make changes to the source code.

Reading value from external property file

Using an external property file, we can configure our application differently in development, QA, and testing.

MovieRecommenderSystemApplication.java

app.properties

RecommenderImplementation.java

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

package io.datajek.spring.basics.movierecommendersystem.lesson15;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.stereotype.Component;

@Component

public class RecommenderImplementation {

    //Filter is a dependency of RecommenderImplementation

    @Autowired

    private Filter filter;

    @Value("${recommender.implementation.favoriteMovie: hello}")

    private String favoriteMovie;

    public RecommenderImplementation(Filter filter) {

        this.filter = filter;

    }

    public Filter getFilter() {

        return filter;

    }

    public String returnFavoriteMovie() {

        return favoriteMovie;

    }

    //use a filter to find recommendations

    public String [] recommendMovies (String movie) {





Run

Save

Reset

# Spring Boot

**What is Spring Boot?**

This lesson discusses the need and features of Spring Boot with a focus on starter projects.

**We'll cover the following**

* [Features of Spring Boot](https://www.educative.io/courses/guide-spring-5-spring-boot-2/myo8OW205z9#Features-of-Spring-Boot)
* [Spring Boot autoconfiguration](https://www.educative.io/courses/guide-spring-5-spring-boot-2/myo8OW205z9#Spring-Boot-autoconfiguration)
* [Starter projects](https://www.educative.io/courses/guide-spring-5-spring-boot-2/myo8OW205z9#Starter-projects)

**Features of Spring Boot**

Spring boot enables robust creation of applications. It provides features like servers, metrics, health checks, etc. Spring Boot allows for integration with many different servers, but by itself, it is neither an application server nor a web server. Auto configuration is a great feature of Spring Boot whereby it provides all the required dependencies to start a particular project. For example, **starter web** automatically configures everything needed for a web application. Similarly, **starter JPA** automatically configures basic JPA along with Hibernate. The main features of Spring Boot are shown below:

Features of Spring Boot

**Spring Boot autoconfiguration**

Without Spring Boot, creating a simple web application can be a daunting and time-consuming task. When creating a project, the frameworks required need to be decided along with the versions of the frameworks that are compatible with each other. Spring Boot does that work for us. It has already created a set of dependencies with compatible versions. The developer only needs to choose the uber dependency (like starter web) and Spring Boot configures appropriate versions of Spring core, dispatcher servlet, view resolver, logging, security and validation frameworks along with exception handling and support for internationalization. Spring Boot takes care of the configuration part for us so we can focus on developing the business logic. Spring Boot saves the time spent on creating a Spring application from scratch. An analogy of Spring and Spring Boot is shown below:

Spring vs Spring Boot analogy

**Starter projects**

Spring Boot offers *starter projects* which have a set of predefined dependencies. These dependencies are automatically provided to the project to simplify the build configuration.

**Starter web** is the preferred starter for building web applications, including RESTful applications, using Spring MVC. It uses Tomcat as the default embedded container. The pom.xml file of spring-boot-starter-web shows a number of dependencies including spring-web, spring-webmvc, starter-tomcat, and starter-json for conversion to JSON when a web service is invoked.

POM file of starter-web

**Starter test** is the starter dependency for testing Spring Boot applications with libraries including JUnit Jupiter, Hamcrest, and Mockito. spring-boot-starter-test enables the developer to write unit and integration tests. The pom.xml file shows dependencies on JUnit, AssertJ, and Mockito. It also has Hamcrest that, in combination with AssertJ, is used for writing matchers.

POM file of starter-test

**Starter JPA** is another frequently-used starter project. JPA is the interface for the Hibernate framework. starter-data-jpa is used for Spring Data JPA with Hibernate. The pom.xml file shows that it has a dependency on spring-boot-strater-aop, spring-boot-starter-jdbc, hibernate-core, and transaction-api.

POM file of starter-data-jpa

# Creating a REST Service

See the ease and simplicity of development using Spring Boot by creating a REST controller.

**We'll cover the following**

* [Spring Initializr](https://www.educative.io/courses/guide-spring-5-spring-boot-2/mEqp5RYMDJ9#Spring-Initializr)
* [@RestController](https://www.educative.io/courses/guide-spring-5-spring-boot-2/mEqp5RYMDJ9#@RestController)
* [Mapping requests](https://www.educative.io/courses/guide-spring-5-spring-boot-2/mEqp5RYMDJ9#Mapping-requests)

In this lesson, we will create a REST service using Spring Boot. We will define a controller for movie recommendations to return a few hard coded values when the URI http://localhost:8080/movies is accessed. The HTTP Get request for the above URI will be mapped to a controller method which returns a JSON response.

The purpose of this lesson is to show how Spring Boot makes application development easy.

## Spring Initializr

1. There are numerous ways of creating a Spring Boot application. The easiest way is using the web UI called Spring Initializr. Other approaches include using the Spring Boot command line interface (CLI) and the Spring Tools Suite (STS) IDE.

The Spring Initializr offers a way to bootstrap a Spring Boot application by offering a set of options to choose from. These are called starter projects. Starter projects make it very easy to develop applications. Since we want to develop a simple REST service, we need the web starter. We will fill the form on [**start.spring.io**](http://start.spring.io/) with the following information:

Group Id: **io.datajek.springbootdemo**

Artifact Id: **recommender-api**

Click **Add Dependencies** and choose the **Spring Web** dependency from the web section. Click on **Generate** button and once the project is downloaded, unzip and import the project in your IDE.

Creating a project with web dependency

1. The **Spring Web** dependency that we added, results in a lot of jars being downloaded. These can be seen in the **Maven Dependencies** folder and include jackson-databind, tomcat, spring-webmvc, and json etc. If we were to create a REST application without Spring Boot, we would need to add the spring-webmvc, jackson-databind, servlet-api for configuring the dispatcher servlet, as well as other dependencies that Spring Boot automatically downloaded for us!

Spring Boot provides us with an embedded server. If we open the application and run it, the log says:

Tomcat started on port 8080

This means that we have a web application up and running. Tomcat runs on port 8080 by default. If we want to verify that the server is running, we can go to the web browser and type http://localhost:8080 or http://127.0.0.1:8080. We will get an error from the Tomcat servlet container saying there is no mapping for /error defined. This is because we have not provided any mapping for localhost. So, when the server tries to go to /error and finds no mapping, it throws an error. We will define a mapping for localhost in step 6.

## @RestController

1. To create a controller, we will create a class called RecommendationsController and use an annotation @RestController to inform Spring that this is a controller class. A controller is a Java class that maps a URI to a method. When the method is executed, it returns a response. A @RestController is a specialized form of @Controller annotation which handles REST requests and responses. It automatically handles conversion to JSON.

import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
public class RecommendationsController {  
    //method to return all movies  
}

1. Next, we will create a class Movie that has id, name, and rating attributes. We will add constructors, getter methods, and a toString method as follows:

public class Movie {  
    int id;  
    String name;  
    double rating;  
  
    public Movie() {  
    }  
   
    public Movie(int id, String name, double rating) {  
        super();  
        this.id = id;  
        this.name = name;  
        this.rating = rating;  
    }  
  
    public int getId() {  
        return id;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public double getRating() {  
        return rating;  
    }  
  
    @Override  
    public String toString() {  
        return "Movie [id=" + id + ", name=" + name + ", rating=" + rating + "]";  
    }  
}

Note, that we have added a no-arg constructor along with a constructor taking movie name and rating as arguments.

1. Now that we have a Movie class, we can create a method in the controller that returns a list of movies. This method is called getAllMovies and its return type is a List of Movie objects. We will create a list inline using the Arrays.asList method as follows:

public List<Movie> getAllMovies() {  
 return Arrays.asList(new Movie(1, "Ice Age", 7.5),     
                      new Movie(2, "Happy Feet", 6.4),   
                      new Movie(3, "Shark Tales", 6.0) );  
}

## Mapping requests

1. We have created a controller and a method. Now we need to inform Spring to execute the method when the HTTP request is received. There are different types of HTTP requests like GET, POST, PUT, etc. The @RequestMapping annotation maps the HTTP request and URI to a method in the controller. Since we are currently handling a GET request, we can also use the shortcut annotation @GetMapping on the getAllMovies method.

@RequestMapping(method=RequestMethod.GET, value="/movies")  
//OR  
@GetMapping("/movies")  
public List<Movie> getAllMovies() {  
    //...  
}

The @GetMapping annotation will map the /movies URI to the getAllMovies method and convert the list of movies returned by the method to a JSON response.

By following these simple steps, our REST controller is ready. If the application in the widget below is executed, a JSON response is returned by the server.

If you are running this application in your IDE, head over to the browser and type localhost:8080/movies or http://127.0.0.1:8080/movies in your browser. The server will respond back with a JSON response of the recommended movies.

###### /

RecommenderApiApplication.java

RecommendationsController.java

Movie.java

**RecommendationsController.java**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

package io.datajek.springbootdemo.recommenderapi;

import java.util.Arrays;

import java.util.List;

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

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

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

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

@RestController

public class RecommendationsController {

    @GetMapping("/movies")

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

    public List<Movie> getAllMovies() {





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/movies>

To be able to run the application again in an IDE, we need to kill the application to free up port 8080.

In this lesson, we focused on the logic of creating a controller while the infrastructure was taken care of by Spring Boot. All the things needed for the REST service to function were configured automatically. Here is a list of things that happened in the background:

* @SpringBootApplication annotation defines a configuration and enables a component scan. The dispatcher servlet gets initialized without us having to configure it.
* During component scan, all classes marked with different @Component annotations are picked up by Spring. The class marked with @RestController annotation is picked up and RecommendationsController is listed as a component.
* Tomcat server is autoconfigured in our application. There is no need to download and install the server. There are some servlet container configurations that may be needed on Tomcat to run our application. Spring Boot makes those servlet container configurations a part of our application which makes it a standalone application.
* In the RecommendationsController class, we are returning an array back which is automatically converted into a JSON response. The starter-json dependency causes the JSON binding.
* spring-boot-autoconfigure is listed as a dependency in the **Maven Dependencies** folder which has a lot of classes. At startup, Spring Boot triggers this JAR which looks through classes on the classpath. If we turn the debug mode on in the application.properties file, we will be able to see a detailed autoconfiguration report.

logging.level.org.springframework = DEBUG

This report mentions all the beans that have been configured due to conditional matching (positive matches). It also shows all the things that were not autoconfigured (negative matches) and why certain beans were not created.

Below, we have reproduced the autoconfiguration report from the output log created by executing the application on our local machine.

755

756

757

758

759

760

761

762

763

764

765

766

767

768

769

770

771

772

773

774

775

776

777

778

779

780

781

782

783

784

785

   WebSocketServletAutoConfiguration.JettyWebSocketConfiguration:

      Did not match:

         - @ConditionalOnClass did not find required class 'org.eclipse.jetty.websocket.jsr356.server.deploy.WebSocketServerContainerInitializer' (OnClassCondition)

   WebSocketServletAutoConfiguration.UndertowWebSocketConfiguration:

      Did not match:

         - @ConditionalOnClass did not find required class 'io.undertow.websockets.jsr.Bootstrap' (OnClassCondition)

   XADataSourceAutoConfiguration:

      Did not match:

         - @ConditionalOnClass did not find required class 'javax.transaction.TransactionManager' (OnClassCondition)

Exclusions:

-----------

    None

Unconditional classes:

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

    org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoConfiguration

    org.springframework.boot.autoconfigure.context.LifecycleAutoConfiguration

    org.springframework.boot.autoconfigure.context.PropertyPlaceholderAutoConfiguration

    org.springframework.boot.autoconfigure.availability.ApplicationAvailabilityAutoConfiguration

    org.springframework.boot.autoconfigure.info.ProjectInfoAutoConfiguration





**Developer Tools**

In this lesson, we will see how the Spring Boot Dev Tools module improves development time.

**We'll cover the following**

* [devtools dependency](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7DOnKyE9V6A#devtools-dependency)
* [Automatic restart](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7DOnKyE9V6A#Automatic-restart)
* [LiveReload server](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7DOnKyE9V6A#LiveReload-server)

We created a simple RecommendationsController in the last lesson that received a request from localhost/movies and returned a list of movies. If we make a change in the movies to be returned by the controller and execute the service again, the change will not be picked up unless the server is stopped and started again. Restarting the application takes a long time and doing that over and over again causes unwanted delays at the developer’s end.

**devtools dependency**

To counter this problem, Spring Boot offers a **Developer Tool** that supports live reloads. This jar is provided by the Spring Boot framework and is useful for development-specific debugging. To use developer tools, add the following dependency to pom.xml file:

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-devtools</artifactId>  
</dependency>

If the scope of this dependency is limited to runtime using the tag <scope>runtime</scope> it will not be packaged in the jar.

When the dependency is downloaded, we can execute the project again. From this point on, any change in the code will be picked up by the server without the need to stop and restart the server. We can make changes to the list of movies being returned and run the application again.

**Automatic restart**

One of the things that you will notice when running the application after making a change in the code is that the application automatically starts again. This is an efficient restart that takes way less time as compared to a typical restart. Spring knows that the Maven dependencies do not change, so only the application beans are loaded again, which is why it takes less time.

Automatic restart of the application is an important feature of **DevTools**. Whenever any change is made to the code, DevTools causes an automatic restart when a file on its classpath changes. Spring Boot provides two classloaders: one for the classes that do not change like third-party jars and the other for application code using the RestartClassLoader. When the code is changed, only the RestartClassLoader is loaded, which causes the restart to be much faster.

DevTools provides automatic restart

**LiveReload server**

To save the time wasted in refreshing the browser after every code change, Spring Boot comes with a LiveReload embedded server which triggers an automatic browser refresh. The LiveReload extensions are available for different browsers. Once installed and enabled, any change in the code is detected by the LiveReload server and the browser is automatically refreshed to reflect the change.

# Actuator

Learn about Spring Boot Actuator and the different monitoring metrics that it provides.

**We'll cover the following**

* [Actuator](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JYzBw0kRklo#Actuator)
* [Actuator dependency](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JYzBw0kRklo#Actuator-dependency)
* [HAL browser dependency](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JYzBw0kRklo#HAL-browser-dependency)
* [Actuator endpoints](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JYzBw0kRklo#Actuator-endpoints)

## Actuator

Spring Boot Actuator is a feature that provides monitoring features for the application during development and after deployment. It provides metadata about the application like the beans configured, how autoconfiguration has worked, how many times a specific service is called, how many times a specific service has failed, etc.

## Actuator dependency

1. The Actuator module can be enabled while creating a Spring Boot application by adding the **Actuator** dependency from the Ops section. For an application that is already running, spring-boot-starter-actuator can be added to the pom.xml file as follows:

<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-actuator</artifactId>  
</dependency>

## HAL browser dependency

1. We need to add one more dependency. The Actuator exposes a lot of REST services that are compliant with the HAL standard. We will use a HAL browser to view the data provided by the services. For this, add the following dependency:

<dependency>  
    <groupId>org.springframework.data</groupId>  
    <artifactId>spring-data-rest-hal-browser</artifactId>  
    <version>3.3.6.RELEASE</version>  
</dependency>

1. Since we have enabled the HAL browser, if we type localhost:8080, we will be taken to the HAL browser that we can use to browse through the Actuator by typing /actuator. The URL to launch the Actuator is http://localhost:8080/actuator (or http://localhost:8080/application for older Spring releases). This endpoint links to two other URLs for health and info. /info is used to display custom information about the application. It will be empty unless we write something for this URL.

localhost:8080/health is a JSON file that shows metrics related to the application’s health like the status, disk space (total as well as free), and embedded database information.

1. The Actuator calculates metrics related to application performance and makes them available as a new endpoint, e.g., /health without us having to write any controller for it. These metrics are deployed along with the application. The application URL will also have the /health URL. This can be changed to some other location by adding the following line in the application.properties file:

Management.server.port = 8081

If this change is made, then the application will be accessible at localhost:8080 and health information at port 8081.

## Actuator endpoints

1. By default, only /health and /info are available. The other endpoints are disabled. To enable the web exposure of all management endpoints, add the following line to application.properties:

1

management.endpoints.web.exposure.include=\*





Now when the application is run and Actuator is viewed, it shows a host of other endpoints. Some of them are listed below:

* /auditevents shows audit information like which users were validated, or how many users failed the authentication test, etc.
* /beans shows all the spring beans that are configured. It shows the name of the bean as well as scope and type, and any dependencies that the bean might have. It can be used to analyze the application.
* /condition shows all the positive and negative matches for autoconfiguration similar to the report that is generated at application startup.
* /httptrace shows the last 100 requests that were executed along with the response that was sent back.
* /mappings shows all the URI’s for @RequestMapping.
* /metrics shows a list of valid metrics. Further information about any metric can be obtained by copying the metric and adding it to the url. For example, the jvm.memory.used will show the amount of memory used.
* /shutdown when enabled, lets the application gracefully shutdown. The feature can be enabled by making the following changes to the application.properties file:

management.endpoint.shutdown.enabled=true

Since we have enabled the HAL browser, we can view it at http://localhost:8080/browser/index.html#

It should be noted that enabling a lot of tracking and monitoring has an impact on performance. For example, enabling httptrace for all requests will hamper performance. Also, careful consideration is required when enabling endpoints since they expose sensitive information about the application.

The code widget below shows the HAL browser for exploring Actuator endpoints. Type **actuator** in the text box under **Explorer** and press **Go!**. All exposed Actuator endpoints will appear in the links section which can be further explored. For example, to view all the beans, type /actuator/beans under **Explorer** and press **Go!**. The list of all beans appears in the properties section. In the same manner, /actuator/metrics shows all metrics. To view the details of a metric, say jvm.memory.used, add it to the end of the URL like so: /actuator/metrics/jvm.memory.used. This will show the value of memory used.

To view the Actuator endpoints as a JSON response, type **/actuator** after **educative.run** in the URL given below.

###### /

RecommenderApiApplication.java

RecommendationsController.java

Movie.java

application.properties

pom.xml

**application.properties**

1

management.endpoints.web.exposure.include=\*





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/actuator>

**Comparing Spring Boot and Spring MVC**

Let's compare Spring Boot and Spring MVC and see which one to use when developing a web application

**We'll cover the following**

* [Spring framework](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMk8XR4QNDL#Spring-framework)
* [Spring MVC](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMk8XR4QNDL#Spring-MVC)
* [Spring Boot](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMk8XR4QNDL#Spring-Boot)

Spring offers the Spring MVC framework for developing web applications. A very common question in the minds of new learners is the difference between Spring Boot and Spring MVC and which one to use for developing web applications. Both Spring Boot and Spring MVC play a separate role and both are designed to solve different problems as we will see in this lesson.

**Spring framework**

Spring Boot and Spring MVC are both components of the **Spring** framework. Spring offers dependency injection and inversion of control, which provides good integration with other frameworks. It defines beans using different annotations and autowires them. Spring helps build loosely coupled applications that are easy to unit test. It also eliminates plumbing code like exception handling, establishing and closing connections, etc.

**Spring MVC**

**Spring MVC** is an HTTP oriented Spring framework, which is concerned with developing web applications. It makes use of the Model View Controller (MVC) design pattern to achieve separation of concerns.

MVC Architecture

The DispatcherServlet is the front controller that handles all requests while the View Resolver is concerned with resolving view names to physical views. This decoupling makes development of web applications and RESTful services very simple because the model, view, and controller operate without dependency. When we built a web service with Spring Boot, it internally made use of Spring MVC. Spring MVC helps create web applications that are scalable as well as testable.

**Spring Boot**

Spring Boot is designed to solve configuration issues. It autoconfigures a lot of dependencies based on the kind of application that is being built. Spring Boot offers pre-configured projects to bootstrap an application in a few simple steps. When building a web application using Spring Boot, the DispatcherServlet, ViewResolver, Data Source, Transaction Manager, etc. get configured automatically. Spring Boot configures compatible versions of the dependencies needed for the frameworks. It also provides monitoring features.

Spring Boot helps kickstart the project by bringing in all the required dependencies. It is a useful tool for someone who is just starting out with Spring and gets overwhelmed with the configuration part. It also saves a lot of time. However, Spring Boot offers a biased view and has strong preferences about the dependencies that are used.

# Spring JDBC

# Creating a JDBC Starter Project

In this step, we will create a JDBC starter project, connect to the H2 database, create a table, and add data to the table.

**We'll cover the following**

* [Spring Boot JDBC](https://www.educative.io/courses/guide-spring-5-spring-boot-2/mEgWrw5ppZn#Spring-Boot-JDBC)
* [In-memory database](https://www.educative.io/courses/guide-spring-5-spring-boot-2/mEgWrw5ppZn#In-memory-database)
* [Using Spring Boot Initializer](https://www.educative.io/courses/guide-spring-5-spring-boot-2/mEgWrw5ppZn#Using-Spring-Boot-Initializer)

Spring JDBC makes talking to databases easy by eliminating the need for establishing a connection, handling expectations, and closing connections. Spring provides a template class called JdbcTemplate to interact with databases which offers a wide array of methods to perform storage and retrieval of data. The JdbcTemplate class hides all the boilerplate code for connection management and error handling whereby the developer can focus on the SQL query.

Using the Spring JdbcTemplate, a Java application can store and retrieve objects of a class to a table in the database.

JdbcTemplate class handles database operations

**Spring Boot JDBC**

Spring Boot simplifies development and makes JDBC programming easy. JDBC with Spring Boot offers the following advantages over JDBC with Spring:

* When using Spring Boot, only one dependency (spring-boot-starter-jdbc) is needed in the pom.xml file as compared to multiple dependencies in Spring (spring-context, spring-jdbc, etc.).
* Spring Boot automatically initializes the datasource bean whereas it needs to be created using XML or Java configuration in Spring.
* Spring Boot also autoconfigures the JdbcTemplate and other template beans that need to be explicitly registered in Spring.
* Lastly, Spring Boot automatically creates the database schema specified in the **schema.sql** file. The schema needs to be explicitly configured if Spring is used.

**In-memory database**

Keeping these benefits in mind, we will use Spring Boot JDBC with an in-memory database for understanding JDBC concepts. In-memory databases are faster because they are stored in memory and do not require disk access. The in-memory database lives as long as the application is running. When the application is stopped, the database is removed from memory. In-memory databases are easy to use as they do not require any setup like installing a server, creating a database, defining a schema, etc. that are required with databases like Oracle or MySQL. We can easily swap the in-memory database with a traditional database with a few changes in the **pom.xml** file. The in-memory database can be viewed in a web browser.

**Using Spring Boot Initializer**

We will create a database of tennis players and see how Spring makes it easy to interact with databases. We will use the in-memory H2 database in this example which offers a web client for viewing the database.

To set up a starter project for JDBC, go to [Spring Initializr](http://start.spring.io/). The Group Id of the project is **springdata** and the Artifact Id is **tennis-player**. We will add the following dependencies:

* JDBC API for querying the database
* H2 Database that supports JDBC API access
* Spring Web for the web console offered by H2

After generating the project and importing it in the IDE, we can see the three dependencies that we just added in the **pom.xml** file. When the application is run, Tomcat starts on port 8080 because of the web dependency.

# Setting up the H2 Database

Learn how to connect to the in-memory H2 database, create tables, and populate data.

**We'll cover the following**

* [Configuring database connection](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B6464558mO2#Configuring-database-connection)
* [Creating a table](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B6464558mO2#Creating-a-table)
* [Inserting data](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B6464558mO2#Inserting-data)

In this lesson, we will set up the in-memory H2 database.

## Configuring database connection

1. The in-memory database H2 has automatically been configured in our application. The URL can be found from the console log. This value is randomly generated each time the server is restarted. To make the database URL a constant, we need to configure this in **application.properties** as follows:

spring.datasource.url=jdbc:h2:mem:testdb

1. The next task is connecting to the H2 database. One of the reasons for using Spring Boot is that its autoconfiguration feature looks at the H2 dependency and automatically configures a connection to the H2 database. The H2 console can be enabled in the **application.properties** file as follows:

spring.h2.console.enabled=true

1. The database can be viewed in the web browser by typing localhost:8080/h2-console or http://127.0.0.1:8080/h2-console. In the login page that shows up, make sure that the JDBC URL is the same as the one that we provided in the **applications.properties** file (jdbc:h2:mem:testdb). If not, change it to jdbc:h2:mem:testdb and click connect to go to the database console.

This will open up the interface of the database.

## Creating a table

1. Now that we have connected to our database, it is time to create a table. We can do that in the H2 web console by using a CREATE TABLE query, but since this is an in-memory database, all changes will be lost when the application is terminated. A better way is to define the query in a SQL file.

For this purpose, create an SQL file in **src/main/resources** called **schema.sql**. This file will be called whenever the application is launched and will initialize the database. It contains all the DDL (Data Definition Language) queries. We will create a table called Player in this file:

  CREATE TABLE Player (  
     ID INTEGER NOT NULL,  
     Name VARCHAR(255) NOT NULL,  
     Nationality VARCHAR(255) NOT NULL,  
     Birth\_date TIMESTAMP,  
     Titles INTEGER,  
     PRIMARY KEY (ID)  
  );

After saving this file, if we connect to the H2 console again, we will see the Player table with five columns. To view this table, we can write a simple query:

SELECT \* FROM Player;

The table is empty at the moment.

## Inserting data

1. We can insert data in the table from the H2 console. However, it will be lost when the application stops. To avoid inserting data every time we restart the application, we will create another SQL file in the **src/main/resources** folder and add all our INSERT queries to that file. This file is called **data.sql** and it contains all DML (Data Manipulation Language) queries.

INSERT INTO Player (ID, Name,   
Nationality, Birth\_date, Titles)  
VALUES(1, 'Djokovic', 'Serbia', '1987-05-22', 81);  
  
INSERT INTO Player (ID, Name,   
Nationality, Birth\_date, Titles)  
VALUES(2, 'Monfils', 'France', '1986-09-01', 10);  
  
INSERT INTO Player (ID, Name,   
Nationality, Birth\_date, Titles)  
VALUES(3, 'Isner', 'USA', '1985-04-26', 15);

Now if we run the application, the data is read from the **data.sql** file and the Player table is populated. We can run the SELECT \* query to confirm that our table has 3 rows.

In the login page that shows up, make sure that the JDBC URL is the same as the one that we specified in step 1 (jdbc:h2:mem:testdb).

application.properties

data.sql

schema.sql

TennisPlayerApplication.java

**application.properties**

1

2

3

spring.datasource.url=jdbc:h2:mem:testdb

spring.h2.console.enabled=true

spring.h2.console.settings.web-allow-others=true





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/h2-console/>

# SELECT Query

In this lesson, we will write two queries, one to return all rows and the other to return rows based on some criterion.

**We'll cover the following**

* [Defining Player bean](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7AM2M7y6vry#Defining-Player-bean)
* [Creating DAO class](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7AM2M7y6vry#Creating-DAO-class)
* [SELECT \* query](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7AM2M7y6vry#SELECT-*-query)
* [Executing the query](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7AM2M7y6vry#Executing-the-query)
* [Select with a WHERE clause](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7AM2M7y6vry#Select-with-a-WHERE-clause)

JDBC involves a lot of boilerplate code that is required just to get the application working. It is a tedious task to write a simple query using JDBC. There are a number of steps that are required to interact with the database.

* The first step is establishing a connection
* The second step is creating a prepared statement or query
* The third step is to execute the query
* The fourth step is looping through the result set to get the objects
* The fifth and final step is to close the connection

Spring data support makes interacting with databases easy. Since we are using Spring Boot, the connection part is automatically taken care of and the data source is automatically configured. Likewise, the connection is automatically closed. Spring provides JdbcTemplate, which makes it easy to write and execute a query. It also provides the BeanPropertyRowMapper which maps rows of a table to a bean.

Spring JdbcTemplate and RowMapper simplify database operations

In this lesson, first we will write a query to return all rows from the Player table. We will learn how to map the data coming from the database to a bean in our application. Next, we will modify our query to return the row that matches an input argument.

## Defining Player bean

1. We have created a Player table but we need to define a class Player with the same fields. The rows in the Player table will map to this class.

public class Player{  
    private int id;  
    private String name;  
    private String nationality;  
    private Date birthDate;  
    private int titles;  
    //. . .  
}

Next, we will create two constructors (with arguments and no-argument), getters and setters, as well as a ToString method for the fields. The no-arg constructor is a requirement of the BeanPropertyRowMapper.

The Player class is a bean and the data coming from the Player table in H2 will be mapped to this class.

## Creating DAO class

1. We will create a new class PlayerDao to interact with the database. Since this class belongs to the data layer, we will use the @Repository annotation instead of the generic @Component annotation.

@Repository  
public class PlayerDao {  
  
}

The PlayerDao class will have methods that execute various queries to manipulate rows of the Player table.

## SELECT \* query

1. The first method in the PlayerDao class returns all rows from the Player table. We will call this method getAllPlayers. The return type will be List&ltPlayer. This method will execute the SELECT \* query. We will autowire the JdbcTemplate in the PlayerDao class. JdbcTemplate offers a number of methods.

@Autowired  
JdbcTemplate jdbcTemplate;

We will make use of the query method of JdbcTemplate to execute a SELECT \* query.

A row mapper is used to match the data coming from the database to the attributes of the bean. The BeanPropertyRowMapper is the default row mapper defined by Spring.

public List<Player> getAllPlayers() {  
    String sql = "SELECT \* FROM PLAYER";  
    return jdbcTemplate.query(sql, new BeanPropertyRowMapper<Player> (Player.class));  
}

## Executing the query

1. To run this query, we will use the CommandLineRunner. A CommandLineRunner is an interface in Spring Boot which has only one method called run. This method is launched as soon as the context is loaded. Our TennisPlayerApplication will implement the CommandLineRunner. We will autowire the PlayerDao class to use an object of this class to call the getAllPlayers method inside the run method of the CommandLineRunner. A logger will display the list of players returned.

@SpringBootApplication  
public class TennisPlayerApplication implements CommandLineRunner {  
  
    private Logger logger = LoggerFactory.getLogger(this.getClass());  
  
    @Autowired  
    PlayerDao dao;  
   
    public static void main(String[] args) {  
        SpringApplication.run(TennisPlayerApplication.class, args);  
    }  
  
    @Override  
    public void run(String... args) throws Exception {  
        logger.info("All Players Data: {}", dao.getAllPlayers());  
    }  
}

Run the code given below and look for “All Players Data:” towards the end of the terminal output. It can be seen that three rows have been retrieved from the database and are displayed in the log.

The database can be viewed in the web browser by typing localhost:8080/h2-console or http://127.0.0.1:8080/h2-console. In the login page that shows up, make sure that the JDBC URL is the same as the one that we provided in the applications.properties file (jdbc:h2:mem:testdb). If not, change it to jdbc:h2:mem:testdb and click connect to go to the database console. This will open up the interface of the database.

schema.sql

data.sql

PlayerDao.java

Player.java

TennisPlayerApplication.java

**TennisPlayerApplication.java**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

package io.datajek.springdatajdbc;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class TennisPlayerApplication implements CommandLineRunner {

    private Logger logger = LoggerFactory.getLogger(this.getClass());

    @Autowired

    PlayerDao dao;

    public static void main(String[] args) {

        SpringApplication.run(TennisPlayerApplication.class, args);

    }

    @Override

    public void run(String... args) throws Exception {

        logger.info("All Players Data: {}", dao.getAllPlayers());

    }

}





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/h2-console/>

It is very easy to write a simple method with Spring JDBC using the **JdbcTemplate**. JdbcTemplate takes out the common functions required before and after the execution of a query so the developer only focuses on the query and any custom result set retrieval functionality. JdbcTemplate creates the connection and automatically closes it if an exception is thrown.

We have an embedded database in our classpath. Spring Boot autoconfiguration sees the database and autoconfigures a datasource. This leads to a JdbcTemplate getting autoconfigured.

## Select with a WHERE clause

Now suppose we want to write a query that returns the information of a player based on his ID. This is a SELECT \* query with a WHERE clause. This query returns only one row from the table. The method is called getPlayerById, and it takes an integer value as input and returns a Player object.

1. If we search by the primary key, we will get one row back. In this case, instead of using the query method, we will use the queryForObject method of JdbcTemplate. This method accepts a list of parameters. We will create a list of objects and pass it to the method. The parameter id will be substituted in the query and a Player object is returned.

public Player getPlayerById(int id) {  
    String sql = "SELECT \* FROM PLAYER WHERE ID = ?";  
    return jdbcTemplate.queryForObject(sql,   
                                       new BeanPropertyRowMapper<Player>(Player.class),   
                                       new Object[] {id});  
}

1. To execute this query, we will call getPlayerById in the run method and pass id 3 to it:

@Override  
public void run(String... args) throws Exception {  
    //logger.info("All Players Data: {}", dao.getAllPlayers());  
    logger.info("Player with Id 3: {}", dao.getPlayerById(3));  
}

The code below shows the output when getPlayerById is called with an ID of 3. The retrieved row can be seen towards the end of the terminal output.

schema.sql

data.sql

PlayerDao.java

Player.java

TennisPlayerApplication.java

**TennisPlayerApplication.java**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

package io.datajek.springdatajdbc;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class TennisPlayerApplication implements CommandLineRunner {

    private Logger logger = LoggerFactory.getLogger(this.getClass());

    @Autowired

    PlayerDao dao;

    public static void main(String[] args) {

        SpringApplication.run(TennisPlayerApplication.class, args);

    }

    @Override

    public void run(String... args) throws Exception {

        //logger.info("All Players Data: {}", dao.getAllPlayers());

        logger.info("Player with Id 3: {}", dao.getPlayerById(3));

    }

}





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/h2-console/>

# INSERT Query

This lesson demonstrates how to insert a record in the database using the JdbcTemplate class.

**We'll cover the following**

* [JdbcTemplate Update method](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMKW26GVAwK#JdbcTemplate-Update-method)

In this lesson, we are going to learn how to write to the database using the methods of the JdbcTemplate class.

## JdbcTemplate Update method

1. To insert a row, we need to send a Player object as input parameter.

The update method is used for an INSERT query. This method takes the SQL query as well as an array of objects containing the values that will be inserted. The method returns an int value which shows the number of rows affected by the query.

public int insertPlayer(Player player)  
{  
    String sql = "INSERT INTO PLAYER (ID, Name, Nationality,Birth\_date, Titles) " +   
                                                                  "VALUES (?, ?, ?, ?, ?)";  
    return jdbcTemplate.update( sql, new Object[]   
                               { player.getId(), player.getName(), player.getNationality(),   
                                 new Timestamp(player.getBirthDate().getTime()),   
                                 player.getTitles()    
                               });  
}

1. We will execute the INSERT query in the run method. We will use the Player constructor with arguments to initialize a Player object and pass it to the method. We are passing the current date for the new record:

@Override  
public void run(String... args) throws Exception   
{  
   logger.info("Inserting Player 4: {}", dao.insertPlayer(  
                                          new Player (4, "Thiem", "Austria",   
                                                       new Date(System.currentTimeMillis()),  
                                                       17 )));    
   logger.info("All Players Data: {}", dao.getAllPlayers());  
}

As can be seen from the output, the insertPlayer method returns 1, which means that 1 row has been affected by the query. To verify the insert, we are calling the getAllPlayers method again. This time 4 rows are returned. We can also execute the **SELECT \*** statement in the H2 console.

The database can be viewed in the web browser by typing localhost:8080/h2-console or http://127.0.0.1:8080/h2-console. In the login page that shows up, make sure that the JDBC URL is the same as the one that we provided in the **applications.properties** file (jdbc:h2:mem:testdb). If not, change it to jdbc:h2:mem:testdb and click connect to go to the database console. This will open up the interface of the database.

###### /

schema.sql

data.sql

PlayerDao.java

Player.java

TennisPlayerApplication.java

**TennisPlayerApplication.java**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

package io.datajek.springdatajdbc;

import java.sql.Date;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class TennisPlayerApplication implements CommandLineRunner {

    private Logger logger = LoggerFactory.getLogger(this.getClass());

    @Autowired

    PlayerDao dao;

    public static void main(String[] args) {

        SpringApplication.run(TennisPlayerApplication.class, args);

    }

    @Override

    public void run(String... args) throws Exception {

/\*      logger.info("All Players Data: {}", dao.getAllPlayers());

        logger.info("Player with Id 3: {}", dao.getPlayerById(3));

\*/

        logger.info("Inserting Player 4: {}", dao.insertPlayer(new Player(4, "Thiem", "Austria", new Date(System.currentTimeMillis()), 17)));

        logger.info("All Players Data: {}", dao.getAllPlayers());





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/h2-console/>

# UPDATE Query

Learn how to update an existing record in the database.

**We'll cover the following**

* [JdbcTemplate Update method](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qAm75X98MEk#JdbcTemplate-Update-method)

## JdbcTemplate Update method

The update method of the JdbcTemplate is used to execute INSERT as well as UPDATE queries. So, to update a record in the table, we will create a method updatePlayer, which will be similar to the insertPlayer method. The only difference is in the query, which will change from INSERT to an UPDATE query.

1. The updatePlayer method will take a Player object and update the row corresponding to the Id value of the object. As mentioned in the previous lesson, the update method of the JdbcTemplate returns the number of rows affected by the query.

public int updatePlayer(Player player){  
    String sql = "UPDATE PLAYER " +  
                 "SET Name = ?, Nationality = ?, Birth\_date = ? , Titles = ? " +  
                 "WHERE ID = ?";  
  
    return jdbcTemplate.update( sql, new Object[] {   
                                   player.getName(),   
                                   player.getNationality(),   
                                   new Timestamp(player.getBirthDate().getTime()),   
                                   player.getTitles(),   
                                   player.getId() }  
                              );  
}

Notice that we have changed the order of getter methods to match the order in which the values will be passed to the query.

1. To execute the UPDATE query in the run method, we will initialize a Player object with the values that we want to update and pass it to the updatePlayer method. The code given below changes the birthDate attribute of the player.

@Override  
public void run(String... args) throws Exception {  
    //Inserting a player  
    logger.info("Inserting Player 4: {}", dao.insertPlayer(   
                        new Player(4, "Thiem", "Austria",   
                                   new Date(System.currentTimeMillis()), 17)));   
   
    //Updating a player  
    logger.info("Updating Player with Id 4: {}",  dao.updatePlayer(  
                        new Player(4, "Thiem", "Austria",   
                                   Date.valueOf("1993-09-03"), 17)));  
   
    //View player by Id  
    logger.info("Players with Id 4: {}", dao.getPlayerById(4));  
}

When the application is run, updatePlayer method returns 1, which means that the UPDATE query affected one row in the table. We can display the details of the player with Id 4 using the getPlayerById to verify that the birthDate has been updated.

The database can be viewed in the web browser by typing localhost:8080/h2-console or http://127.0.0.1:8080/h2-console. In the login page that shows up, make sure that the JDBC URL is the same as the one that we provided in the **applications.properties** file (jdbc:h2:mem:testdb). If not, then change it to jdbc:h2:mem:testdb and click connect to go to the database console. This will open up the interface of the database.

###### /

schema.sql

data.sql

PlayerDao.java

Player.java

TennisPlayerApplication.java

**PlayerDao.java**

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

import org.springframework.stereotype.Repository;

@Repository

public class PlayerDao {

    @Autowired

    JdbcTemplate jdbcTemplate;

    public int updatePlayer(Player player){

        String sql = "UPDATE PLAYER SET Name = ?, Nationality = ?, Birth\_date = ? , Titles = ? WHERE ID = ?";

        return jdbcTemplate.update(sql, new Object[] {player.getName(), player.getNationality(),

                            new Timestamp(player.getBirthDate().getTime()), player.getTitles(), player.getId()});

    }

    public int insertPlayer(Player player){

        String sql = "INSERT INTO PLAYER (ID, Name, Nationality, Birth\_date, Titles) VALUES (?, ?, ?, ?, ?)";

        return jdbcTemplate.update(sql,

               new Object[] {player.getId(), player.getName(), player.getNationality(),

                             new Timestamp(player.getBirthDate().getTime()), player.getTitles()});

    }

    public Player getPlayerById(int id) {

        String sql = "SELECT \* FROM PLAYER WHERE ID = ?";

        return jdbcTemplate.queryForObject(sql, new BeanPropertyRowMapper<Player>(Player.class), new Object[] {id});

    }

    public List<Player> getAllPlayers() {

        String sql = "SELECT \* FROM PLAYER";

        return jdbcTemplate.query(sql, new BeanPropertyRowMapper<Player>(Player.class));

    }

}





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/h2-console/>

# DELETE Query

Learn how to use the update method to delete a record from the database.

**We'll cover the following**

* [JdbcTemplate Update method](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVLOxy40xKr#JdbcTemplate-Update-method)

## JdbcTemplate Update method

In this step, we will write a query to delete a row based on the ID value. We will use the update method of the JdbcTemplate class.

1. The delete method is called deletePlayerById. This method returns an integer value of the number of rows that were affected by the query.

public int deletePlayerById(int id) {  
    String sql="DELETE FROM PLAYER WHERE ID = ?";  
    return jdbcTemplate.update(sql, new Object[] {id});  
}

1. We will execute the DELETE query in the run method by calling the deletePlayerById method:

@Override  
public void run(String... args) throws Exception {  
    logger.info("Deleting Player with Id 2: {}", dao.deletePlayerById(2));  
    logger.info("All Players Data: {}", dao.getAllPlayers());  
}

The log shows that deleteById method returns 1. We can confirm the deletion by checking the H2 console and executing the SELECT \* FROM Player query or by calling the getAllPlayers method of PlayerDao class.

Apart from inserting, updating, and deleting records, the update method of JdbcTemplate can also be used to run stored procedures. We can define a stored procedure in the same manner as we have defined a query and then pass it as an argument to the update method.

The database can be viewed in the web browser by typing localhost:8080/h2-console or http://127.0.0.1:8080/h2-console. In the login page that shows up, make sure that the JDBC URL is the same as the one that we provided in the **applications.properties** file (jdbc:h2:mem:testdb). If not, change it to jdbc:h2:mem:testdb and click connect to go to the database console. This will open up the interface of the database.

###### /

schema.sql

data.sql

PlayerDao.java

Player.java

TennisPlayerApplication.java

**TennisPlayerApplication.java**

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

import java.sql.Date;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class TennisPlayerApplication implements CommandLineRunner {

    private Logger logger = LoggerFactory.getLogger(this.getClass());

    @Autowired

    PlayerDao dao;

    public static void main(String[] args) {

        SpringApplication.run(TennisPlayerApplication.class, args);

    }

    @Override

    public void run(String... args) throws Exception {

        logger.info("All Players Data: {}", dao.getAllPlayers());

        logger.info("Deleting Player with Id 2: {}", dao.deletePlayerById(2));

        logger.info("All Players Data: {}", dao.getAllPlayers());

    }

}





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/h2-console/>

It can be seen how simple it is to retrieve, insert, update and delete rows using Spring JdbcTemplate. All the complexity lies in creating the query. Once the query has been created, all that remains is passing the appropriate parameters and a RowMapper.

# DDL Queries

This lesson touches upon how to execute a DDL query using the JdbcTemplate class.

**We'll cover the following**

* [Create table query](https://www.educative.io/courses/guide-spring-5-spring-boot-2/YMnkQ6LD062#Create-table-query)

Data Manipulation Language (DML) queries are the one where we manipulate the data in the table. In a rare scenario, we might want to create a table which is a Data Definition Language (DDL). In that case, we can use the execute method of the JdbcTemplate.

## Create table query

1. Let’s say we want to create a Tournament table to store details of a tennis tournament. We will create a method called createTournamentTable with a CREATE TABLE query.

public void createTournamentTable() {  
    String sql = "CREATE TABLE TOURNAMENT (ID INTEGER, NAME VARCHAR(50),   
                                           LOCATION VARCHAR(50), PRIMARY KEY (ID))";  
    jdbcTemplate.execute(sql);  
    System.out.println("Table created");  
}

1. We can call the createTournamentTable method in the run method and check if a table has been created from the output log.

@Override  
public void run(String... args) throws Exception {  
        dao.createTournamentTable();  
}

We can verify if the table has been created from the H2 console which shows an empty Tournament table.

The database can be viewed in the web browser by typing localhost:8080/h2-console or http://127.0.0.1:8080/h2-console. In the login page that shows up, make sure that the JDBC URL is the same as the one that we provided in the **applications.properties** file (jdbc:h2:mem:testdb). If not, change it to jdbc:h2:mem:testdb and click connect to go to the database console. This will open up the interface of the database.

###### /

schema.sql

data.sql

PlayerDao.java

Player.java

TennisPlayerApplication.java

**TennisPlayerApplication.java**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

package io.datajek.springdatajdbc;

import java.sql.Date;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class TennisPlayerApplication implements CommandLineRunner {

    private Logger logger = LoggerFactory.getLogger(this.getClass());

    @Autowired

    PlayerDao dao;

    public static void main(String[] args) {

        SpringApplication.run(TennisPlayerApplication.class, args);

    }

    @Override

    public void run(String... args) throws Exception {

        dao.createTournamentTable();

    }

}





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/h2-console/>

# Custom RowMapper

Learn how to use the RowMapper interface to define your own row mapper.

**We'll cover the following**

* [RowMapper interface](https://www.educative.io/courses/guide-spring-5-spring-boot-2/gxyXq6z9626#RowMapper-interface)
* [mapRow method](https://www.educative.io/courses/guide-spring-5-spring-boot-2/gxyXq6z9626#mapRow-method)
* [Using custom row mapper](https://www.educative.io/courses/guide-spring-5-spring-boot-2/gxyXq6z9626#Using-custom-row-mapper)

The BeanPropertyRowMapper can be used to map the results of the query to our bean. If the database table has a different structure or column names, we need to define our custom mapping.

## RowMapper interface

1. We can define our own row mapper by implementing the RowMapper interface and providing the bean onto which the rows will be mapped. The custom row mapper, PlayerMapper is created as an inner class because it will only be used inside JdbcPlayerDao. It is best practice to make it static and final.

@Repository  
public class PlayerDao {  
    //...  
    private static final class PlayerMapper implements RowMapper<Player> {  
  
    }  
  
}

The PlayerMapper class is reusable and can be used in all methods of the PlayerDao class to map rows from the Player table to the Player bean.

## mapRow method

1. The Rowmapper interface has one method, mapRow, for which we will write our custom implementation to initialize a Player object. This method defines how a row is mapped. It takes two arguments, the first being the result set which JdbcTemplate gets after running the query and the second being the row number. The row number of every row in the result set is different. The JdbcTemplate calls the mapRow method for every row in the result set and passes its row number as an argument. The method returns an object of Player type.

@Override  
public Player mapRow(ResultSet resultSet, int rowNum) throws SQLException {  
    Player player = new Player();  
    player.setId(resultSet.getInt("id"));  
    player.setName(resultSet.getString("name"));  
    player.setNationality(resultSet.getString("nationality"));  
    player.setBirthDate(resultSet.getTime("birth\_date"));  
    player.setTitles(resultSet.getInt("titles"));  
    return player;   
}

## Using custom row mapper

1. To use PlayerMapper, we can simply pass it in any method instead of the BeanPropertyRowMapper. We will create a method that finds players based on their nationality and use our custom mapper to convert the result set to objects as follows:

public List<Player> getPlayerByNationality(String nationality) {  
    String sql = "SELECT \* FROM PLAYER WHERE NATIONALITY = ?";  
    return jdbcTemplate.query(sql, new PlayerMapper(), new Object[] {nationality});  
}

The PlayerMapper is called for every row in the result set and each time it returns a new Player object. We do not have to worry about keeping track of the rows. JdbcTemplate does that work for us.

1. We will call the getPlayerByNationality to test our row mapper and find all French players.

@Override  
public void run(String... args) throws Exception {  
    logger.info("French Players: {}", dao.getPlayerByNationality("France"));  
}

In this example, it doesn’t make a difference whether we use our custom row mapper or the BeanPropertyRowMapper. Custom row mappers come in handy when the table definitions are different from the bean definitions.

The custom mapper can replace BeanPropertyRowMapper in all other methods that we wrote in the PlayerDao class.

###### /

schema.sql

data.sql

PlayerDao.java

Player.java

TennisPlayerApplication.java

**PlayerDao.java**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

package io.datajek.springdatajdbc;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Timestamp;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.jdbc.core.BeanPropertyRowMapper;

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.jdbc.core.RowMapper;

import org.springframework.stereotype.Repository;

@Repository

public class PlayerDao {

    @Autowired

    JdbcTemplate jdbcTemplate;

    public List<Player> getAllPlayers() {

        String sql = "SELECT \* FROM PLAYER";

        return jdbcTemplate.query(sql, new BeanPropertyRowMapper<Player>(Player.class));

    }

    public Player getPlayerById(int id) {

        String sql = "SELECT \* FROM PLAYER WHERE ID = ?";

        return jdbcTemplate.queryForObject(sql, new BeanPropertyRowMapper<Player>(Player.class), new Object[] {id});

    }

    public int insertPlayer(Player player){

        String sql = "INSERT INTO PLAYER (ID, Name, Nationality, Birth\_date, Titles) VALUES (?, ?, ?, ?, ?)";

        return jdbcTemplate.update(sql,

               new Object[] {player.getId(), player.getName(), player.getNationality(),





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/h2-console/>

# Spring Data JPA

**What is JPA?**

Let's explore how Spring JPA makes database operations easy in this introductory lesson.

**We'll cover the following**

*Note: JPA helps in reducing the boilerplate code needed for querying and mapping the result object to the java bean.*

* [How JPA works](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7nw4D5DvJpA#How-JPA-works)
* [JPA implementations](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7nw4D5DvJpA#JPA-implementations)

JDBC requires the developer to write the queries, map values to the query, pass a set of parameters to execute the query, and map rows of the result set to a bean. For simple queries, this task is manageable, but in large applications with hundreds of tables, the queries become complex. Writing and maintaining those queries requires expertise beyond the skillset of a Java developer. Java Persistence API (JPA) is designed to ease that task.

**How JPA works**

JPA challenges the notion of writing queries and mapping the data back. It creates entities that are Java objects which map to a row in a database table. JPA creates a schema based on the entities and defines relationships between entities. The Object-Relational Mapping (ORM) layer maps the objects to a database table.

ORM maps Java objects to table rows

Using JPA, we can map a Java class or bean to a table. The members of the class map columns in the table. When this mapping is defined, JPA can write queries on its own. It takes the responsibility of creating and executing queries for CRUD operations. This is due to the fact that the operations performed on a database are identical and can be generalized. The types of objects change based on the database schema but the operations remain the same.

Class members are mapped to columns in a database table

**JPA implementations**

Hibernate, the most popular ORM framework in the last decade, prompted the creation of the JPA standard. JPA is a standard of Object Relational Mapping. It is an interface that defines a set of annotations for creating the object relational mapping. There are numerous implementations of the JPA interface like Hibernate, EclipseLink, Apache OpenJPA, etc. Hibernate is by far the most popular implementation of JPA. It is a lightweight framework and can easily be integrated with Spring.

The benefit of using JPA instead of Hibernate is that JPA is a standard and one can switch to any other implementation later.

# Defining an Entity

In this lesson, we will create an entity and look at different JPA annotations for defining a relational mapping.

**We'll cover the following**

* [JPA dependency](https://www.educative.io/courses/guide-spring-5-spring-boot-2/m7gvJvD3Y2n#JPA-dependency)
* [@Entity](https://www.educative.io/courses/guide-spring-5-spring-boot-2/m7gvJvD3Y2n#@Entity)
* [@Table](https://www.educative.io/courses/guide-spring-5-spring-boot-2/m7gvJvD3Y2n#@Table)
* [@Id and @GeneratedValue](https://www.educative.io/courses/guide-spring-5-spring-boot-2/m7gvJvD3Y2n#@Id-and-@GeneratedValue)
* [@Column](https://www.educative.io/courses/guide-spring-5-spring-boot-2/m7gvJvD3Y2n#@Column)
* [JPA schema creation](https://www.educative.io/courses/guide-spring-5-spring-boot-2/m7gvJvD3Y2n#JPA-schema-creation)

## JPA dependency

To use Spring Data JPA, we will add the starter JPA dependency to the **pom.xml** file as follows:

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-jpa</artifactId>  
</dependency>

When the **pom.xml** file is saved, we can see the JPA API in the **Maven Dependencies** folder. This API defines a lot of different annotations like @Entity, @Column, @Table, etc. Hibernate is an implementation of the JPA API which automatically gets configured in our application. The hibernate-core jar can be seen in the Maven Dependencies folder.

We will take the tennis player database example to understand Spring Data JPA. We will make a copy of the Player and TennisPlayerApplication classes and save them to a new package, io.datajek.springdatajpa. As for the PlayerDao class, we will implement the functionality of all methods using the JPA API.

## @Entity

1. In our example, we have a Player class that lists attributes of a tennis player like his name, nationality, date of birth, and a number of titles won. We need to tell JPA that the objects of this class need to be mapped to a table in a database. JPA will create a table with the same name as the class and create columns for all the members of the class. Every instance of the Player class will become a row in the Player table. We will use the @Entity annotation to map this class to the Player table.

@Entity  
@Table(name="Player")  
Public class Player {  
  
}

## @Table

In case we want to map this class to a table with a different name, we can use the @Table annotation and provide the name of the table to which the bean maps to, as shown in the code above. Since the name of the entity and table match, we do not need the @Table annotation.

## @Id and @GeneratedValue

1. Every table in a relational database has a primary key. In our case, the Id attribute uniquely identifies each object. The @Id annotation is used to indicate the primary key. We can also let JPA generate the primary key value when we insert rows. The @GeneratedValue annotation will automatically generate Id values.

@Entity  
public class Player {  
    @Id  
    @GeneratedValue  
    private int id;  
    private String name;  
    private String nationality;  
    private Date birthDate;  
    private int titles;  
    // ...  
}

Now whenever a new row is inserted, we do not need to supply the Id value. We need another constructor that initializes all attributes except Id.

//constructor without Id attribute  
public Player(String name, String nationality, Date birthDate, int titles) {  
    super();  
    this.name = name;  
    this.nationality = nationality;  
    this.birthDate = birthDate;  
    this.titles = titles;  
}

The Player class now has three constructors; a no-arg constructor, one that initializes all five attributes, and one that initializes all attributes except the primary key.

## @Column

1. Another annotation provided by the JPA API is @Column annotation, which is used to define column mappings. @Column annotation mentions the name of the column that matches an attribute of the class. For example:

@Entity  
public class Player {  
    @Id  
    @GeneratedValue  
    private int id;  
    private String name;  
   
    @Column(name="nationality")  
    private String nationality;  
   
    private Date birthDate;  
    private int titles;  
    //...  
}

In our example, the column mapping annotation is not needed since the column names match.

## JPA schema creation

1. The Spring Boot autoconfiguration triggers a schema update and creates a table by the same name as the class marked with the @Entity annotation. When using JPA, we do not need to create a table. We will comment out the table creation query in schema.sql as it is not needed anymore.

After running the application, it can be confirmed from the H2 console — http://localhost:8080/h2-console (using JDBC URL: jdbc:h2:mem:testdb) — that a Player table has been created.

The output log shows the table creation query:

Hibernate: drop table if exists player CASCADE Hibernate: drop sequence if exists hibernate\_sequence Hibernate: create sequence hibernate\_sequence start with 1 increment by 1 Hibernate: create table player (id integer not null, birth\_date date, name varchar(255), nationality varchar(255), titles integer not null, primary key (id))

###### /

applications.properties

schema.sql

Player.java

TennisPlayerApplication.java

**TennisPlayerApplication.java**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

package io.datajek.springdatajpa;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class TennisPlayerApplication {

    public static void main(String[] args) {

        SpringApplication.run(TennisPlayerApplication.class, args);

    }

}





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/h2-console/>

# Creating a Repository

Let's create a repository — a class that manages entities — in this lesson.

**We'll cover the following**

* [@Repository annotation](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JQvr8M2WG6v#@Repository-annotation)
* [Enabling transaction management](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JQvr8M2WG6v#Enabling-transaction-management)
* [EntityManager and @PersistenceContext annotation](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JQvr8M2WG6v#EntityManager-and-@PersistenceContext-annotation)

In JPA terms, a repository is a class that manages the entities.

## @Repository annotation

1. We will create a PlayerRepository class to manage the Player entity and to store and retrieve data. We can mark this class with the @Repository annotation.

@Repository  
public class PlayerRepository {  
  
}

## Enabling transaction management

1. Database queries contain multiple steps. We will also enable transaction management to allow all steps in a query to succeed. In case of an error or runtime exception, all steps will be rolled back. Transactions are implemented at the business layer, but in this example, we will implement them at the repository level. Spring provides all the boilerplate code to start, commit, and roll back a transaction, which can also be integrated with JPA’s transaction management. This is enabled using the @Transactional annotation on a method or a class.

@Repository  
@Transactional  
public class PlayerRepository{  
  
}

Using this annotation on the PlayerRepository class, all the methods will be executed in a transactional context. So if we have INSERT and UPDATE in a single method, something called an EntityManager will keep track of both of them. If one of them fails, the whole operation will be rolled back.

## EntityManager and @PersistenceContext annotation

1. A JPA EntityManager manages connection to a database as well as to database operations. EntityManager is associated with a PersistenceContext. All operations that are performed in a specific session are stored inside the PersistenceContext. EntityManager is the interface to the Persistence Context. All operations on the entity go through the EntityManager. We will declare an EntityManager object in our class and mark it with the @PersistenceContext annotation.

public class PlayerRepository{  
    @PersistenceContext  
    EntityManager entityManager;  
    //...  
}

EntityManager provides a number of methods that perform SELECT, INSERT, UPDATE, and DELETE queries.

In this lesson, we created a repository and an instance of EntityManager. Now we are ready to perform operations on the database.

The code shown below, when run, creates an empty table of players. We will insert data in the next step.

The database can be viewed in the web browser by typing localhost:8080/h2-console or http://127.0.0.1:8080/h2-console. In the login page that shows up, make sure that the JDBC URL is the same as the one that we provided in the **applications.properties** file (jdbc:h2:mem:testdb). If not, change it to jdbc:h2:mem:testdb and click connect to go to the database console. This will open up the interface of the database.

###### /

PlayerRepository.java

Player.java

TennisPlayerApplication.java

**TennisPlayerApplication.java**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

package io.datajek.springdatajpa;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class TennisPlayerApplication {

    public static void main(String[] args) {

        SpringApplication.run(TennisPlayerApplication.class, args);

    }

}





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/h2-console/>

# CRUD Operations

Learn how to perform CRUD operations using methods offered by the EntityManager.

**We'll cover the following**

* [merge method for INSERT and UPDATE queries](https://www.educative.io/courses/guide-spring-5-spring-boot-2/Bn0zjqkYYgx#merge-method-for-INSERT-and-UPDATE-queries)
* [Executing a query](https://www.educative.io/courses/guide-spring-5-spring-boot-2/Bn0zjqkYYgx#Executing-a-query)
* [The show-sql property](https://www.educative.io/courses/guide-spring-5-spring-boot-2/Bn0zjqkYYgx#The-show-sql-property)
* [find method for SELECT query](https://www.educative.io/courses/guide-spring-5-spring-boot-2/Bn0zjqkYYgx#find-method-for-SELECT-query)
* [remove method for DELETE query](https://www.educative.io/courses/guide-spring-5-spring-boot-2/Bn0zjqkYYgx#remove-method-for-DELETE-query)

The EntityManager offers a large number of methods to perform various queries. We will write methods to insert, fetch, update, and delete data.

## merge method for INSERT and UPDATE queries

1. The EntityManager offers a merge method for both INSERT and UPDATE operations. merge checks if the primary key value is being passed to it or not. If it finds the primary key, it updates the corresponding record. If the primary key is not passed, it generates a value and inserts a new record in the table. The merge method returns a Player object.

The logic of insertPlayer and updatePlayer methods will be the same. The difference lies in the arguments being passed to the merge method.

public Player insertPlayer(Player player)   
{  
   return entityManager.merge(player);  
}  
  
public Player updatePlayer(Player player)   
{  
   return entityManager.merge(player);  
}

The merge method will create an SQL INSERT or UPDATE query and map values to it from the Player object on its own. It also converts the row returned to a Player object on its own.

## Executing a query

1. We will use the CommandLineRunner interface to use the run method.

In order to execute insertPlayer, we will autowire PlayerRepository in the TennisPlayerApplication class and use it to call methods in the run method as follows:

@Autowired  
PlayerRepository repo;  
  
@Override  
public void run(String... args) throws Exception {  
    logger.info("\n\n>> Inserting Player: {}\n", repo.insertPlayer(  
                 new Player("Djokovic", "Serbia", Date.valueOf("1987-05-22"), 81)));  
  
    logger.info("\n\n>> Inserting Player: {}\n", repo.insertPlayer(  
                 new Player("Monfils", "France", Date.valueOf("1986-09-01"), 10)));  
}

This will insert two rows in the Player table. Notice that we are not passing the Id attribute as it will be auto-generated by Hibernate. The insertion can be confirmed from the H2 web console by executing the SELECT \* query. Hibernate has assigned id 1 and 2 to the rows it inserted. The ids would be incremented in sequence if more rows are inserted.

## The show-sql property

1. We have inserted two rows in the database without writing any queries. To see the queries that Hibernate has generated, we can enable the show-sql property in the **application.properties** file as follows:

spring.jpa.show-sql=true

## find method for SELECT query

1. Another way to confirm insert is to implement the getPlayerById method. This method will return a player based on the primary key value. EntityManager offers a number of find methods. We will pick the one that takes the name of the class and the primary key as arguments.

public Player getPlayerById(int id) {  
    return entityManager.find(Player.class, id);                  
}

Like the merge method, the find method also creates the SQL query and maps values to it on its own. It also converts the row returned to a Player object.

We will call getPlayerById in the run method:

@Override  
public void run(String... args) throws Exception {  
    logger.info("\n\n>> Inserting Player: {}\n", repo.insertPlayer(  
               new Player("Djokovic", "Serbia", Date.valueOf("1987-05-22"), 81)));  
    logger.info("\n\n>> Inserting Player: {}\n", repo.insertPlayer(  
               new Player("Monfils", "France", Date.valueOf("1986-09-01"), 10)));  
  
    logger.info("\n\n>> Player with id 2: {}\n", repo.getPlayerById(2));  
}

The database can be viewed in the web browser by typing localhost:8080/h2-console or http://127.0.0.1:8080/h2-console. In the login page that shows up, make sure that the JDBC URL is the same as the one that we provided in the **applications.properties** file (jdbc:h2:mem:testdb). If not, change it to jdbc:h2:mem:testdb and click connect to go to the database console. This will open up the interface of the database.

###### /

TennisPlayerApplication.java

PlayerRepository.java

Player.java

application.properties

**TennisPlayerApplication.java**

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class TennisPlayerApplication implements CommandLineRunner {

    private Logger logger = LoggerFactory.getLogger(this.getClass());

    @Autowired

    PlayerRepository repo;

    public static void main(String[] args) {

        SpringApplication.run(TennisPlayerApplication.class, args);

    }

    @Override

    public void run(String... args) throws Exception {

        logger.info("\n\n>> Inserting Player: {}\n", repo.insertPlayer(new Player("Djokovic", "Serbia", Date.valueOf("1987-05-22"), 81)));

        logger.info("\n\n>> Inserting Player: {}\n", repo.insertPlayer(new Player("Monfils", "France", Date.valueOf("1986-09-01"), 10)));

        logger.info("\n\n>> Player with id 2: {}\n", repo.getPlayerById(2));

    }

}





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/h2-console/>

1. To test the updatePlayer method created in step 1, we will insert a new player Thiem and then update his date of birth as follows:

@Override  
public void run(String... args) throws Exception {  
    //insert players  
   logger.info("\n\n>> Inserting Player: {}\n", repo.insertPlayer(  
            new Player("Djokovic", "Serbia", Date.valueOf("1987-05-22"), 81)));  
   logger.info("\n\n>> Inserting Player: {}\n", repo.insertPlayer(  
            new Player("Monfils", "France", Date.valueOf("1986-09-01"), 10)));  
   logger.info("\n\n>> Inserting Player: {}\n", repo.insertPlayer(  
            new Player("Thiem", "Austria",   
                                  new Date(System.currentTimeMillis()), 17)));    
   //update player  
   logger.info("\n\n>> Updating Player with Id 3: {}\n", repo.updatePlayer(  
            new Player(3, "Thiem", "Austria", Date.valueOf("1993-09-03"), 17)));  
   //get player  
   logger.info("\n\n>> Player with id 3: {}\n", repo.getPlayerById(3));  
}

###### /

TennisPlayerApplication.java

PlayerRepository.java

Player.java

application.properties

**TennisPlayerApplication.java**

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

@SpringBootApplication

public class TennisPlayerApplication implements CommandLineRunner {

    private Logger logger = LoggerFactory.getLogger(this.getClass());

    @Autowired

    PlayerRepository repo;

    public static void main(String[] args) {

        SpringApplication.run(TennisPlayerApplication.class, args);

    }

    @Override

    public void run(String... args) throws Exception {

        //insert players

        logger.info("\n\n>> Inserting Player: {}\n", repo.insertPlayer(

            new Player("Djokovic", "Serbia", Date.valueOf("1987-05-22"), 81)));

        logger.info("\n\n>> Inserting Player: {}\n", repo.insertPlayer(

            new Player("Monfils", "France", Date.valueOf("1986-09-01"), 10)));

        logger.info("\n\n>> Inserting Player: {}\n", repo.insertPlayer(

            new Player("Thiem", "Austria",

                                new Date(System.currentTimeMillis()), 17)));

        //update player

        logger.info("\n\n>> Updating Player with Id 3: {}\n", repo.updatePlayer(

            new Player(3, "Thiem", "Austria", Date.valueOf("1993-09-03"), 17)));

        //get player

        logger.info("\n\n>> Player with id 3: {}\n", repo.getPlayerById(3));

    }

}





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/h2-console/>

## remove method for DELETE query

1. The remove method of EntityManager is used to delete a row from the table. This is a two step process: first, we will get the object using the find method by passing the id of the player. Then, we will pass the object to the remove method. remove is a void method and does not return anything.

public void deleteById(int id){  
    Player player = entityManager.find(Player.class, id);  
    entityManager.remove(player);  
}

When this method is called inside the run method, we can see the DELETE query generated by Hibernate. The delete can be confirmed from the H2 console using a SELECT \* query.

JPA makes interacting with databases really simple. All it needs is entity definition and mapping, and it takes care of the rest.

###### /

TennisPlayerApplication.java

PlayerRepository.java

Player.java

application.properties

**TennisPlayerApplication.java**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

package io.datajek.springdatajpa;

import java.sql.Date;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class TennisPlayerApplication implements CommandLineRunner {

    private Logger logger = LoggerFactory.getLogger(this.getClass());

    @Autowired

    PlayerRepository repo;

    public static void main(String[] args) {

        SpringApplication.run(TennisPlayerApplication.class, args);

    }

    @Override

    public void run(String... args) throws Exception {

        //insert players

        logger.info("\n\n>> Inserting Player: {}\n", repo.insertPlayer(

            new Player("Djokovic", "Serbia", Date.valueOf("1987-05-22"), 81)));

        logger.info("\n\n>> Inserting Player: {}\n", repo.insertPlayer(

            new Player("Monfils", "France", Date.valueOf("1986-09-01"), 10)));

        logger.info("\n\n>> Inserting Player: {}\n", repo.insertPlayer(

            new Player("Thiem", "Austria",

                                new Date(System.currentTimeMillis()), 17)));

        //update player

        logger.info("\n\n>> Updating Player with Id 3: {}\n", repo.updatePlayer(

            new Player(3, "Thiem", "Austria", Date.valueOf("1993-09-03"), 17)));





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/h2-console/>

# JPQL Named Query

Let's write a named query in Java Persistence Query Language (JPQL) in this lesson.

**We'll cover the following**

* [#NamedQuery](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7A5X4xEGxRO##NamedQuery)
* [Using a named query for SELECT \* query](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7A5X4xEGxRO#Using-a-named-query-for-SELECT-*-query)

In this lesson, we will implement the SELECT \* query using a named query in JPQL.

A named query is defined on an entity, which in our case is the Player class. The named query will be used in the getAllPlayers method.

## #NamedQuery

1. To create a named query, we will use the @NamedQuery annotation on the Player class. This annotation requires two parameters: the name of the query and the query itself. When using JPA, we will write the query in JPQL instead of SQL. JPQL uses entities in place of tables. Since we want to return a list of Player objects, the query will be "SELECT p FROM Player p".

@Entity  
@NamedQuery(name="get\_all\_players", query="select p from Player p")  
public class Player {  
    //...  
}

Now we can use the named query in the getAllPlayers method.

## Using a named query for SELECT \* query

In the getAllPlayers method, we will use the createNamedQuery method. We need to pass the name of the query and specify what the query will return. The name of the query as defined in the previous step is get\_all\_players. This query will return objects of the Player class. The createNamedQuery returns a TypedQuery, which we will assign to a variable called getAll. Then, we can use the getResultList method to return a list of players as follows:

public List<Player> getAllPlayers() {  
    TypedQuery<Player> getAll = entityManager.createNamedQuery(  
                                                        "get\_all\_players", Player.class);  
    return getAll.getResultList();  
}

1. We will now call getAllPlayers in the run method to display the records in the Player table.

@Override  
public void run(String... args) throws Exception {  
    logger.info("Inserting Player: {}", repo.insertPlayer(new Player(  
                                     "Djokovic", "Serbia", Date.valueOf("1987-05-22"), 81)));  
    logger.info("Inserting Player: {}", repo.insertPlayer(new Player(  
                                     "Monfils", "France", Date.valueOf("1986-09-01"), 10)));  
    logger.info("Inserting Player: {}", repo.insertPlayer(new Player(  
                                     "Thiem", "Austria", Date.valueOf("1993-09-03"), 17)));   
    logger.info("All Players Data: {}", repo.getAllPlayers());  
}

The database can be viewed in the web browser by typing localhost:8080/h2-console or http://127.0.0.1:8080/h2-console. In the login page that shows up, make sure that the JDBC URL is the same as the one that we provided in the **applications.properties** file (jdbc:h2:mem:testdb). If not, change it to jdbc:h2:mem:testdb and click connect to go to the database console. This will open up the interface of the database.

###### /

TennisPlayerApplication.java

PlayerRepository.java

Player.java

application.properties

**TennisPlayerApplication.java**

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

@SpringBootApplication

public class TennisPlayerApplication implements CommandLineRunner {

    private Logger logger = LoggerFactory.getLogger(this.getClass());

    @Autowired

    PlayerRepository repo;

    public static void main(String[] args) {

        SpringApplication.run(TennisPlayerApplication.class, args);

    }

    @Override

    public void run(String... args) throws Exception {

        //insert players

        logger.info("\n\n>> Inserting Player: {}\n", repo.insertPlayer(

            new Player("Djokovic", "Serbia", Date.valueOf("1987-05-22"), 81)));

        logger.info("\n\n>> Inserting Player: {}\n", repo.insertPlayer(

            new Player("Monfils", "France", Date.valueOf("1986-09-01"), 10)));

        logger.info("\n\n>> Inserting Player: {}\n", repo.insertPlayer(

            new Player("Thiem", "Austria",

                                new Date(System.currentTimeMillis()), 17)));

        //update player

        logger.info("\n\n>> Updating Player with Id 3: {}\n", repo.updatePlayer(

            new Player(3, "Thiem", "Austria", Date.valueOf("1993-09-03"), 17)));

        //get player

        logger.info("\n\n>> Player with id 3: {}\n", repo.getPlayerById(3));

        //delete player

        repo.deletePlayerById(2);

        //get all players

        logger.info("\n\n>> All Players Data: {}", repo.getAllPlayers());

    }

}





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/h2-console/>

# Spring Data JPA

This lesson explores the methods provided by JpaRepository to further simplify the CRUD operations.

**We'll cover the following**

* [JpaRepository interface](https://www.educative.io/courses/guide-spring-5-spring-boot-2/NEAwNA3Dx7N#JpaRepository-interface)
  + [save method](https://www.educative.io/courses/guide-spring-5-spring-boot-2/NEAwNA3Dx7N#save-method)
  + [findById method](https://www.educative.io/courses/guide-spring-5-spring-boot-2/NEAwNA3Dx7N#findById-method)
  + [findAll method](https://www.educative.io/courses/guide-spring-5-spring-boot-2/NEAwNA3Dx7N#findAll-method)
  + [deleteById method](https://www.educative.io/courses/guide-spring-5-spring-boot-2/NEAwNA3Dx7N#deleteById-method)

We have written methods to perform CRUD operations on the Player entity. If we add more entities to the project like **Coach** and **Tournament**, we will have to write the same code for CRUD operations and plug a different entity type.

The methods that we implemented as part of the CRUD operations are all generic methods. The logic of the methods remains the same, and only the entity changes.

Spring Data identified this duplication of code when writing repositories and created some predefined repositories. The developer provides the entity type and its primary key and Spring Data comes up with the CRUD methods for the entity. Spring Data JPA adds a layer of abstraction over the JPA provider (Hibernate in our case).

The JpaRepository interface extends the Repository interface. It contains the API provided by CrudRespository as well as the PagingAndSortingRepository for CRUD operations along with pagination and sorting of records.

## JpaRepository interface

1. We will create an interface that extends the JpaRepository. We will call this interface PlayerSpringDataRepository. We need to specify the entity that will be managed by this repository, as well as the primary key of the entity as follows:

import org.springframework.data.jpa.repository.JpaRepository;  
  
public interface PlayerSpringDataRepository extends JpaRepository<Player, Integer>{  
  
    //no implementation required!  
}

Simply by extending the JpaRepository, we get all basic CRUD operations without having to write any implementation.

1. To use the repository created in the last step, we will create a copy of the TennisPlayerApplication and call it TennisPlayerSpringDataApplication. We also autowire the PlayerSpringDataRepository in place of the PlayerRepository.

@Autowired  
PlayerSpringDataRepository repo;

### save method

1. To insert and update an object, Spring Data has a save method that works in the same manner as the merge method of the EntityManager.

//Inserting rows  
logger.info("Inserting Player: {}", repo.save(new Player("Djokovic", "Serbia",   
                                                Date.valueOf("1987-05-22"), 81)));  
logger.info("Inserting Player: {}", repo.save(new Player("Monfils", "France",   
                                                Date.valueOf("1986-09-01"), 10)));  
logger.info("Inserting Player: {}", repo.save(new Player("Thiem", "Austria",   
                                                new Date(System.currentTimeMillis()), 17)));

//Updating row  
logger.info("Updating Player with Id 3: {}", repo.save(new Player(3, "Thiem", "Austria",   
                                                Date.valueOf("1993-09-03"), 17)));  
}

### findById method

1. Spring Data has a method called findById, which takes in the primary key and returns an object.

logger.info("Player with Id 2: {}", repo.findById(2));

### findAll method

1. It also has a findAll method which returns all entity objects.

logger.info("All Players Data: {}", repo.findAll());

### deleteById method

1. For deletion, Spring Data has a deleteById method that takes the primary key of the record to be deleted.

repo.deleteById(2);

If we use the above mentioned methods, we can run the application without any change in the output.

Using Spring Data, we can run the same application again without writing an implementation for any of the CRUD operations. The JpaRepository provides us with methods needed to perform those operations. This results in a significant reduction in boilerplate code. If we compare the code in the **PlayerRepository.java** with the code in **PlayerSpringDataRepository.java**, the former had more than 20 lines of code while the latter has just 3 lines of code for the same functionality.

The CRUD methods in Spring Data are annotated with @Transactional. Spring Data can parse a method name and create a query from it. In the code widget below, we have a method findByNationality in the **PlayerSpringDataRepository** for which we have not provided any implementation. This method name is converted to the following JPQL query using the JPA Criteria API:

select p from Player p where p.nationality = ?1

We can use keywords such as And, Or, GreaterThan, LessThan, Like, IsNull, Not etc., in the method name and OrderBy clause can be used to sort the results.

###### /

TennisPlayerSpringDataApplication.java

PlayerSpringDataRepository.java

Player.java

application.properties

**PlayerSpringDataRepository.java**

1

2

3

4

5

6

7

8

9

10

11

package io.datajek.springdatajpa;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;

public interface PlayerSpringDataRepository extends JpaRepository<Player, Integer>{

    public List<Player> findByNationality(String nationality);

}





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/h2-console/>

The database can be viewed in the web browser by typing localhost:8080/h2-console or http://127.0.0.1:8080/h2-console. In the login page that shows up, make sure that the JDBC URL is the same as the one that we provided in the **applications.properties** file (jdbc:h2:mem:testdb). If not, change it to jdbc:h2:mem:testdb and click connect to go to the database console. This will open up the interface of the database.

**Connecting to Other Databases**

Learn about the changes required to change the data source.

**We'll cover the following**

* [Replacing the H2 dependency](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JYBMDp3n2zy#Replacing-the-H2-dependency)
* [Configuring property values](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JYBMDp3n2zy#Configuring-property-values)

An in-memory H2 database is easy to set up and use. Now it’s time to learn how to switch to another database. In real-life applications, we might want to connect to Oracle, MySql, or SQL server databases. Using Spring Boot, switching databases is very straightforward. We just need to add some dependencies and change some property values.

This lesson assumes that the database is installed and the schema is properly populated.

**Replacing the H2 dependency**

1. The first step is to remove the H2 dependency from **pom.xml** and replace it with the dependency of the database we wish to connect to. Suppose we wish to connect to MySql. The dependency for MySql is given below. Dependencies for other databases are available online.

<dependency>  
   <groupId>mysql</groupId>  
   <artifactId>mysql-connector-java</artifactId>  
</dependency>

**Configuring property values**

1. For any database, we need to configure its URL, username, and password. These values are specified in the **application.properties** file.

spring.jpa.hibernate.ddl-auto=none   
spring.datasource.url = jdbc:mysql://localhost:3306/movie\_example  
spring.datasource.username = demo  
spring.datasource.password = demo

Since we already have a schema and do not want Hibernate to create it for us, we have set spring.jpa.hibernate.ddl-auto setting to none. Other options for this setting are create-only, drop, create, validate, and update. If we use create instead of none, Hibernate would look at the entities and create the schema itself (after dropping any already existing schema).

The username and password are used to connect to the data source.

1. After making these changes, restart the application and Spring Boot will connect to the database of your choice.

This shows how switching from an in-memory database to a real database is simple in Spring Boot.

# Spring REST

**Developing REST services in Spring**

This lesson discusses basic concepts of REST and Spring's support of RESTful web services.

**We'll cover the following**

* [REST support in Spring](https://www.educative.io/courses/guide-spring-5-spring-boot-2/Bny9VO09Z82#REST-support-in-Spring)
* [What is REST?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/Bny9VO09Z82#What-is-REST?)

**REST support in Spring**

Spring REST

Spring framework is a very good choice for creating REST applications. Spring provides specialized annotations that make RESTful application development easy. Some examples include @RestController, @ResponseBody and @ResponseStatus etc.

Spring also automatically handles Jackson integration which makes conversion of data from JSON to POJO and vice versa a breeze. Any JSON data received by the controller is automatically converted to a POJO and any POJO returned by the controller is converted to JSON.

**What is REST?**

REST stands for the **RE**presentational **S**tate **T**ransfer. It provides a mechanism for communication between applications. In the REST architecture, the client and server are implemented independently and they do not depend on one another. REST is language independent, so the client and server applications can use different programming languages. This gives REST applications a lot of flexibility.

REST Request Response

The REST architecture is stateless meaning that the client only manages session state and the server only manages the resource state. The communication between the client and server is such that every request contains all the information necessary to interpret it without any previous context.

Both the client and server know the communication format and are able to understand the message sent by the other side. REST calls can be made over HTTP. The client can send HTTP request message to the server where it is processed and an HTTP response is sent back.

REST Request Response

The request message has three parts:

1. The request line contains the HTTP method like (GET or POST etc.)
2. The request header contains data with additional information about the request.
3. The request body contains the contents of the message, e.g., if it is a POST request, the request body will contain the contents of the entity to be created.

The response message also has three parts:

1. The response line contains the status code for success or redirection etc.
2. The response header contains additional information about the response like the content type and the size of the response. The client can render the data based on the content type so if it is text/html, it is displayed according to the HTML tags and if it is application/json or application/xml, it is processed accordingly.
3. The response body contains the actual message sent in response to the request.

The HTTP methods for CRUD operations are:

* POST for creating a resource
* GET for reading a resource
* PUT for updating a resource
* DELETE for deleting a resource

HTTP Request Methods

**JSON Data Binding**

In this lesson, we will shed light on the popular data format used in REST applications - JSON.

**We'll cover the following**

* [Syntax](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JYRQkVDpoGg#Syntax)
* [Java - JSON data binding](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JYRQkVDpoGg#Java---JSON-data-binding)
* [Jackson Project](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JYRQkVDpoGg#Jackson-Project)

The most commonly used data formats in a REST application are JSON and XML. JSON stands for **J**ava**S**cript **O**bject **N**otation. It is a plain text data format used for exchanging data between applications.

JSON is a collection of name-value pairs, which the application processes as a string. So, instead of using HTML or JSP to send data, it is passed as a String and the application can process and render the data accordingly. JSON is language independent and can be used with any programming language.

**Syntax**

A sample JSON object is shown below along with rules regarding the syntax.

JSON Object

* A JSON object is defined between curly braces ({ }).
* The object consists of members in the form of comma separated name-value pairs.
* The names and values are separated by colon (:).
* Names are provided in double quotes and are on the left side of the colon.
* The values are on the right side of the colon.
* If the value is a string, it is written in double quotes.
* JSON also supports arrays written within square brackets ([ ]) that contains a comma separated list of values.
* An object can contain nested objects.
* JSON objects can have **null** value.
* Boolean values **true** and **false** are also allowed.

**Java - JSON data binding**

JSON Data Binding

A Java object (POJO) can be converted into a JSON object and vice versa through a process called data binding. We can read JSON and use it to populate a Java object. In the same manner, we can use a Java object to create JSON.

**Jackson Project**

Jackson Project handles data binding between Java and JSON. It also provides support for data binding with XML. Spring framework uses the Jackson project for data binding. The Jackson data binding API is present in the **com.fasterxml.jackson.databind** package.

The following maven dependency adds Jackson support to the project:

<dependency>  
    <groupId>com.fasterxml.jackson.core</groupId>  
    <artifactId>jackson-databind</artifactId>  
    <version>2.12.3</version>  
</dependency>

The latest version at the time of writing this lesson is **2.12.3**.

Jackson handles the conversion between JSON and POJOs by making use of the getter and setter methods of a class. To convert a JSON object to POJO, the setter methods are called. Vice versa, to create a JSON object from a POJO, the getters methods are used. Jackson has access to the getters and setters only, not the private fields of the class.

The figure below illustrates how Jackson converts JSON data to a Java object. It calls the corresponding setter methods to populate the class members. If a setter method matching a JSON property is not found, an exception is thrown.

Creating a POJO from JSON

The Jackson annotation @JsonIgnoreProperties can be used to bypass the exception by setting the IgnoreUnknown attribute to true. This feature is useful when the JSON file contains more properties but we are only interested in a few of them.

**REST Client - Postman**

Learn how to install and use the REST client, Postman.

**We'll cover the following**

* [Install Postman](https://www.educative.io/courses/guide-spring-5-spring-boot-2/N8YjJ9y7nVm#Install-Postman)
* [Web Browser or Postman](https://www.educative.io/courses/guide-spring-5-spring-boot-2/N8YjJ9y7nVm#Web-Browser-or-Postman)

We will be creating a REST Service. To test our service, we need a client tool to send HTTP requests to our REST API and view the response sent back by our service.

There are many REST client tools for testing REST APIs like Curl, which is a command line tool, and Postman, which is a GUI tool.

We will be using **Postman** which is widely used because of its simplicity and advance features.

**Install Postman**

To install Postman, visit the [Download Postman](https://www.postman.com/downloads/) page. You may be prompted to log in or create an account but it is not a requirement as Postman is a free software.

The top part of the screen is for the HTTP request and the bottom part shows the HTTP response. Different request methods like GET, POST, PUT, etc., are available from the dropdown list. You can also view the headers, authorization items, and status codes.

Postman client

To send a request using Postman, select the **HTTP method** from the dropdown list, then type the request URL. If the request sends some data to the server (e.g., POST or PUT), it can be added by selecting the **Body** option. We can also set the format in which data is sent. The request is sent by hitting the **Send** button. The response from the REST Service is displayed in the bottom part of the screen. It shows the status code and may contain some data in the body as well.

**Web Browser or Postman**

For a simple GET request, a web browser can also be used in place of Postman. However, for other types of requests, Postman offers better support like posting JSON data, setting the content type, passing over request headers, and authentication etc.

**REST API Design**

Here we cover the basic design principles for a REST API and start creating one using Spring Initializr.

**We'll cover the following**

* [REST API design](https://www.educative.io/courses/guide-spring-5-spring-boot-2/xlJzlODnEr3#REST-API-design)
* [Required dependencies](https://www.educative.io/courses/guide-spring-5-spring-boot-2/xlJzlODnEr3#Required-dependencies)
* [Data source configuration](https://www.educative.io/courses/guide-spring-5-spring-boot-2/xlJzlODnEr3#Data-source-configuration)
* [Entity Player class](https://www.educative.io/courses/guide-spring-5-spring-boot-2/xlJzlODnEr3#Entity-Player-class)
* [Database Initialization](https://www.educative.io/courses/guide-spring-5-spring-boot-2/xlJzlODnEr3#Database-Initialization)
* [Creating a repository](https://www.educative.io/courses/guide-spring-5-spring-boot-2/xlJzlODnEr3#Creating-a-repository)
* [Service layer](https://www.educative.io/courses/guide-spring-5-spring-boot-2/xlJzlODnEr3#Service-layer)

In this lesson, we will start building a REST API that performs CRUD operations. We will make use of the **tennis player** project to create a REST API to perform CRUD operations on the data stored in the in-memory H2 database.

**REST API design**

Our REST API will expose endpoints which allow a REST client to perform the following functions:

* Get a list of players
* Get a player by ID
* Add a new player
* Update an existing player
* Update the titles of a player
* Delete a player

We will make use of the HTTP GET, POST, PUT, PATCH, and DELETE methods to perform these operations. To create service endpoints for our REST API, we need to identify the *entity* (main resource) which is the most notable noun from the requirements listed above. A glance at our requirements shows that it is **player**. A REST API design convention is to use the plural of the entity as the endpoint, so we will use **/players**.

The following image shows the HTTP methods and their REST endpoints for the corresponding CRUD operations:

REST CRUD operations

As it can be seen from the image, we have not used any action words (e.g., show or create) in the endpoints. Rather, the HTTP method, like GET and POST, defines the action to be performed.

To create a project, we will use [spring initializr](http://start.spring.io/) and provide the following details:

Group ID: **io.datajek** Artifact ID: **tennis-player-rest** Next, we will choose dependencies for the project.

**Required dependencies**

To create a REST API in Spring, we need the spring-webmvc dependency. This dependency supports web as well as RESTful applications. It loads all the supporting dependencies like spring-core, spring-context, spring-web etc., and comes with the embedded Tomcat server.

Secondly, we need the jackson-databind dependency for JSON data binding. By having the Jackson dependency on the classpath, Spring will automatically handle the conversion of JSON data to POJO and vice versa.

Spring Boot makes the task of adding required dependencies easy. Using the **spring initializr**, add the starter dependency for **Web**. The spring-boot-starter-web takes care of both abovementioned dependencies. The transitive dependencies of starter web are shown below:

Dependencies in starter-web

As it can be seen, spring-webmvc and spring-boot-starter-json are included along with other dependencies like spring-boot-starter-tomcat.

Other than spring-boot-starter-web, we will add the spring-boot-starter-data-jpa dependency for Hibernate ORM support, h2 dependency for the in-memory database, and spring-boot-devtools dependency which provides the auto restart feature.

Spring Boot also has built in support for testing and adds the spring-boot-starter-test dependency to the **pom.xml** file.

**Data source configuration**

Spring Boot automatically configures the data source using the entries in the **pom.xml** file where we have added the spring-boot-starter-data-jpa and h2 dependencies. In the **application.properties** file, we can set up the data source URL by using the following property:

spring.datasource.url = jdbc:h2:mem:testdb

**Entity Player class**

Now, we will create a **Player** class for our REST API. We will use the @Entity annotation to map the class to a database table. The name of the table is the same as the class, unless otherwise specified. We will store the player’s id, name, nationality, date of birth and number of titles.

@Entity  
public class Player {  
    @Id  
    @GeneratedValue (strategy=GenerationType.IDENTITY)  
    private int id;  
  
    private String name;  
    private String nationality;  
      
    @JsonFormat(pattern = "dd-MM-yyyy")  
    private Date birthDate;  
  
    private int titles;  
      
    //constructors  
    //getter and setter methods  
}

The @Id and @GeneratedValue annotations are used to mark the primary key and define the manner in which values are generated. By default, dates are saved as Timestamp by Hibernate. When we annotate the birthDate field with @JsonFormat, Jackson will use the provided format for serializing and deserializing the field.

We will create two constructors; a no-argument constructor and a constructor with fields other than the primary key (as it will be auto-generated). We will also generate getters and setters for the fields which are needed by Jackson to handle conversion between Player POJO and JSON.

**Database Initialization**

Hibernate can generate the DDL based on the spring.jpa.hibernate.ddl.auto property. We will use create-drop as its value in the **application.properties** file.

spring.jpa.hibernate.ddl-auto=create-drop  
spring.jpa.show-sql=true

The spring.jpa.show-sql property, when set to true, shows the queries used by Hibernate when creating the database. The database is populated using the **import.sql** file placed at the root of the classpath. This file, shown below, is executed on startup.

INSERT INTO player (ID, Name, Nationality, Birth\_date, Titles) VALUES(1,'Djokovic', 'Serbia', '1987-05-22', 81);  
INSERT INTO player (ID, Name, Nationality, Birth\_date, Titles) VALUES(2,'Monfils', 'France', '1986-09-01', 10);  
INSERT INTO player (ID, Name, Nationality, Birth\_date, Titles) VALUES(3,'Isner', 'USA', '1985-04-26', 15);

**Creating a repository**

We will create an interface **PlayerRepository.java** which extends the JpaRepository interface and provide the entity and the data type of the primary key:

import org.springframework.data.jpa.repository.JpaRepository;   
  
public interface PlayerRepository extends JpaRepository <Player, Integer> {  
  
  
}

Simply by extending the JpaRepository, we get all basic CRUD operations like findAll(), findById(), save(), and deleteById() etc., without having to write any implementation.

**Service layer**

As a best practice, we will introduce a service layer on top of the repository. We will create a class **PlayerService.java** and use the @Service annotation to indicate that this class belongs to the service layer.

To use the PlayerRepository in the service layer, we will autowire it and then delegate calls to the methods provided by the JpaRepository.

@Service  
public class PlayerService {  
      
    @Autowired  
    private PlayerRepository repo;  
  
    //method to return all players  
  
    //method to find player by id  
  
    //method to add player  
  
    //...  
}

TennisPlayerSpringRestApplication.java

PlayerService.java

import.sql

application.properties

PlayerRepository.java

Player.java

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

→   public·String·getNationality()·{

→   →   return·nationality;

→   }

→   public·void·setNationality(String·nationality)·{

→   →   this.nationality·=·nationality;

→   }

→   public·Date·getBirthDate()·{

→   →   return·birthDate;

→   }

→   public·void·setBirthDate(Date·birthDate)·{

→   →   this.birthDate·=·birthDate;

→   }

→   public·int·getTitles()·{

→   →   return·titles;

→   }

→   public·void·setTitles(int·titles)·{

→   →   this.titles·=·titles;

→   }

→   @Override

→   public·String·toString()·{

→   →   return·"\nPlayer·[id=·"·+·id·+·",·name=·"·+·name·+·",·nationality=·"·+·nationality·+·",·birthDate=·"·+·birthDate

→   →   →   →   +·",·titles=·"·+·titles·+·"]";

→   }

}





Back

REST Client - Postman

Next

# Creating a REST Controller

In this lesson, we will create a controller to map requests to our REST API.

**We'll cover the following**

* [@RestController](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7A7r0QLXjZB#@RestController)
* [Adding request mapping](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7A7r0QLXjZB#Adding-request-mapping)
* [Testing the RESTful endpoint](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7A7r0QLXjZB#Testing-the-RESTful-endpoint)

In this lesson, we will create a REST service with a request mapping **/welcome** and send it a request from a REST client (web browser or Postman). The service will respond with a welcome message.

## @RestController

1. The first step is creating a controller class called **PlayerController** with the @RestController annotation. This annotation is an extension of @Controller annotation. The @RestController annotation has support for REST requests and responses and automatically handles the data binding between the Java POJOs and JSON.

@RestController  
public class PlayerController{  
  
}

## Adding request mapping

1. Next, set up a mapping for the request using the @GetMapping annotation as follows:

@GetMapping("/welcome")  
public String welcome() {  
    return "Tennis Player REST API";  
}

The above code creates a RESTful endpoint **/welcome** which can be accessed from the REST client and returns the message string to the client. The @GetMapping annotation is discussed in detail in the next lesson.

## Testing the RESTful endpoint

1. We will test the REST endpoint with a web browser as well as Postman. You can run the code given below and access the results at the link shown below the code widget. If you are working on a local machine, simply type the URL **localhost:8080/welcome** in the web browser and it will return the response.

We can test the same URL (the one given below the code widget or **localhost:8080/welcome**) with Postman as well. For this, we will send a GET request to the URL as shown below:

GET request to /welcome

The response from the REST service appears at the bottom part of the screen.

###### /

PlayerController.java

TennisPlayerSpringRestApplication.java

**PlayerController.java**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

package io.datajek.springrest;

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

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

@RestController

public class PlayerController {

    @GetMapping("/welcome")

    public String welcome() {

        return "Tennis Player REST API";

    }

}





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/welcome>

# @GetMapping

Learn to create a GET mapping in the REST service that returns a list of POJOs as JSON.

**We'll cover the following**

* [findAll()](https://www.educative.io/courses/guide-spring-5-spring-boot-2/YQ2kmrlG6PO#findAll())

Starting from this lesson, we will build a REST service that provides basic CRUD functionality. The client sends an HTTP request to the REST service. The dispatcher servlet handles the request and if the request has JSON data, the HttpMessageConverter converts it to Java objects. The request is mapped to a controller which calls service layer methods. The service layer delegates the call to repository and returns the data as POJO. The MessageConverter converts the data to JSON and it is sent back to the client. The flow of request is shown below:

REST service architecture

In this lesson, we will create an endpoint for retrieving the list of players from the database. The REST client will send a request to **/players**. The REST service will respond with a JSON array of all players.

GET request to /players

We will write the code for the REST service to return List of Player objects when it receives a request for **/players**. The REST API will automatically convert that list of POJOs to JSON when it finds Jackson on the classpath and a JSON response is returned to the client.

1. In order to retrieve the list of players from the database, we will write a method getAllPlayers in the PlayerService class. This method will call a repository method and return a list of objects.

public List<Player> getAllPlayers() {  
    //call repository method  
}

To be able to use methods from the PlayerRepository, we have autowired it in the PlayerService class.

@Service  
public class PlayerService {  
   
    @Autowired  
    PlayerRepository repo;  
   
    //...  
}

## findAll()

1. Remember, that we have used the JpaRepository interface for our PlayerRepository which provides all the basic methods for CRUD operations. This means that there is no need to write any implementation for the methods. We can simply call methods provided by the JpaRepository from the service layer. The JpaRepository provides the findAll() method that returns a List of objects.

public List<Player> getAllPlayers() {  
    return repo.findAll();        
}

1. Now that the service layer is set up, we will move to the controller and create a method on which the **/players** request will be mapped. We will call this method getAllPlayers. Since **/players** is a GET request, we will use the @GetMapping annotation on the method as follows:

@GetMapping("/players")  
public List<Player> getAllPlayers() {  
    //call service layer method  
}

The @GetMapping annotation maps HTTP GET requests to controller methods. It is a shortcut for:

@RequestMapping(method = RequestMethod.GET)

1. The controller method calls the service layer method, getAllPlayers. To be able to use the PlayerService methods, we will autowire PlayerService in the PlayerController class.

@RestController  
public class PlayerController {  
  
    @Autowired  
    PlayerService service;  
  
}

The getAllPlayers method returns a List of Player objects. The REST API will convert that list of POJOs to JSON and return it to the client.

@GetMapping("/players")  
public List<Player> getAllPlayers() {  
    return service.getAllPlayers();       
}

1. To test the REST endpoint, we will use both the browser and Postman as REST clients. In the browser, we will use the URL given below the code widget or type **localhost:8080/players** if working locally. The REST service will respond with a JSON array of all players from the H2 database.
2. In Postman, we can create a GET request using the URL given under the code widget.

The results are displayed at the bottom part of the screen. It can be seen from the header section that the content type is application/JSON.

GET request to /players

###### /

PlayerController.java

PlayerService.java

PlayerRepository.java

Player.java

TennisPlayerSpringRestApplication.java

**PlayerController.javaPlayerService.javaPlayerRepository.javaPlayer.javaTennisPlayerSpringRestApplication.java**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

package io.datajek.springrest;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class TennisPlayerSpringRestApplication{

    public static void main(String[] args) {

        SpringApplication.run(TennisPlayerSpringRestApplication.class, args);

    }

}





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/players>

# @PathVariable

Learn how to use path variables for parameterizing the REST endpoints.

**We'll cover the following**

* [findById](https://www.educative.io/courses/guide-spring-5-spring-boot-2/q2rJWxyJX3p#findById)
* [getOne](https://www.educative.io/courses/guide-spring-5-spring-boot-2/q2rJWxyJX3p#getOne)
* [@PathVariable](https://www.educative.io/courses/guide-spring-5-spring-boot-2/q2rJWxyJX3p#@PathVariable)
* [Error handling](https://www.educative.io/courses/guide-spring-5-spring-boot-2/q2rJWxyJX3p#Error-handling)

In this lesson we will create a new endpoint for our REST API to retrieve a single player based on the id field. We will fetch the required player from the **player** table.

Path variables are a way of parameterizing the path or endpoint to accept data. Path variables are written in curly braces. When the client sends a request, it passes a value in place of the path variable. For example, we could say **/players/1** to give us the player with Id 1, or **/players/3** for player with Id 3.

Path variable

The REST client will send a request to **/players/{playerId}**, where playerId is a path variable. So the actual call may be **/players/2**. The REST service will return the player with id 2 from the **player** table, which is Monfils.

GET request to /players/2

1. JpaRepository interface provides us with methods for all basic CRUD operations. We need to write a service method to call the repository. We will call this method getPlayer. It takes an integer id as input and returns a Player object.

@Service  
public class PlayerService {  
   
    @Autowired  
    PlayerRepository repo;  
  
    public Player getPlayer(int id) {  
        //find and return the player  
    }  
    //...  
 }

## findById

1. To retrieve an entity based on the id, JpaRepository provides the findById method. This method has a return type of Optional. Optional is a design pattern introduced in Java 8, where instead of writing code to check for null values, we can see if a value is present in the Optional.

public Player getPlayer(int id) {  
    return repo.findById(id);     //results in type mismatch error...  
}

Since findById does not return a Player object, we will get a type mismatch error when we try to return the result. The way to go about it, is to create a temporary variable to store the result of the method call. We will create an Optional named tempPlayer.

public Player getPlayer(int id) {  
  
    Optional<Player> tempPlayer = repo.findById(id);  
      
    return null;  
}

We also need a Player object named p which is assigned null value.

public Player getPlayer(int id) {  
  
    Optional<Player> tempPlayer = repo.findById(id);  
      
    Player p = null;  
   
    return p;  
}

Next, we will check the tempPlayer using the isPresent method to check if a value is present. The get method can be used to retrieve the value and assign it to p. Player p is returned at the end of the method.

public Player getPlayer(int id) {  
  
    Optional<Player> tempPlayer = repo.findById(id);      
    Player p = null;  
  
    //if the Optional has a value, assign it to p         
    if(tempPlayer.isPresent())  
        p = tempPlayer.get();  
  
    return p;  
}

In case the if condition evaluates to false, meaning that the Player was not found, we can throw an exception to let the user know that the player with the given ID does not exist.

public Player getPlayer(int id) {  
  
    Optional<Player> tempPlayer = repo.findById(id);  
      
    Player p = null;  
      
    //if the Optional has a value, assign it to p         
    if(tempPlayer.isPresent())  
        p = tempPlayer.get();  
  
    //if value is not found, throw a runtime exception        
    else  
        throw new RuntimeException("Player with id "+ id + " not found.");  
      
    return p;  
}

So, in this approach of coding, we used Optional and avoided explicitly checking for null values.

## getOne

Alternatively, the JpaRepository provides the getOne method which returns an object. This method throws an EntityNotFoundException if the object does not exist. When using this method, we explicitly need to check for null values.

1. In the controller class, we will create a method getPlayer for retrieving a player by id. The endpoint **/players/{id}** maps to this method. We will set up a @GetMapping for **/players/{id}**, where id in curly braces means that it is a path variable.

@GetMapping("/players/{id}")  
public Player getPlayer() {  
    //call service layer method  
}

Behind the scenes, Jackson will convert the player object to JSON.

## @PathVariable

1. Since there is a path variable in the endpoint, we need to bind it with a method parameter. The @PathVariable annotation binds the path variable {id} from the URL to the method parameter id. By default, both the names must be the same for the binding to work.

public Player getPlayer(@PathVariable int id) {  
    return service.getPlayer(id);         
}

If the names of the path variable and the method parameter are different, then we need to specify the path variable as argument of the @PathVariable annotation. In the code snippet below, the path variable id will bind with the variable playerId.

public Player getPlayer(@PathVariable("id") int playerId) {  
  
}

1. To test the newly created endpoint, we will use the url given below the code widget (or [**http://localhost:8080/players/2**](http://localhost:8080/players/2) if working locally), where 2 is the player id which will be used to fetch a player record from the database. Id 2, corresponds to Monfils.

We can test the new endpoint in both the browser and through Postman.

## Error handling

1. If a GET request is sent for a player record which is not present in the list or the user enters a character in place of an int for the path variable, the application will throw an error.

###### /

PlayerController.java

PlayerService.java

PlayerRepository.java

Player.java

TennisPlayerSpringRestApplication.java

**Player.java**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

package io.datajek.springrest;

import java.util.Date;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import com.fasterxml.jackson.annotation.JsonFormat;

@Entity

//@JsonIgnoreProperties({"hibernateLazyInitializer"}) Needed if No Serializer Found error is encountered

public class Player {

    @Id

    @GeneratedValue(strategy=GenerationType.IDENTITY)

    private int id;

    private String name;

    private String nationality;

    @JsonFormat(pattern = "dd-MM-yyyy")

    private Date birthDate;

    private int titles;

    public Player() {

    }

    public Player(String name, String nationality, Date birthDate, int titles) {

        super();

        this.name = name;

        this.nationality = nationality;

        this.birthDate = birthDate;





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/players/2>

# @PostMapping

Learn how to add a record to the database using a POST request.

**We'll cover the following**

* [save method](https://www.educative.io/courses/guide-spring-5-spring-boot-2/YM9ZjMDPXNM#save-method)
* [@PostMapping](https://www.educative.io/courses/guide-spring-5-spring-boot-2/YM9ZjMDPXNM#@PostMapping)
* [@RequestBody](https://www.educative.io/courses/guide-spring-5-spring-boot-2/YM9ZjMDPXNM#@RequestBody)
* [Creating a POST request](https://www.educative.io/courses/guide-spring-5-spring-boot-2/YM9ZjMDPXNM#Creating-a-POST-request)

In this lesson, we will create an endpoint for the REST API which creates a new player and adds it to the database.

The REST client will send a POST request to **/players**. The body of the request contains the JSON data for a player. Since this is a new player, the client will not pass the ID (primary key). The backend system will generate the key for the new record.

The REST service will convert JSON data to POJO and add it to the database. The primary key of the added player is automatically generated by Hibernate, which is the default ORM used by Spring Data JPA. The response to the client is an echo of the player details along with the newly generated ID value.

POST request to /players

1. We will begin by writing the service layer method to add a player. This method, addPlayer takes a Player object as parameter and returns the entity that has been added.

public Player addPlayer(Player p) {  
    //call repository method to add a player object to the player table  
}

## save method

1. The JpaRepository interface inherits a method from the CrudRepository called save. This method handles both inserts and updates. To distinguish between an INSERT and UPDATE operation, the save method checks the primary key of the object that is being passed to it. If the primary key is empty or null, an INSERT operation is performed, otherwise an UPDATE to an existing record is performed.

public Player addPlayer(Player p) {  
    return repo.save(p);  
}

The calling method will ensure that the primary key, Id, in the Player object is empty so that the record gets inserted.

1. After writing the service layer method for adding a player, we are ready to move on to the controller layer. We will create a method addPlayer in the **PlayerController** class. This method will have a mapping for a POST request to **/players**. The method will return the inserted record back to the client.

@PostMapping("/players")  
public Player addPlayer(Player player) {  
  
}

## @PostMapping

The @PostMapping annotation maps HTTP POST requests to controller methods. It is a shortcut annotation for:

@RequestMapping(method = RequestMethod.POST)

## @RequestBody

The client will send the player data in the request body as JSON. Jackson will convert the incoming JSON data to POJO. The @RequestBody annotation handles this conversion and binds the data in the request body to a method parameter.

@PostMapping("/players")  
public Player addPlayer(@RequestBody Player player) {  
  
}

In the above code snippet, the @RequestBody annotation binds the JSON from the request to the Player object. It converts JSON to POJO without us having to parse the request body. We can directly use the data in the player object now.

1. Inside the addPlayer method, we will set the primary key to zero. This is done to ensure that if the client accidentally passes the id of a player to be added, we remove it from the request before delegating the call to the service layer. The save method offered by the JpaRepository works for both INSERT and UPDATE requests by checking the primary key and performs an INSERT or UPDATE operation depending upon its value.

@PostMapping("/players")  
public Player addPlayer(@RequestBody Player player) {  
    player.setId(0);  
}

We are explicitly setting the Id to zero to ensure the insertion of a new player instead of an update to an existing player. By overwriting the Id with zero, we are effectively setting it to null or empty.

Then, we just delegate this request to the service layer.

@PostMapping("/players")  
public Player addPlayer(@RequestBody Player player) {  
    player.setId(0);  
    return service.addPlayer(player);  
}

The response from the REST controller will contain the player’s name, nationality, birth date, and titles. It will also contain the Id that was automatically generated by Hibernate.

## Creating a POST request

1. We can now test the newly created endpoint with Postman. To send data to a REST service, we need to create a POST request and use the URL given below the code widget.

When sending JSON data to a REST controller we need to set an HTTP request header. The information in the header tells the controller how to process the data.

* + To set up the request, we will choose **Body**, and then choose a radio item for **raw** because we will send raw data in the request.
  + A dropdown on the right contains different content types. We will choose the content type of **JSON**, because our controller method expects JSON data.

These settings ensure that Postman will automatically set the correct HTTP request header and then send the data across accordingly.

Setting up a POST request in Postman

The actual body of the request consists of JSON for the player containing the name, nationality, birth date, and titles information. We will not pass the id because the system will auto generate that value for us. The controller method also checks the Id and sets it to zero, even if it is passed in the request.

{  
    "name": "Federer",  
    "nationality": "Switzerland",  
    "birthDate": "22-11-1984",  
    "titles": 151  
}

When we send the request, the response from the REST service appears in the bottom part of the screen. The player has been added to the database and Hibernate has generated a new ID for the player as 4.

1. We can verify the INSERT by sending a GET request to the REST API at \_\_/players\_\_. The response now contains 4 players, which verifies that a player has been added to the database.

###### /

PlayerController.java

PlayerService.java

PlayerRepository.java

Player.java

TennisPlayerSpringRestApplication.java

**Player.java**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

package io.datajek.springrest;

import java.util.Date;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import com.fasterxml.jackson.annotation.JsonFormat;

@Entity

public class Player {

    @Id

    @GeneratedValue(strategy=GenerationType.IDENTITY)

    private int id;

    private String name;

    private String nationality;

    @JsonFormat(pattern = "dd-MM-yyyy")

    private Date birthDate;

    private int titles;

    public Player() {

    }

    public Player(String name, String nationality, Date birthDate, int titles) {

        super();

        this.name = name;

        this.nationality = nationality;

        this.birthDate = birthDate;

        this.titles = titles;





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/players>

# @PutMapping

Learn how to update a record in the database using a PUT request.

**We'll cover the following**

* [@PutMapping](https://www.educative.io/courses/guide-spring-5-spring-boot-2/my3n2Y8PPw9#@PutMapping)

In this lesson, we will create an endpoint to edit a player record and update his information in the database.

The HTTP PUT request is used for updates. The REST client will send a PUT request to **/players/{playerId}** with JSON data containing the information to be updated. The player’s ID is a path variable.

The REST service will convert the JSON data to Player object and using the Id of the player, it will overwrite the player’s record in the database with the one sent in the PUT request. On success, the REST service will respond with the updated player record (which is an echo of the request).

PUT request to /players/4

1. To handle the update based on the player’s ID, we will create a method called updatePlayer in the PlayerService class. This method takes in two arguments, a player’s Id and a Player object. It returns the updated Player object.

public Player updatePlayer(int id, Player p) {  
    //call repository method to update the player  
}

1. The primary key passed in the method will be used to fetch the existing record from the database. We will use the getOne method as follows:

public Player updatePlayer(int id, Player p) {  
      
    //get player object by Id  
    Player player = repo.getOne(id);  
      
    //update player information in database  
}

1. Once we get the old player record from the database in player, we need to modify it based on the information in the Player object p, received as argument.

For each field, we will call the getter methods on p to fetch the new information. Then, we will overwrite the old information by calling setter methods on player.

//update player details  
player.setName(p.getName());  
player.setNationality(p.getNationality());  
player.setBirthDate(p.getBirthDate());  
player.setTitles(p.getTitles());

1. The last step is to call repository methods to save the updated information. JpaRespository inherits the save method offered by the CrudRepository. This method handles both insert and update operations on the repository. It works by checking the primary key of the object and performs an update if the key is present. If the primary key is zero or null, a new record is inserted into the database.

In our case, the player object contains a non-null id value. So, if we call the save method, it will perform an update of the existing record.

public Player updatePlayer(int id, Player p) {  
      
    //get player object by ID  
    Player player = repo.getOne(id);  
      
    //update player details  
    player.setName(p.getName());  
    player.setNationality(p.getNationality());  
    player.setBirthDate(p.getBirthDate());  
    player.setTitles(p.getTitles());  
      
    //save updates  
    return repo.save(player);  
}

The save method returns the object.

1. After setting up the service layer method to handle updates, we can write the controller method to receive a PUT request. We will send the Id of the player to be updated as a path variable. The request body contains the JSON with the updated information of the player. The method will return a Player object which will be converted to JSON and sent back to the client.

The following code snippet shows the updatePlayer method with a @PutMapping of **/players/{id}**:

@PutMapping("/players/{id}")  
public Player updatePlayer(@PathVariable int id, @RequestBody Player player) {  
   
}

## @PutMapping

The @PutMapping is a shortcut annotation for mapping a PUT request to a controller method. It is the same as:

@RequestMapping(method=RequestMethod.PUT)

The updatePlayer method accepts JSON data in the request body. The @RequestBody annotation binds the JSON data to the Player object player. It handles the conversion from JSON to POJO.

We have also used the @PathVariable annotation which will extract the path variable id from the incoming request **/players/{id}** and bind it with the id method parameter.

The controller method delegates the call to the service layer by calling its update method and passing the id and the Player object containing the information to be updated:

@PutMapping("/players/{id}")  
public Player updatePlayer(@RequestBody Player player, @PathVariable int id) {  
    return service.updatePlayer(id, player);  
}

1. To test the new endpoint, we will create a PUT request in Postman. Suppose we want to update the information of player with id 2 so the URL is **/players/2**. We will provide the data for a new player, Nadal. To send the JSON in the request, choose **Body**, then choose **raw** and from the dropdown choose **JSON**.

The body of the request is as follows:

{  
    "name" : "Nadal",  
    "nationality": "Spain",  
    "birthDate" : "03-06-1986",  
    "titles" : 88  
}

When the request is sent, we can see a response at the bottom part of the screen. The response is an echo of the request and indicates that the request has been successful. The player with id 2 is now Nadal.

###### /

PlayerController.java

PlayerService.java

PlayerRepository.java

Player.java

TennisPlayerSpringRestApplication.java

**PlayerService.java**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

package io.datajek.springrest;

import java.util.List;

import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

@Service

public class PlayerService {

    @Autowired

    PlayerRepository repo;

    //Get all players

    public List<Player> getAllPlayers() {

        return repo.findAll();

    }

    //Get player by ID

    public Player getPlayer(int id) {

        Optional<Player> tempPlayer = repo.findById(id);

        if(tempPlayer.isEmpty())

            throw new RuntimeException("Player with id {"+ id +"} not found");

        return tempPlayer.get();

    }

    //Update a player

    public Player updatePlayer(int id, Player p) {

        Optional<Player> tempPlayer = repo.findById(id);





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/players>

# @PatchMapping

In this lesson, we will learn how to perform partial updates to a record in the database.

**We'll cover the following**

* [Partial update](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7DgPR2KrQ5w#Partial-update)
* [Using reflection](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7DgPR2KrQ5w#Using-reflection)
* [@PatchMapping](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7DgPR2KrQ5w#@PatchMapping)
* [Queries for partial update](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7DgPR2KrQ5w#Queries-for-partial-update)

## Partial update

The PUT method updates the whole record. There may be a scenario when only one or two fields needs to be updated. In that case, sending the whole record does not make sense. The HTTP PATCH method is used for partial updates.

Sometimes we may need to update a single field. For example, once we enter a player in our database, the field that will most likely change is his **titles** count. The player entity only has a few fields and PUT can be used for update. But if the entity is large and contains nested objects, it will have a performance impact to send the whole entity only to update a single field.

So, in our example, partial request means that we only send the titles in the request body instead of the whole Player object. If we use PUT to send a partial request, all other fields are set to null. The code widget below illustrates the point if a PUT request with the following request body is sent to **/players/1**:

{  
    "titles": 161  
}

We get the following response:

{  
    "id": 1,  
    "name": null,  
    "nationality": null,  
    "birthDate": null,  
    "titles": 161  
}

###### /

PlayerController.java

PlayerService.java

PlayerRepository.java

Player.java

TennisPlayerSpringRestApplication.java

**PlayerController.java**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

package io.datajek.springrest;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;

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

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

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

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

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

@RestController

public class PlayerController {

    @Autowired

    PlayerService service;

    @GetMapping("/players")

    public List<Player> allPlayers() {

        return service.getAllPlayers();

    }

    @PutMapping("/players/{id}")

    public Player updatePlayer(@PathVariable int id, @RequestBody Player player) {

        return service.updatePlayer(id, player);

    }

}





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/players>

The titles have been modified but the rest of the values are now null which is not the desired outcome. The PUT method requires the entire object to be sent, even when we want to modify a single field. If partial data is sent, then all other fields are set to null. PATCH comes in handy in such situations. It allows a list of changes to be applied to the entity, as we will see in this lesson.

PATCH request to /players/1

1. In the **PlayerService** class, we will implement a method to handle partial updates to the Player object. The method patch will have two arguments. The first is the id of the player on which the patch is to be applied. The second argument is the Map containing the key-value pairs of the fields that will be updated. The key (field name) is a String while the value is an Object as it can have different datatypes.

public Player patch( int id, Map<String, Object> playerPatch) {  
  
}

Inside the method, we will use the id to fetch the existing Player object from the database using the findById method of the JpaRepository. This method loads the entity from the database unlike the getOne method, which returns a proxy object without hitting the database. The findById returns an Optional and we need to check if a Player object is returned using the isPresent() method.

public Player patch( int id, Map<String, Object> playerPatch) {  
  
    Optional<Player> player = repo.findById(id);  
    if (player.isPresent()){  
        //update fields using Map  
    }  
    return repo.save(player);                 
}

## Using reflection

1. Next, we will loop through the Map, find the field that will be updated, and then change the value of that field in the existing Player object that we retrieved from the database in the previous step. The Reflection API is used to examine and modify fields, methods, and classes at runtime. It allows access to the private fields of a class and can be used to access the fields irrespective of their access modifiers. Spring provides the **ReflectionUtils** class for handling reflection and working with the Reflection API.
   * The **ReflectionUtils** class has a findField method to identify the field of an object using a String name. The findField method takes two arguments, the class having the field and the name of the field which in our case is contained in the variable key. This method will return a Field object.

Field field = ReflectionUtils.findField(Player.class, key);

* + To set a value for this field, we need to set the field’s accessible flag to true. **ReflectionUtils** setAccessible method, when called on a field, toggles its accessible flag. We can also use another method called makeAccessible. This method makes the given field accessible by calling the setAccessible(true) method if necessary.

ReflectionUtils.makeAccessible(field);

* + Lastly, we will call the setField method and use the value from the Map to set the field in the player object. The setField method takes three arguments, the reference of the field, the object in which the field is to be set, and the value to set. This method requires that the given field is accessible.

ReflectionUtils.setField(field, player.get(), value);

Here, we have used the get method on the Optional player object to retrieve it.

1. In this way using reflection, a field can be updated in an object. Since we are passing the fields to be updated as a Map, we will use the above steps while iterating through the map of key-value pairs as follows:
2. playerPatch.forEach( (key, value) -> {  
       Field field = ReflectionUtils.findField(Player.class, key);  
       ReflectionUtils.makeAccessible(field);  
       ReflectionUtils.setField(field, player.get(), value);  
   });
3. This code will iterate through the Map and make desired changes in the player object. At the end we will call the save method to update the player record. The complete code of the method is shown below:
4. public Player patch( int id, Map<String, Object> playerPatch) {  
     
       Optional<Player> player = repo.findById(id);  
         
       if(player.isPresent()) {              
           playerPatch.forEach( (key, value) -> {  
               Field field = ReflectionUtils.findField(Player.class, key);  
               ReflectionUtils.makeAccessible(field);  
               ReflectionUtils.setField(field, player.get(), value);  
           });  
       }  
       return repo.save(player.get());               
   }
5. Now, we will write a controller method called partialUpdate to handle PATCH mapping. It will have an @PatchMapping annotation with endpoint **/players/{id}**, where id is a path variable. The method takes a Map argument containing the key-value pair of the fields we want to update. Since the field names are String and the values can be any datatype, we will use Map<String, Object>. The list of fields and their values will come in the request body and the @RequestBody annotation binds the JSON to the Map variable.

@PatchMapping("/players/{id}")  
public Player partialUpdate( @PathVariable int id,   
                             @RequestBody Map<String, Object> playerPatch) {  
    //call service layer method for patch     
  
}

## @PatchMapping

The @PatchMapping is a shortcut annotation. It is the same as:

@RequestMapping(method = RequestMethod.PATCH);

Inside the method, we will simply delegate the call to the service method and pass the player id and the Map with the fields to be updated, along with their values.

@PatchMapping("/players/{id}")  
public Player partialUpdate( @PathVariable int id,   
                             @RequestBody Map<String, Object> playerPatch) {  
    return service.patch(id, playerPatch);            
}

1. To test, use Postman to send a PATCH request to **/players/1**. For sending JSON data in request body, choose **Body**, **raw** and select **JSON** as type. The request body will look like this:

{  
    "titles": 161  
}

A GET request to **/players** can be sent to verify that the patch has indeed been applied.

###### /

PlayerController.java

PlayerService.java

PlayerRepository.java

Player.java

TennisPlayerSpringRestApplication.java

**PlayerController.java**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

package io.datajek.springrest;

import java.util.List;

import java.util.Map;

import org.springframework.beans.factory.annotation.Autowired;

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

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

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

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

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

@RestController

public class PlayerController {

    @Autowired

    PlayerService service;

    @GetMapping("/players")

    public List<Player> allPlayers() {

        return service.getAllPlayers();

    }

    @PatchMapping("/players/{id}")

    public Player partialUpdate(@PathVariable int id, @RequestBody Map<String, Object> playerPatch) {

        return service.patch(id, playerPatch);

    }

}





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/players>

## Queries for partial update

A point to note here is that in step 2, we have used the save method to applying the patch. This method updates all the columns in the table. For large objects with a lot of fields, this can have a performance impact. To avoid this, we can implement queries for partial updates. These queries can target frequently updated columns. If we want to update the **titles** column of the **player** table, we can create an updateTitles method implementing a query in the PlayerRepository. This method takes two arguments, the id of the player and the title count. The @Query annotation is used to implement an update query as follows:

@Modifying  
@Query("update Player p set p.titles = :titles where p.id = :id")  
void updateTitles(@Param("id") int id, @Param("titles") int titles);

The query must be used with the @Modifying annotation to execute the UPDATE query. The @Param annotation binds the method parameters to the query. This method will only change a single column of the table unlike the save method which updates all the columns of the table.

After writing the repository method, we will move to the service layer. The service layer will implement the updateTitles method as follows:

@Transactional  
public void updateTitles(int id, int titles) {  
    repo.updateTitles(id, titles);  
}

The @Transactional annotation ensures that the database is left in a consistent state. The transaction is either committed or rolled back in case of failure.

Now, in the PlayerController class, we can define a new PATCH mapping for **/players/{id}/titles** as follows:

@PatchMapping("/players/{id}/titles")  
public void updateTitles(@PathVariable int id, @RequestBody int titles) {  
    service.updateTitles(id, titles);             
}

id is the path variable. This method accepts an integer from the request body. To test the new endpoint, we can send a PATCH request to **/players/1/titles**. The request body contains an integer value of the title count:

157

The REST API responds with 200 status code which indicates that the request was successful.

###### /

PlayerController.java

PlayerService.java

PlayerRepository.java

Player.java

TennisPlayerSpringRestApplication.java

**PlayerController.java**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

package io.datajek.springrest;

import java.util.List;

import java.util.Map;

import org.springframework.beans.factory.annotation.Autowired;

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

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

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

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

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

@RestController

public class PlayerController {

    @Autowired

    PlayerService service;

    @GetMapping("/players")

    public List<Player> allPlayers() {

        return service.getAllPlayers();

    }

    @GetMapping("/players/{id}")

    public Player getPlayer(@PathVariable int id){

        return service.getPlayer(id);

    }

    @PatchMapping("/players/{id}")

    public Player partialUpdate(@PathVariable int id, @RequestBody Map<String, Object> playerPatch) {

        return service.patch(id, playerPatch);

    }



Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/players>

# @DeleteMapping

In this lesson, we show how to delete a record from the database using DELETE request.

**We'll cover the following**

* [@DeleteMapping](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qV8PAjoMpA2#@DeleteMapping)

The HTTP DELETE request deletes a record. The primary key of the record to be deleted can be sent as part of the request URI or the record itself can be sent as part of the request body. The client will send a DELETE request to our REST service with the id of the player to be deleted. The REST Service deletes the record and responds with the 200 (OK) status code to the client.

DELETE request to /players/3

JpaRepository inherits two methods of the CrudRepository for deleting a record. One is the delete method which takes an entity to be deleted, and the other is deleteById which takes the primary key of the entity to be deleted. They both have the same function, and internally the deleteById method calls the delete method:

void deleteById(ID id) {   
    delete(findById(id));   
}

The difference lies in the way both methods respond when the entity to be deleted is not found. The deleteById method throws the **EmptyResultDataAccessException** while delete throws a **NoSuchElementException**.

1. Following the same approach adopted in the previous lessons, we will first define the service layer method to handle delete functionality. In the PlayerService class, create a method called deletePlayer. This method takes an int argument which is the primary key of the player to be deleted. It returns a String message to indicate success or failure of the delete operation.

public String deletePlayer(int id) {  
    //call repo method for delete  
}

1. Inside the method, we will call the repository method deleteById and pass the player’s id. As mentioned above, the deleteById method internally calls the findById method, and then deletes the record.

public String deletePlayer(int id) {              
    repo.deleteById(id);  
    return "Deleted player with id: "+id;  
}

To handle the exception thrown when the player record is not found, we can enclose the method call in a try catch block:

try {  
    repo.deleteById(id);  
    } catch(Exception e) {  
      return "Player with id " + id + "not found";  
}  
      
return "Deleted player with id: " + id;

1. Moving on to the controller layer, we will create a method deletePlayer to handle the DELETE request. It will map to the endpoint **/players/{id}**, with **id** being the path variable. The REST controller will respond with a message informing the client of the success or failure. The method is annotated with @DeteleMapping. The @PathVariable annotation binds the id path variable to the method parameter id.

@DeleteMapping("/players/{id}")  
public String deletePlayer(@PathVariable int id) {  
    //call service layer method   
}

## @DeleteMapping

The @DeleteMapping annotation is a shortcut annotation. It is the same as:

@RequestMapping(method = RequestMethod.DELETE)

The deletePlayer method will delegate the call to the service layer and pass the id received in the DELETE request. On success, the method will return the deleted player’s id by plugging the id received in the request. If the player is not found in the database, the method will return with a message informing the client that the player was not found.

@DeleteMapping("/players/{id}")  
public String deletePlayer(@PathVariable int id) {  
    return service.deletePlayer(id);          
}

1. To test the new endpoint, launch the application given below, and create a **DELETE**\_ request in Postman. We will provide the id of the player as part of the URI, so the request body will be empty. Send two DELETE requests, first to **/players/3** and then to **/players/4**. In the first case, the REST controller responds with a success message, while in the second case the player with id 4 is not found.
2. The results can be verified by using a **GET** request to **/players**. It can be seen that only 2 players remain in the database.

###### /

PlayerController.java

PlayerService.java

PlayerRepository.java

Player.java

TennisPlayerSpringRestApplication.java

**Player.java**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

package io.datajek.springrest;

import java.util.Date;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import com.fasterxml.jackson.annotation.JsonFormat;

@Entity

public class Player {

    @Id

    @GeneratedValue(strategy=GenerationType.IDENTITY)

    private int id;

    private String name;

    private String nationality;

    @JsonFormat(pattern = "dd-MM-yyyy")

    private Date birthDate;

    private int titles;

    public Player() {

    }

    public Player(String name, String nationality, Date birthDate, int titles) {

        super();

        this.name = name;

        this.nationality = nationality;

        this.birthDate = birthDate;

        this.titles = titles;





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/players>

**Exception Handling I**

In this lesson, we will learn how to throw an exception of a custom type.

**We'll cover the following**

* [Custom error response class](https://www.educative.io/courses/guide-spring-5-spring-boot-2/q2PEoY3vmQ7#Custom-error-response-class)
* [Custom exception class](https://www.educative.io/courses/guide-spring-5-spring-boot-2/q2PEoY3vmQ7#Custom-exception-class)
* [throw exception](https://www.educative.io/courses/guide-spring-5-spring-boot-2/q2PEoY3vmQ7#throw-exception)

When the client sends a request to fetch, update or delete a player record not found in the database, an internal server error occurs. The information contained in the response is verbose and of interest to developers only.

We will learn how to send a client-friendly JSON response when a player record is not found. The message can contain the error code, time when the error occurred, and information about why the error occurred. So when the client sends a request for an Id that does not exist, say 537, the REST Service sends it a custom JSON response instead of the 500 Internal Server Error.

Custom error response

**Custom error response class**

1. In order to send a custom response to the client, we will create a class with fields like status code, error message, path/URI, and timestamp. An object of this class will be created when an exception occurs and sent back to the client as the response. Jackson will automatically handle data binding and send this object as a JSON response.

Create a class PlayerErrorResponse in **io.datajek.springrest**. The class contains the following fields:

public class PlayerErrorResponse {  
    @JsonFormat(pattern="yyyy-MM-dd HH:mm:ss")  
    private ZonedDateTime timestamp;  
    private int statusCode;  
    private String path;  
    private String message;  
  
    //constructor  
    //getters and setters  
}

The class has a constructor to create an object using fields as well as getter and setter methods for all fields.

**Custom exception class**

1. When the REST service receives a bad request, we want it to throw an exception of our custom type. Since the exception is thrown when the player is not found , we will call the class PlayerNotFoundException. This class extends the RunTimeException class.

public class PlayerNotFoundException extends RuntimeException{  
  
    //superclass constructors  
}

The RunTimeException class contains multiple constructors and we will use one of them to throw an exception when the player is not found.

**throw exception**

1. Now that we have defined a custom exception class, we are in a position to throw exceptions of this class. PlayerNotFoundException will be thrown at multiple places in the code. In the PlayerService class, the player Id is received by the following methods: getPlayer, updatePlayer, patch, updateTitles, and deletePlayer. We will update these methods to check if the Id received is valid and throw a PlayerNotFoundException if there is no record in the database for the given Id.

In the getPlayer method, the player record is retrieved in an Optional called tempPlayer. The isEmpty method on the Optional checks for null values and the PlayerNotFoundException is thrown as shown below:

public Player getPlayer(int id) {  
    Optional<Player> tempPlayer = repo.findById(id);  
  
    if(tempPlayer.isEmpty())  
        throw new PlayerNotFoundException("Player with id {"+ id +"} not found");  
      
    return tempPlayer.get();  
}

The exception is created with a custom message containing the Id of the player.

We will use a similar approach in the updatePlayer, deletePlayer and updateTitles methods. First, use the given Id to retrieve the player from the database. If the player record does not exist, throw PlayerNotFoundException, else, update/delete the player record. The modified methods are shown below:

public Player updatePlayer(int id, Player p) {  
    Optional<Player> tempPlayer = repo.findById(id);  
  
    if(tempPlayer.isEmpty())  
        throw new PlayerNotFoundException("Player with id {"+ id +"} not found");  
  
    p.setId(id);  
    return repo.save(p);  
}

public void deletePlayer(int id) {  
    Optional<Player> tempPlayer = repo.findById(id);  
  
    if(tempPlayer.isEmpty())  
        throw new PlayerNotFoundException("Player with id {"+ id +"} not found");  
  
    repo.delete(tempPlayer.get());  
}

@Transactional  
public void updateTitles(int id, int titles) {  
    Optional<Player> tempPlayer = repo.findById(id);  
  
    if(tempPlayer.isEmpty())  
        throw new PlayerNotFoundException("Player with id {"+ id +"} not found");  
  
    repo.updateTitles(id, titles);  
}

For the patch method, we will take a slightly different approach and use isPresent() instead of isEmpty() on the Optional. If the record has been retrieved, we will apply the patch, otherwise throw the PlayerNotFoundException. The code is shown below:

public Player patch( int id, Map<String, Object> partialPlayer) {  
    Optional<Player> player = repo.findById(id);  
   
    if(player.isPresent()) {              
        partialPlayer.forEach( (key, value) -> {  
            System.out.println("Key: " + key + " Value: " + value);  
            Field field = ReflectionUtils.findField(Player.class, key);  
           ReflectionUtils.makeAccessible(field);  
           ReflectionUtils.setField(field, player.get(), value);  
        });  
    }  
    else  
       throw new PlayerNotFoundException("Player with id {"+ id +"} not found");  
   
    return repo.save(player.get());               
}

PlayerNotFoundException.java

PlayerErrorResponse.java

PlayerService.java

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

package io.datajek.springrest;

import java.time.ZonedDateTime;

import com.fasterxml.jackson.annotation.JsonFormat;

public class PlayerErrorResponse {

    @JsonFormat(pattern="yyyy-MM-dd HH:mm:ss")

    private ZonedDateTime timestamp;

    private int statusCode;

    private String path;

    private String message;

    public PlayerErrorResponse(ZonedDateTime timestamp, int statusCode, String path, String message) {

        super();

        this.timestamp = timestamp;

        this.statusCode = statusCode;

        this.path = path;

        this.message = message;

    }

    public ZonedDateTime getTimestamp() {

        return timestamp;

    }

    public void setTimestamp(ZonedDateTime timestamp) {

        this.timestamp = timestamp;

    }

    public int getStatusCode() {

# Exception Handling II

In this lesson, we will learn how a global exception handler works.

**We'll cover the following**

* [@ControllerAdvice](https://www.educative.io/courses/guide-spring-5-spring-boot-2/YQz17Bo3mAp#@ControllerAdvice)
* [@ExceptionHandler](https://www.educative.io/courses/guide-spring-5-spring-boot-2/YQz17Bo3mAp#@ExceptionHandler)
* [Generic exception handler](https://www.educative.io/courses/guide-spring-5-spring-boot-2/YQz17Bo3mAp#Generic-exception-handler)

PlayerErrorResponse is an error response class containing information that we want to send to the client. PlayerNotFoundException is a custom exception class. The REST service throws exceptions of this class when a player record is not found in the database.

## @ControllerAdvice

A best practice in exception handling, is to have centralized exception handlers that can be used by all controllers in the REST API. Since exception handling is a cross cutting concern, Spring provides the @ControllerAdvice annotation. This annotation intercepts requests going to the controller and responses coming from controllers.

@ControllerAdvice annotation

The @ControllerAdvice annotation can be used as an interceptor of exceptions thrown by methods annotated with @RequestMapping or any of its shortcut annotations. The exception handling logic is contained in the global exception handler which handles all exceptions thrown by the **PlayerController**.

Custom error response

1. We will create a new class PlayerExceptionHandler, and annotate it with the @ControllerAdvice annotation so it can act as a global exception handler.

@ControllerAdvice  
public class PlayerExceptionHandler {  
  
}

This class will have methods to handle different types of exceptions. We will write a handler method to catch the PlayerNotFoundException exception thrown by the methods in **PlayerService** class. This handler method will create an appropriate response for the client.

1. Create a method playerNotFoundHandler in the PlayerExceptionHandler class. The input to this method is the type of the exception it will handle as well as the HttpServletRequest object. In our case, the exception will be of type PlayerNotFoundException. The method returns a ResponseEntity object containing the HTTP response when the exception occurs. An HTTP response message has three parts: response line, header and body. We can set these attributes in our handler method and configure the HTTP response. The ResponseEntity object is generic and we can send any type as the response body. In our case, the response body will contain an object of the PlayerErrorResponse class.

public ResponseEntity<PlayerErrorResponse> playerNotFoundHandler (  
                                                           PlayerNotFoundException exception,   
                                                           HttpServletRequest req) {  
  
}

## @ExceptionHandler

1. The @ExceptonHandler annotation on a method, marks it as a method that will handle exceptions. Spring automatically checks all methods marked with this annotation when an exception is thrown. If it finds a method whose input type matches the exception thrown, the method will be executed.

@ExceptonHandler  
public ResponseEntity<PlayerErrorResponse> playerNotFoundHandler (  
                                                           PlayerNotFoundException exception,   
                                                           HttpServletRequest req) {  
  
}

1. Inside the handler method, we will create an object of the PlayerErrorResponse class and set its fields, then return it as a ResponseEntity object.

Recall from the last lesson, that the PlayerErrorResponse class had the following fields:

private ZonedDateTime timestamp;  
private int statusCode;  
private String path;  
private String message;

The code below creates a PlayerErrorResponse object called error. This will form the body of the error response.

PlayerErrorResponse error = new PlayerErrorResponse(  
                                              ZonedDateTime.now(),  
                                              HttpStatus.NOT\_FOUND.value(),  
                                              req.getRequestURI(),  
                                              ex.getMessage());

* + To set the current time, we have used the now() function.
  + The HTTP status code of **NOT\_FOUND** is 404. To use the integer value, we have used HttpStatus.NOT\_FOUND.value().
  + We have used the getRequestURI() method on the HttpServletRequest object to get the path at which the exception occurred.
  + The details about the error are contained in the exception and we have used the getmessage() method to extract the message and use it in our response.

In addition to the body of the response, we will also return the appropriate status code with the response. The status code for NOT\_FOUND is 404. The last step is to create and return a ResponseEntity object as follows:

return new ResponseEntity<> (error, HttpStatus.NOT\_FOUND);

The ResponseEntity class provides a variety of constructors to create an object using the status code, header and body or a combination of the three. Here, we have used the constructor which creates a ResponseEntity object with a given body and status code. The other constructor variants for creating the ResponseEntity object are:

* + ResponseEntity(HttpStatus status)
  + ResponseEntity(MultiValueMap<String, String> headers, HttpStatus status)`
  + ResponseEntity(T body, MultiValueMap<String, String> headers, HttpStatus status)`

The complete code of the handler method is shown below:

@ExceptionHandler  
public ResponseEntity<PlayerErrorResponse> playerNotFoundHandler (  
                                                            PlayerNotFoundException ex,   
                                                            HttpServletRequest req) {  
      
    PlayerErrorResponse error = new PlayerErrorResponse(  
                                                  ZonedDateTime.now(),  
                                                  HttpStatus.NOT\_FOUND.value(),  
                                                  req.getRequestURI(),  
                                                  ex.getMessage());  
      
    return new ResponseEntity<> (error, HttpStatus.NOT\_FOUND);  
}

The body of the response will automatically be converted to JSON and sent to the client.

1. We can test the application by sending a GET request to **/players/537**. When the REST Service receives a bad request, it will return a custom JSON response instead of the 500 Internal Server Error. The response header also shows the 404 status code.

###### /

PlayerNotFoundException.java

PlayerExceptionHandler.java

PlayerErrorResponse.java

PlayerController.java

PlayerService.java

PlayerRepository.java

Player.java

TennisPlayerSpringRestApplication.java

**PlayerNotFoundException.java**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

package io.datajek.springrest;

public class PlayerNotFoundException extends RuntimeException{

    public PlayerNotFoundException() {

        super();

    }

    public PlayerNotFoundException(String message, Throwable cause, boolean enableSuppression,

            boolean writableStackTrace) {

        super(message, cause, enableSuppression, writableStackTrace);

    }

    public PlayerNotFoundException(String message, Throwable cause) {

        super(message, cause);

    }

    public PlayerNotFoundException(String message) {

        super(message);

    }

    public PlayerNotFoundException(Throwable cause) {

        super(cause);

    }





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/players/537>

## Generic exception handler

1. It is always a good idea to create a handler to catch all exceptions and send a custom response. We will define another exception handler method called genericHandler. The method signature is the same as the previous handler except for the input type, which is the parent class, Exception (as opposed to PlayerNotFoundException used in the previous handler method).

@ExceptionHandler  
public ResponseEntity<PlayerErrorResponse> genericHandler (  
                                                     Exception ex,   
                                                     HttpServletRequest req){  
  
  
}

Inside the method, we will create a PlayerErrorResponse object which is the body of the response. Since this method is a generic exception handler, it will set the status code to 400 which corresponds to HTTP status code for **BAD\_REQUEST**.

@ExceptionHandler  
public ResponseEntity<PlayerErrorResponse> genericHandler (  
                                                    Exception ex,   
                                                    HttpServletRequest req){  
  
    PlayerErrorResponse error = new PlayerErrorResponse(  
                                               ZonedDateTime.now(),  
                                               HttpStatus.BAD\_REQUEST.value(),  
                                               req.getRequestURI(),  
                                               ex.getMessage());  
      
    return new ResponseEntity<> (error, HttpStatus.BAD\_REQUEST);  
}

The ResponseEntity object is returned with the response body and the Http status of BAD\_REQUEST.

1. The new handler can be tested by sending a GET request to **players/abc**. This handler is also be executed in case of a bad POST request such as:

{  
    "name": "Federer",  
    "nationality": "Switzerland",  
    "birthDate": "22/05/84",  
    "titles": 151  
}

Here, we are sending the birthDate in the wrong format. An exception is also thrown when the body is missing from the POST or PUT request.

###### /

PlayerNotFoundException.java

PlayerExceptionHandler.java

PlayerErrorResponse.java

PlayerController.java

PlayerService.java

PlayerRepository.java

Player.java

TennisPlayerSpringRestApplication.java

**PlayerExceptionHandler.java**

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

import org.springframework.http.ResponseEntity;

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

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

@ControllerAdvice

public class PlayerExceptionHandler {

    @ExceptionHandler

    public ResponseEntity<PlayerErrorResponse> playerNotFoundHandler (PlayerNotFoundException ex, HttpServletRequest req){

        PlayerErrorResponse error = new PlayerErrorResponse(

                                                              ZonedDateTime.now(),

                                                              HttpStatus.NOT\_FOUND.value(),

                                                              req.getRequestURI(),

                                                              ex.getMessage());

        return new ResponseEntity<> (error, HttpStatus.NOT\_FOUND);

    }

    @ExceptionHandler

    public ResponseEntity<PlayerErrorResponse> genericHandler (Exception ex, HttpServletRequest req){

        PlayerErrorResponse error = new PlayerErrorResponse(

                                                              ZonedDateTime.now(),

                                                              HttpStatus.BAD\_REQUEST.value(),

                                                              req.getRequestURI(),

                                                              ex.getMessage());

        return new ResponseEntity<> (error, HttpStatus.BAD\_REQUEST);

    }

}





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/players>

# Spring Data REST

We can quickly create a REST API with full CRUD functionality without writing any controller or service layer code using Spring Data REST.

**We'll cover the following**

* [Add dependency](https://www.educative.io/courses/guide-spring-5-spring-boot-2/YMJ6jzBrGGW#Add-dependency)
* [Define entity](https://www.educative.io/courses/guide-spring-5-spring-boot-2/YMJ6jzBrGGW#Define-entity)
* [Create repository](https://www.educative.io/courses/guide-spring-5-spring-boot-2/YMJ6jzBrGGW#Create-repository)
* [HATEOAS](https://www.educative.io/courses/guide-spring-5-spring-boot-2/YMJ6jzBrGGW#HATEOAS)
* [Customization](https://www.educative.io/courses/guide-spring-5-spring-boot-2/YMJ6jzBrGGW#Customization)
  + [Base path](https://www.educative.io/courses/guide-spring-5-spring-boot-2/YMJ6jzBrGGW#Base-path)
  + [Sorting](https://www.educative.io/courses/guide-spring-5-spring-boot-2/YMJ6jzBrGGW#Sorting)
  + [Paging](https://www.educative.io/courses/guide-spring-5-spring-boot-2/YMJ6jzBrGGW#Paging)
  + [Resource name](https://www.educative.io/courses/guide-spring-5-spring-boot-2/YMJ6jzBrGGW#Resource-name)

Spring Data REST

Spring Data REST is a project similar to Spring Data JPA which aims to eliminate boilerplate code. With Spring Data JPA, we get the basic CRUD functionality without writing any code simply by specifying the entity and the type of primary key.

Similarly, Spring Data REST provides a REST API based on the repository and entity without us having to write any code in the controller and service layer. It uses the repository to expose endpoints to perform GET, POST, PUT, PATCH and DELETE on every entity in the application. Spring Data REST works with data sources implementing the repository programming model. It supports Spring Data JPA, Spring Data MongoDB, Spring Data Cassandra as well as other Spring Data projects.

Reduction in boilerplate code

Spring Data REST provides a basic REST API which can be customized. Custom queries can be added using JPQL and Query DSL.

Earlier, we saw how to implement a Tennis Player REST API. We had a Player entity and created a PlayerRepository implementing the JpaRespository interface. Spring Data REST can provide us a similar API simply by scanning our repository and exposing the **/players** endpoint.

Spring Data REST creates endpoints using the entity name by making the first letter lowercase and adding an ‘s’ to the end of the name. For example, we have the following repository interface:

public interface PlayerRepository extends JpaRepository<Player, Integer> {   
  
}

Spring Data REST will convert the entity name **Player** to its uncapitalized, pluralized form **players** and expose the REST endpoints at **/players**. It also exposes **/players/{id}** for each item managed by the repository.

To create a Spring Data REST application using Spring Boot, we need the following:

* spring-boot-starter-data-rest dependency in **pom.xml**
* An entity (e.g., Player)
* A repository interface (e.g., JpaRepository or CrudRepository)

## Add dependency

1. Create a Spring Boot application using the [Spring Initializr](https://www.educative.io/courses/guide-spring-5-spring-boot-2/start.spring.io). Provide **io.datajek** as the group id and **tennis-player-spring-data-rest** as the artifact id. Then, add the following dependencies:
   * Spring Data JPA spring-boot-starter-data-jpa
   * REST Repositories spring-boot-starter-data-rest
   * H2 Database h2
   * Spring Boot Devtools spring-boot-devtools

To add the Spring Data Rest functionality to an existing project, add the following dependency to the **pom.xml** file:

<dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-data-rest</artifactId>  
</dependency>

## Define entity

1. Spring Data REST scans the repositories to create the endpoints based on the entities defined. We will define an entity class **Player** and use the @Entity annotation so it will be marked by the JPA. The **Player** entity has different fields. The id is the primary key and marked by the @Id annotation as well as the @GeneratedValue annotation, in which we provide the primary key generation strategy in parenthesis.

@Entity  
public class Player {  
    @Id  
    @GeneratedValue(strategy=GenerationType.IDENTITY)  
    private int id;  
    private String name;     
    private String nationality;  
    @JsonFormat(pattern = "dd-MM-yyyy")  
    private Date birthDate;  
    private int titles;  
    //constructors  
    //getters and setters  
}

We will generate getters and setters as well as constructors for the POJO class. The in-memory H2 database can be populated using a script placed in the **/src/main/resources** folder. This script is called **import.sql**, and has the following queries:

INSERT INTO player (ID, Name, Nationality, Birth\_date, Titles) VALUES(1,'Djokovic', 'Serbia', '1987-05-22', 81);  
INSERT INTO player (ID, Name, Nationality, Birth\_date, Titles) VALUES(2,'Monfils', 'France', '1986-09-01', 10);  
INSERT INTO player (ID, Name, Nationality, Birth\_date, Titles) VALUES(3,'Isner', 'USA', '1985-04-26', 15);

On a local machine, the web console of the H2 database is accessed at [**http://localhost:8080/h2-console**](http://localhost:8080/h2-console). We will set the following properties in the **application.properties** file in **src/main/resources**:

spring.h2-console.enabled=true  
spring.datasource.url=jdbc:h2:mem:testdb

## Create repository

1. Lastly, we will create a repository called PlayerRepository by extending the JpaRepository interface as follows:

public interface PlayerRepository extends JpaRepository<Player, Integer>{  
}

The entity Player and type of primary key Integer are specified in brackets.

1. At this point, we have the Maven dependency of Spring Data REST, the entity and repository and we are ready to test our REST API without writing any controller or service classes. The application is available at the link provided under the widget (or at [**http://localhost:8080/players**](http://localhost:8080/players) on a local machine) and displays the JSON data of the players in the database.

###### /

PlayerRespository.java

Player.java

TennisPlayerSpringDataRestApplication.java

**PlayerRespository.java**

1

2

3

4

5

6

7

8

package io.datajek.tennisplayerspringdatarest;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.data.rest.core.annotation.RepositoryRestResource;

public interface PlayerRepository extends JpaRepository<Player, Integer> {

}





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/players>

## HATEOAS

1. Spring Data REST uses the HATEOAS application architecture and provides metadata in the response including the information about the current page, as well as the total number of pages and the number of records per page. This is different from the REST API we created in the previous lessons. HATEOAS stands for **H**ypermedia **A**s **T**he **E**ngine **O**f **A**pplication **S**tate which basically is metadata about the REST data. The server includes hypermedia links in the response and the client can navigate those links to the data. For a collection of items, in addition to the JSON array of items, the metadata information includes the page size and total number of elements etc. This can be seen in the response to **GET** request to **/players**:

HATEOAS data

* Each item also has a link which can be used to get details of an individual player. If we click the link for player 2, a new tab with a GET request to **/players/2** opens up which shows the JSON response with details of player with id 2, as well as the link for the player.

Using Postman, we can add a new player by sending the following JSON data in the request body:

{  
    "name": "Federer",  
    "nationality": "Switzerland",  
    "birthDate": "22-05-1984",  
    "titles": 151  
}

When we send a **POST** request to **/players**, the HATEOAS response shows the hyperlink to the newly created resource (**/players/4**).

To update a player, send the following **PUT** request to **/players/4** with the following JSON data:

{  
    "name": "Federer",  
    "nationality": "Switzerland",  
    "birthDate": "22-11-1984",  
    "titles": 161  
}

For partial updates, we can send a **PATCH** request to **/players/1** and update the titles to 157.

{  
    "titles": 157  
}

To delete a player, we will send the player id in the request such as **/players/4**. On success, Spring Data REST response contains status code 204 which stands for no content found.

## Customization

1. Spring Data REST allows for customization of the REST API through a number of properties. It is possible to change the path of the API using the basePath property. Similarly, we can choose how many items to display per page using the defaultPageSize and maxPageSize properties. The returnBodyOnCreate and returnBodyOnUpdate properties can be used to change the response after a **POST** and **PUT** request. We can also specify which repositories are exposed in the REST API by using the detectionStrategy property. We will explore some of the properties in the following steps.

### Base path

1. We can change the base path of our API. To do so, open the **application.properties** file and add the following property:

spring.data.rest.basePath=/api

With this change, the application is available at **/api/players** and accessing it at **/players** will result in a 404 error.

### Sorting

1. We can sort the results returned by the field names of the entity using the **sort** property in the request URL. The Player entity had the id, name, nationality, brithDate and titles fields. We can sort the players based on their date of birth by sending a **GET** request to **/api/players?sort=birthDate**. The default sort order is ascending.

To sort in descending order, we need to specify the desc keyword. A **GET** request to **/api/players?sort=titles,desc** will sort the players with the most number of titles on top.

We can also sort by multiple fields, say, nationality and name as **/api/players?sort=nationality,name**.

### Paging

1. By default, Spring Data REST returns 20 records per page. If there are more than 20 records, they are moved to the next page. This default behavior can be changed using properties.

When we send a **GET** request to **/api/players**, we get a JSON response of all players and at the bottom, we get the meta-data about the page, which shows that the number of items per page, or size is 20. The metadata also contains information about the total number of elements, and the total number of pages in the response, as well as the current page number. The image below shows the metadata information:

Metadata

* We will use the defaultPageSize property and set the items to be displayed per page to 2. That way, we will get a multi page response to the **/players** request.

spring.data.rest.defaultPageSize = 2

Now, the response will be split in 2 pages, with page 0 being the first page and page 1 being the second page. The metadata information is shown below:

Changing default values

* In a multi-page response, the metadata also contains links to the **first** and **last** pages as well as the previous (**prev**) and **next** pages of the result. There is no **prev** link for the first page and no **next** link for the last page of the response. We can navigate to the different pages using the links provided in the response. The page numbering starts at 0. Our database has 3 players and the response contains 2 pages. The second page can be accessed at **/players/?page=1**.

The page size can also be modified by using the **size** parameter in the query string. Suppose we want to show 3 elements per page. We can send a request to **/api/players/?size=3**.

### Resource name

1. Spring Data Rest uses the up-capitalized, pluralized form of the entity name as the resource name for the endpoints. From the Player entity it created the **/players** endpoint. If we want to change the resource name, we can do so by using the @RepositoryRestResource annotation on the repository and provide the desired resource name as path:

@RepositoryRestResource(path="athletes")  
public interface PlayerRepository extends JpaRepository<Player, Integer> {  
  
}

Spring Data REST will now expose the **/api/athletes** endpoint instead of **/api/players**.

###### /

application.properties

PlayerRespository.java

Player.java

TennisPlayerSpringDataRestApplication.java

**application.properties**

1

2

3

4

5

spring.h2-console.enabled=true

spring.datasource.url=jdbc:h2:mem:testdb

#spring.data.rest.basePath=/api

#spring.data.rest.defaultPageSize=2

#spring.data.rest.returnBodyOnUpdate=false





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/players>

# Database Relationships in Spring

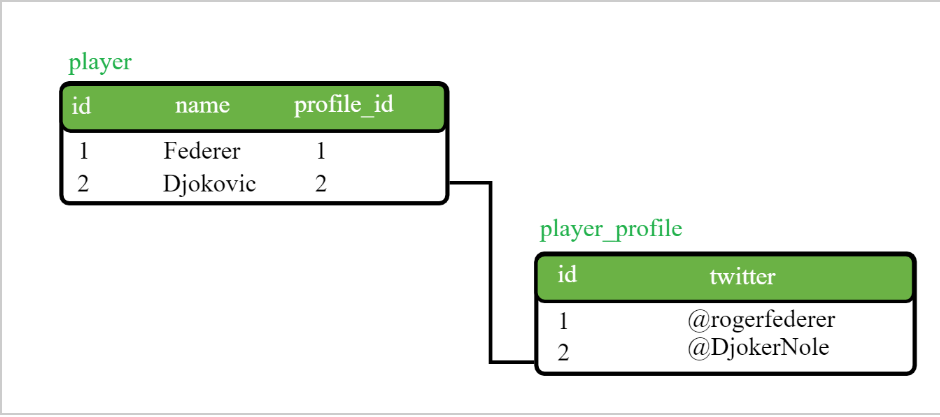
# Basic Concepts

**We'll cover the following**

* [Primary key](https://www.educative.io/courses/guide-spring-5-spring-boot-2/xoWJxR8zYxE#Primary-key)
* [Foreign key](https://www.educative.io/courses/guide-spring-5-spring-boot-2/xoWJxR8zYxE#Foreign-key)
* [Database relationships](https://www.educative.io/courses/guide-spring-5-spring-boot-2/xoWJxR8zYxE#Database-relationships)
  + [One to One relationship](https://www.educative.io/courses/guide-spring-5-spring-boot-2/xoWJxR8zYxE#One-to-One-relationship)
  + [One to Many relationship](https://www.educative.io/courses/guide-spring-5-spring-boot-2/xoWJxR8zYxE#One-to-Many-relationship)
  + [Many to Many relationship](https://www.educative.io/courses/guide-spring-5-spring-boot-2/xoWJxR8zYxE#Many-to-Many-relationship)
* [Referential integrity](https://www.educative.io/courses/guide-spring-5-spring-boot-2/xoWJxR8zYxE#Referential-integrity)
* [Cascading](https://www.educative.io/courses/guide-spring-5-spring-boot-2/xoWJxR8zYxE#Cascading)
* [Fetch types](https://www.educative.io/courses/guide-spring-5-spring-boot-2/xoWJxR8zYxE#Fetch-types)
* [Orphan records](https://www.educative.io/courses/guide-spring-5-spring-boot-2/xoWJxR8zYxE#Orphan-records)

## Primary key

Primary key is used to uniquely identify a row in a table. The id column in the **player** table shown below acts as a unique identifier for all records. No two records can have the same primary key value.



## Foreign key

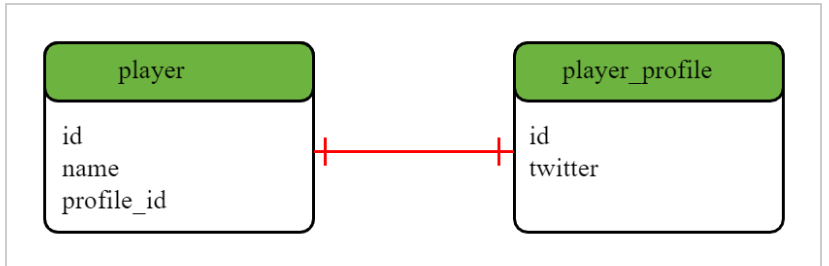
Primary key can be used to link two tables together. When a primary key of one table is used in another table, it is known as a foreign key. To link the **player** and **player\_profile** tables, the id column from the **player\_profile** table is placed in the **player** table. The column profile\_id is called the foreign key column and is used to point to the record in the **player\_profile** table that is linked to the record in the **player** table.

## Database relationships

The tables in a database are linked in different ways.

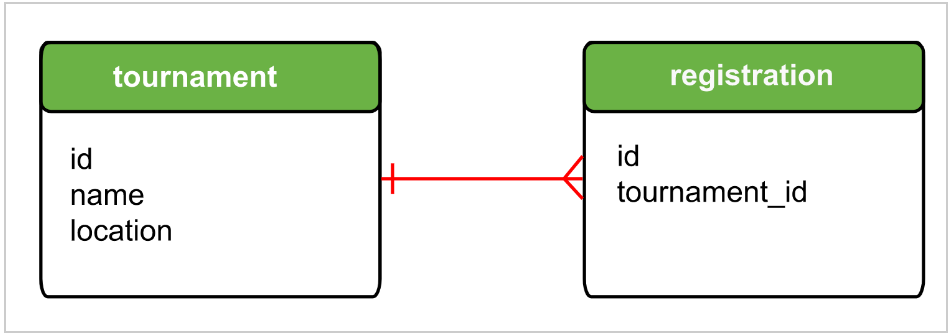
### One to One relationship

When one row in table\_a corresponds to one row in table\_b, then it is called a one to one relationship. Take the example of **player** and **player\_profile** tables. One player has one player profile entry so there is a one-to-one relationship between the tables.



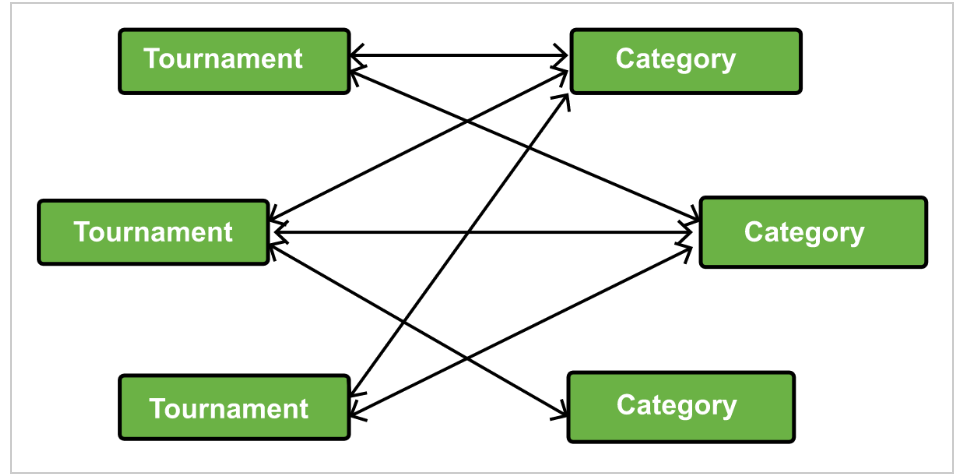
### One to Many relationship

When one row in table\_a corresponds to many rows in table\_b, then it is called a one-to-many relationship. The inverse of a one-to-many relationship is a many-to-one relationship where many rows in table\_b correspond to one row in table\_a. For example, a player can register for many tournaments. There is a one-to-many relationship between the **player** table and the **registration** table.



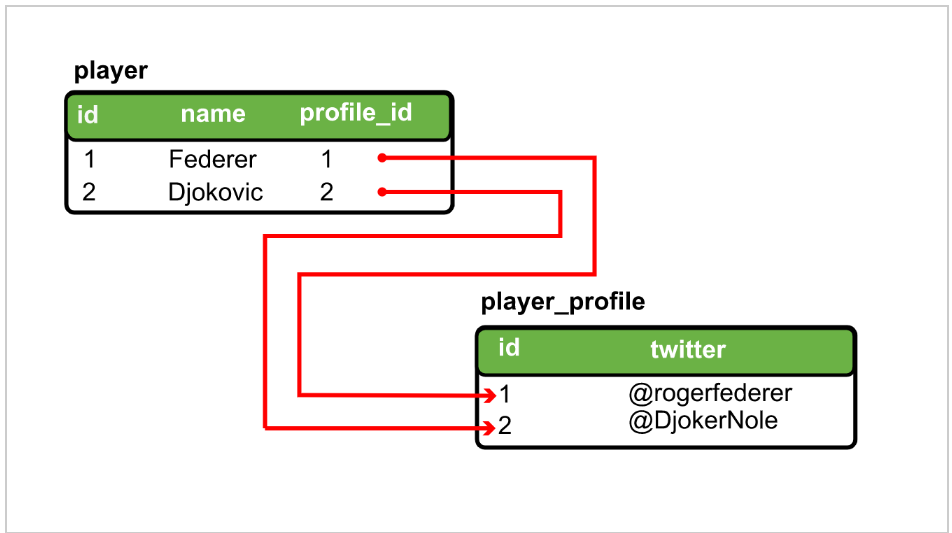
### Many to Many relationship

When one row in table\_a corresponds to many rows in table\_b and one row in table\_b corresponds to many rows in table\_a, then we have what is called a many-to-many relationship. Take the example of tournaments and playing categories (like singles, doubles etc.). One tournament features one than one playing category. In the same way, one playing category is part of many tournaments. There is a many-to-many relationship between the tournament and category tables.



## Referential integrity

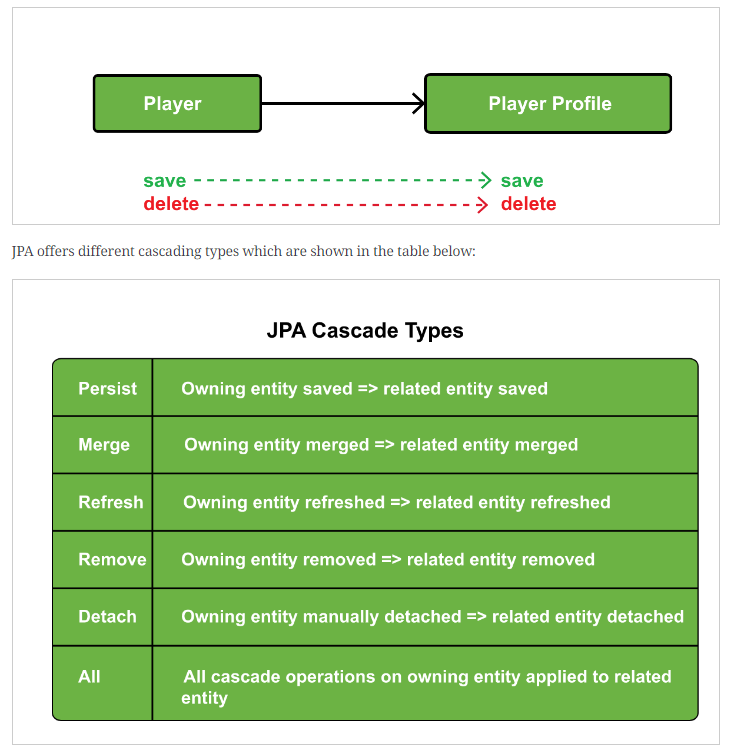
When two tables are related to one another, data should reflect a consistent state. All foreign keys should refer to valid, existing primary key values. For example, a profile belongs to a player so the player must exist. If a player record is deleted, then the corresponding record in the player profile table should also be deleted. Absence of referential integrity can lead to records being lost in the database.



## Cascading

Cascading means propagating an operation from the owning entity to the related entity. When a record in the owning entity (parent table) is saved, updated or deleted, the change should be reflected in the related entity (child table) as well.

If we save a Player object containing a nested Profile object, the save operation is cascaded from the **player** table to the **player\_profile** table and two records are inserted in the database.

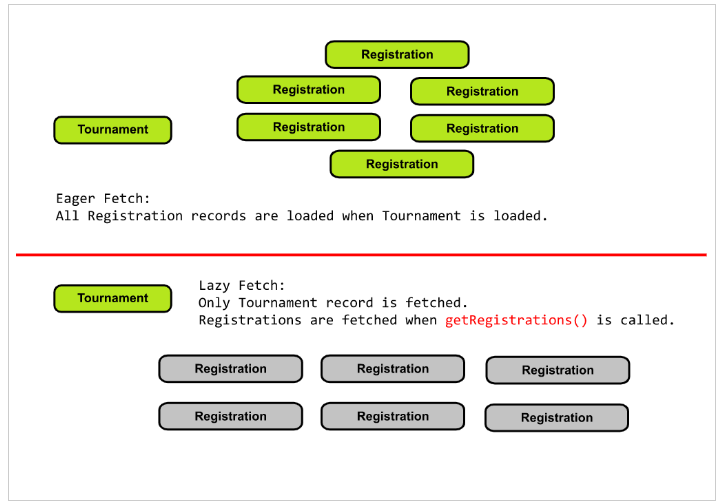


JPA offers different cascading types which are shown in the table below:

## Fetch types

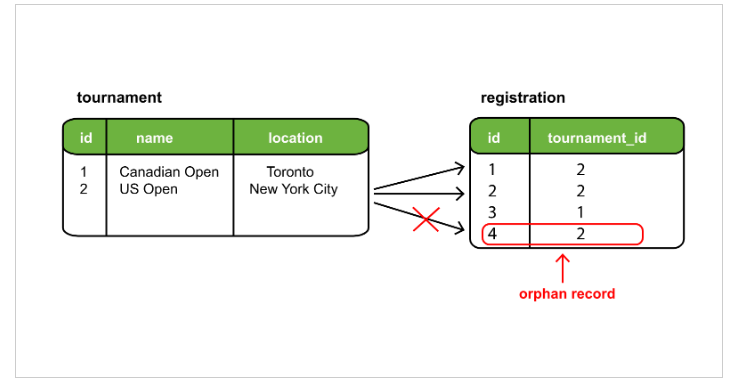
There are two ways in which data is loaded: eager and lazy. Eager fetch means that when a record is fetched from the database, all the associated records from related tables are also fetched. So if we fetch a tournament record, all the registrations for the tournament are also fetched.

Eager fetch is the default fetch type used by Hibernate but it is not always the most efficient. Lazy fetch on the other hand, fetches the records only when they are needed.



## Orphan records

When we remove the relationship between a parent and child, the child record becomes an orphan record meaning that it does not have a parent record. Consider the example of tournament and registration tables where a player withdraws from a tournament. The registration gets removed from the tournament entity. The registration record becomes an orphan record. Orphan records mean that the database is in an inconsistent state.



# One-to-One Unidirectional Relationship

We will learn how to model a unidirectional 1-1 relationship in this lesson.

**We'll cover the following**

* [Project Creation](https://www.educative.io/courses/guide-spring-5-spring-boot-2/N0RJ2PnMKk2#Project-Creation)
  + [Dependencies:](https://www.educative.io/courses/guide-spring-5-spring-boot-2/N0RJ2PnMKk2#Dependencies:)
  + [Database Configuration](https://www.educative.io/courses/guide-spring-5-spring-boot-2/N0RJ2PnMKk2#Database-Configuration)
* [One-to-One Relationship](https://www.educative.io/courses/guide-spring-5-spring-boot-2/N0RJ2PnMKk2#One-to-One-Relationship)
  + [cascade property](https://www.educative.io/courses/guide-spring-5-spring-boot-2/N0RJ2PnMKk2#cascade-property)
  + [@JoinColumn](https://www.educative.io/courses/guide-spring-5-spring-boot-2/N0RJ2PnMKk2#@JoinColumn)
* [Persisting entities](https://www.educative.io/courses/guide-spring-5-spring-boot-2/N0RJ2PnMKk2#Persisting-entities)
* [Hibernate implementation of @OneToOne](https://www.educative.io/courses/guide-spring-5-spring-boot-2/N0RJ2PnMKk2#Hibernate-implementation-of-@OneToOne)

To model a one-to-one relationship, consider the scenario where a player has a profile which stores his details. We have a **player** table which stores the basic information about the player like id and name and a **player\_profile** table which stores the additional details like the social media accounts of the players. There is a one-to-one relationship between player and player-profile tables and in this lesson, we will model it as a one way/ unidirectional relationship.

Player has a one-to-one relationship with PlayerProfile

## Project Creation

We will create a project for understanding database relationships by going to [Spring Initializr](http://start.spring.io/) and use the following values for group Id and artifact Id:

* group id: **io.datajek**
* artifact id: **database-relationships**

### Dependencies:

For this project, the following dependencies are needed:

* Spring’s JPA implementation spring-boot-starter-jpa which uses Hibernate ORM framework.
* spring-boot-starter-web dependency which supports web applications as well as REST services.
* In-memory H2 database dependency.
* Spring Boot devtools dependency for auto restart functionality.

Once the project has been imported in the IDE, we will add the Jackson dependency for Hibernate 5 as follows:

<dependency>  
    <groupId>com.fasterxml.jackson.datatype</groupId>  
    <artifactId>jackson-datatype-hibernate5</artifactId>  
    <version>2.13.2</version>  
</dependency>

This dependency provides support for Hibernate datatypes and specifically handles aspects of lazy-loading.

### Database Configuration

Since we have used Spring Boot to create the provide, the datasource has been automatically configured. We will add the data source URL in the **application.properties** file. In addition, we will also enable the web console of the database.

spring.datasource.url = jdbc:h2:mem:testdb  
spring.h2.console.enabled = true

We can also enable the show-sql property to displays the SQL queries executed by Hibernate as follows:

spring.jpa.show-sql = true

1. We will begin by creating a package called **onetoone**. Create a Player class and mark it as an entity using @Entity annotation. We will only create id and player name fields at the moment to keep this example simple.

package io.datajek.databaserelationships.onetoone;  
  
@Entity  
public class Player {  
  private int Id;  
  private String name;  
}

Since Id is the primary key, we will mark it with @Id annotation and let Hibernate generate values for this column using the @GeneratedValue annotation and a GenerationType of IDENTITY.

@Entity  
public class Player {  
  @Id  
  @GeneratedValue(Strategy = GenerationType.IDENTITY)  
  private int Id;  
  private String name;  
  //...  
}

Hibernate requires a default constructor. We will also create an overloaded constructor to create the object using fields, getters and setters for the fields, and a toString method. The full code of the Player class is shown in the code widget at the end of the lesson.

1. Next, we will create the PlayerProfile class in the **onetoone** package to hold miscellaneous information about a player and also mark it as an entity. For now, this class will only store the player’s Twitter account handle.

package io.datajek.databaserelationships.onetoone;  
  
@Entity  
public class PlayerProfile {  
  private int Id;  
  private String twitter;  
}

We will mark the primary key with @Id annotation and generate the constructors, getters, setters and ToString method as we did for the Player class.

@Entity  
public class PlayerProfile {  
  @Id  
  @GeneratedValue(Strategy = GenerationType.IDENTITY)  
  private int Id;  
  private String twitter;  
  
  //constructors  
  //getters and setters  
}

1. If we run the application now, and visit the H2 console at <http://localhost:8080/h2-console> (and use jdbc:h2:mem:testdb as the connection URL), we can see that two tables **player** and **player\_profile** have been created. Right now, there is no relationship between the two tables.

## One-to-One Relationship

1. In SQL, we represent relationships using primary key and foreign key. Foreign keys are used to link tables together. A foreign key is a field in one table that refers to the primary key in another table. We will see how the same can be achieved using Hibernate (which is the default implementation of Spring Data JPA).

For the example taken in this lesson, the Player class has a one-to-one relationship with the PlayerProfile class. To show this relationship, we will add a field playerProfile to the Player class and use the JPA annotation @OneToOne on this field.

@Entity  
public class Player {  
  @Id  
  @GeneratedValue(Strategy = GenerationType.IDENTITY)  
  private int Id;  
  
  private String name;  
  
  @OneToOne  
  private PlayerProfile playerProfile;  
  
}

@OneToOne is a JPA annotation which maps a source entity to a target entity.

### cascade property

1. The cascade property ensures that changes made to a Player entity are reflected in the PlayerProfile entity. The PlayerProfile entity does not have a meaning of its own, rather, it defines the Player entity. If we delete a Player entity, the associated details should also be deleted. Cascading allows an operation on the Player entity to be propagated to the PlayerProfile entity.

@OneToOne(cascade= CascadeType.ALL)  
private PlayerProfile playerProfile;

We have set the CascadeType to ALL. This means that all JPA and Hibernate specific operations on the Player entity will be propagated to the PlayerProfile entity.

The absence of the cascade property, results in the TransientPropertyValueException exception when Hibernate tries to save a Player object containing a nested PlayerProfile object.

### @JoinColumn

1. In relationships, one side is the owning side. We use the @JoinColumn annotation on the owning side. Here, the Player class is the owning side of the relationship. The @JoinColumn annotation specifies the name of the foreign key column in the **player** table. We will call the column **profile\_id**. If the name is not specified, then JPA names the column based on some rules. In the **player\_profile** table, the column that is being referenced is **id**. The name of the corresponding field in the PlayerProfile class is Id which we specify as referencedColumnName.

@Entity  
public class Player {  
  @Id  
  @GeneratedValue(Strategy = GenerationType.IDENTITY)  
  private int id;  
  
  private String name;  
  
  @OneToOne(cascade= CascadeType.ALL)  
  @JoinColumn(name="profile\_id", referencedColumnName="id")  
  private PlayerProfile playerProfile;  
  
}

Create a getter and setter for the new field and update the ToString method.

@JoinColumn annotation

1. After adding the @OneToOne annotation, if we run the application and check the H2 database, we can see that the **player** table has changed. It now contains a **profile\_Id** column which references the **id** column in the **player\_profile** table. The **player** table is called the owning table because here we store the foreign key.

Table structure

It is a unidirectional relationship because we have the reference of the PlayerProfile entity in the Player entity but we don’t have any reference of the Player entity in the PlayerProfile entity. We can retrive the PlayerProfile object using the Player object but not the other way round.

Unidirectional relationship

## Persisting entities

1. Next, we will create repositories for both classes, Player and PlayerProfile, that extend the JpaRepository interface. Since JpaRepository is a generic type, we need to specify the type of object as well as the datatype of the primary key.

We will create a package named repository and create two interfaces named PlayerRepository and PlayerProfileRepository and annotate them with @Repository to leverage the exception translation facility offered by Spring.

package io.datajek.databaserelationships.onetoone;  
  
@Repository  
public interface PlayerRepository extends JpaRepository <Player, Integer> {  
}

package io.datajek.databaserelationships.onetoone;  
  
@Repository  
public interface PlayerProfileRepository extends JpaRepository <PlayerProfile, Integer> {  
}

1. After creating repositories, we will create classes in the service layer to perform the CRUD operations on the entities. First, create a package called service for all the service layer classes. The PlayerService class is shown below. The PlayerProfileService class is also defined in the same manner. The complete code of both the classes can be seen in the code widget at the end of the lesson.

package io.datajek.databaserelationships.onetoone;  
  
@Service  
public class PlayerService {  
  
    @Autowired  
    PlayerRepository repo;  
      
    public List<Player> allPlayers() {  
        return repo.findAll();        
    }  
  
    public Player addPlayer(Player player) {  
        player.setId(0);  
        return repo.save(player);  
    }  
  
     //...  
}

1. We will create controller classes and use Postman to create and persist entities. First, create a package named controller for all the controller classes of the project. Then, create two classes PlayerController and PlayerProfileController in the newly created package.

Use the @RestController annotation on the classes to create a RESTful controller. The @RequestMapping annotation on class level creates a base mapping for the methods in the class. We will use the **\players** mapping for the PlayerController class and **\profiles** mapping for the PlayerProfileController class.

The controllers contain methods to map GET, POST, and DELETE requests.

package io.datajek.databaserelationships.onetoone;  
  
@RestController  
@RequestMapping("/players")  
public class PlayerController {  
  
    @Autowired  
    PlayerService service;  
  
    //GET all players   
    //GET player by Id  
    //POST player   
    //DELETE player   
}

package io.datajek.databaserelationships.onetoone;  
  
@RestController  
@RequestMapping("/profiles")  
public class PlayerProfileController {  
  
    @Autowired  
    PlayerProfileService service;  
   
    //GET all player profiles  
    //GET player profile by Id  
    //POST player profile  
    //DELETE player profile  
}

If we run the application and send a GET request to <http://localhost:8080/players/> or <http://localhost:8080/profiles/> we get **[]** as response since the tables are empty at the moment.

###### /

DatabaseRelationshipsApplication.java

Player.java

PlayerProfile.java

PlayerRepository.java

PlayerProfileRepository.java

PlayerService.java

PlayerProfileService.java

PlayerController.java

PlayerProfileController.java

**PlayerProfile.java**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

package io.datajek.databaserelationships.onetoone;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

@Entity

public class PlayerProfile {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private int id;

    private String twitter;

    public PlayerProfile() {

    }

    public PlayerProfile(String twitter) {

        super();

        this.twitter = twitter;

    }

    public int getId() {

        return id;

    }

    public void setId(int id) {

        this.id = id;

    }





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/players>

For the code widget given above, use the URL at which the application is running in place of <http://localhost:8080/>. For example, **/players** means <http://localhost8080/players> for local dev environment. If using POSTMAN with code widget above, use the URL shown under the code widget to access **/players**.

1. We can create a Player entity by sending a POST request to **/players** with the following request body:

{  
    "name": "Federer"  
}

From the response, it can be seen that Hibernate assigns the player an id of 1.

Next we will create a PlayerProfile entity by sending a POST request to **/profiles**.

{  
    "twitter" : "@rogerfederer"  
}

Right now, both entities are not connected. To assign the PlayerProfile to Player, we need to create a **PUT** mapping in the PlayerController class. This will enable us to send a request to **/players/{id}/profiles/{profile\_id}** and update the Player entity. Create a method named assignDetail in the PlayerController as follows:

@PutMapping("/{id}/profiles/{profile\_id}")  
public Player assignDetail(@PathVariable int id, @PathVariable int profile\_id) {  
    PlayerProfile profile = profileService.getPlayerProfile(profile\_id);   
    return service.assignProfile(id, profile);  
}

Using the profile\_id, we retrieve the PlayerProfile entity and then pass it to the assignProfile method in the PlayerService class.

The assignProfile method in the PlayerService class is responsible for updating a Player record. It simply uses the setter method for the playerProfile field and then calls save to update the record in the database.

public Player assignProfile(int id, PlayerProfile profile) {  
    Player player = repo.findById(id).get();  
    player.setPlayerProfile(profile);  
    return repo.save(player);  
}

With the above methods in place, we can send a **PUT** request to **/players/1/profiles/1**. This request will update the Player with id 1 and assign the PlayerProfile object with id 1 to it.

A **GET** request to **/players** shows that the relationship has been established.

GET request to /players

1. We can also create a Player object with a nested PlayerProfile object as follows:

{  
    "name": "Djokovic",  
    "playerProfile": {  
        "twitter" : "@DjokerNole"  
    }  
}

This will not only insert a row in the **player** table but also insert a corresponding row in the **player\_profile** table. Hibernate fires two INSERT queries because we have set the CascadeType to ALL. This ensures that changes to the **player** table are propagated to the **player\_detail** table.

Save is cascaded from player to player\_profile

We can run the application and check the database tables in the H2 console (at **/h2-console** with **jdbc:h2:mem:testdb** as datasource URL) to verify that the **player** record has the correct foreign key for the **player\_profile** record.

This is an example of a unidirectional one-to-one relationship. It is possible to retrieve a PlayerProfile object using a Player object but no way to retrieve a Player object using a PlayerProfile object as can be seen from **GET** request to **/players** and **/profiles**.

GET request to /players and /profiles

The above results can be verified from the web console of H2 database by visiting **/h2-console** with jdbc:h2:mem:testdb as the connection URL.

## Hibernate implementation of @OneToOne

Hibernate supports three variations of the @OneToOne mapping.

* Using foreign key with the @JoinColumn annotation.
* Using a common join table which has foreign keys of both tables. The @JoinTable annotation defines a new table name which has the foreign key from both tables. This helps in modelling optional one-to-one relationships. If a player does not have a PlayerProfile entry, we have to use null value in that column.
* Using a shared primary key to save space. This approach uses a common primary key (player\_id in this case) in both tables using the @PrimaryKeyJoinColumn. It eliminates the need of having an Id column for the **player\_profile** table.

The figure below illustrates the three ways in which @OneToOne annotation can be used.

# 

# One-to-One Bidirectional Relationship

Learn how to make the one-to-one relationship bidirectional. This lesson also shows how to resolve the JSON infinite recursion issue.

**We'll cover the following**

* [Bi-directional relationship](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7nqXk4gP8mr#Bi-directional-relationship)
  + [mappedByattribute](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7nqXk4gP8mr#mappedByattribute)
* [JSON Infinite Recursion](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7nqXk4gP8mr#JSON-Infinite-Recursion)
  + [Solution 1: @JsonManagedReference and @JsonBackReference](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7nqXk4gP8mr#Solution-1:-@JsonManagedReference-and-@JsonBackReference)
  + [Solution 2: @JsonIdentityInfo](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7nqXk4gP8mr#Solution-2:-@JsonIdentityInfo)
* [Cascade Type](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7nqXk4gP8mr#Cascade-Type)
* [optional attribute:](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7nqXk4gP8mr#optional-attribute:)
* [Pros and cons of bidirectional relationship](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7nqXk4gP8mr#Pros-and-cons-of-bidirectional-relationship)

Right now, we have a unidirectional one-to-one mapping which means that if we have a Twitter account, we cannot find the name of the player who has that account. A **GET** request to **/profiles** only gets the PlayerProfile object and not the Player it is associated with.

It is however possible to find the Twitter account, if we have the Player entity. As can be seen from a **GET** request to **/players**, the PlayerProfile entities are also fetched.

GET request to /players and /profiles

In the unidirectional one-to-one relationship, the Player class maintains the relationship. The PlayerProfile class cannot see any change in the relationship. To make this relationship bidirectional, we need to make some modifications to the PlayerProfile class. We will add a field to reference back to the Player class and add the @OneToOne annotation. We will also add getter and setter methods to set the Player value in the PlayerProfile class. This will enable us to fetch the entities in both directions.

Bidirectional relationship

## Bi-directional relationship

1. To set up a bidirectional relationship, we will add a field of Player class in the PlayerProfile class and add getter and setter methods for the field. This field holds the reference to the associated Player entity.

package io.datajek.databaserelationships.onetoone;  
  
@Entity  
public class PlayerProfile {  
    @Id  
    @GeneratedValue(GenertionType.IDENTITY)  
    private int id;  
    private String twitter;  
  
    private Player player;  
    //add getter and setter for player field.  
  
}

The toString method also needs to be updated to include the newly added player field.

1. Next, we will add the @OneToOne annotation on the player field.

### mappedByattribute

mappedBy is an optional attribute of the @oneToOne annotation which specifies the name of the field which owns the relationship. In our case, it is the playerProfile field in the Player class. The mappedBy attribute is placed on the inverse side on the relationship only. The owning side cannot have this attribute.

@Entity  
public class PlayerProfile {  
  @Id  
  @GeneratedValue(GenertionType.IDENTITY)  
  private int id;  
   
  private String twitter;  
   
  @OneToOne(mappedBy="playerProfile")  
  private Player player;  
  
}

1. If we visit the web console of the H2 database (at <http://localhost:8080/h2-console> with jdbc:h2:mem:testdb as the connection URL), we will see that there is no change in the database structure. The primary key of the **player\_profile** table is stored as the foreign key in the **player** table as shown in the Entity Relationship Diagram below. On the Java side however, we can now access the Player entity using the PlayerProfile entity.

Table structure

1. To test the bidirectional relationship, we will create a Player with a nested PlayerProfile object using the following JSON **POST** request to <http://localhost8080/players> :

{  
    "name": "Djokovic",  
    "playerProfile": {  
        "twitter" : "@DjokerNole"  
    }  
}

This request results in two INSERT queries in the **player** and **player\_profile** tables respectively.

Bidirectional relationship means that Djokovic owns the @DjokerNole Twitter account and vice versa, the @DjokerNole account belongs to Djokovic. If we now send a **GET** request to **/profiles** we will get the player details as well.

GET request to /profiles in bidirectional relationship

## JSON Infinite Recursion

1. If you are following along in your local dev environment, your must have encountered an error when trying to access **/profiles**.

When using a bidirectional relationship, JSON throws an infinite recursion error when we try to retrieve the objects. This is because the Player object contains the reference of PlayerProfile object and the PlayerProfile object also contains the reference to the Player object.

Infinite recursion issue in bidirectional relationships

The response in Postman looks like this:

Infinite recursion error on /players

### Solution 1: @JsonManagedReference and @JsonBackReference

To solve this issue, we can use the @JsonManagedReference and @JsonBackReference annotations in the classes. As a result, only the owning side of the relationship is serialized and the inverse side is not serialized.

The @JsonManagedReference annotation is used on the playerProfile field in the owning side (Player class). On the inverse side (PlayerProfile class), the @JsonBackReference annotation is used to the player field. These annotations solve the infinite recursion problem.

public class Player{  
    //...  
    @OneToOne(cascade=CascadeType.ALL)  
    @JoinColumn(name = "profile\_id", referencedColumnName = "id")  
    @JsonManagedReference  
    private PlayerProfile playerProfile;  
    //...  
}

public class PlayerProfile{  
    //...  
    @OneToOne(mappedBy= "playerProfile")  
    @JsonBackReference  
    private Player player;  
    //...  
}

### Solution 2: @JsonIdentityInfo

Another solution is to use the @JsonIdentityInfo annotation at class level. Both Player and PlayerProfile classes are annotated with @JsonIdentityInfo to avoid infinite recursion while converting POJOs to String.

@JsonIdentityInfo(generator= ObjectIdGenerators.PropertyGenerator.class, property="id")

The property attribute specifies the property name of the target reference. Here, id field is used to break out of the recursion. The first time id is encountered, it is replaced with the object and for subsequent occurrences of id, the numerical value is used instead of replacing it with the object.

## Cascade Type

1. For the bidirectional relationship, we can specify the cascade type in the PlayerProfile class as follows:

@OneToOne(mappedBy= "playerProfile", cascade= CascadeType.ALL)  
private Player player;

CascadeType.ALL means that if we delete a PlayerProfile object, the associated Player object will also be deleted.

1. If we do not want that to happen, we need to break the association between the two objects before calling delete() on the PlayerProfile object.

The deletePlayerProfile method in the PlayerProfileService class is shown below. The following code removes the link between the PlayerProfile and Player object by manually setting the references to null before deleting from the database.

public void deletePlayerProfile(int id) {  
    PlayerProfile tempPlayerProfile = repo.findById(id).get();   
    //set the playerProfile field of the Player object to null  
    tempPlayerProfile.getPlayer().setPlayerProfile(null);  
    //set the player field of the PlayerProfile object to null  
    tempPlayerProfile.setPlayer(null);  
    //save changes  
    repo.save(tempPlayerProfile);  
    //delete the PlayerProfile object  
    repo.delete(tempPlayerProfile);   
}

Now when the PlayerProfile object is deleted, the Player object is not affected.

Break the bidirectional association before deleting the PlayerProfile entity.

###### /

DatabaseRelationshipsApplication.java

Player.java

PlayerProfile.java

PlayerRepository.java

PlayerProfileRepository.java

PlayerService.java

PlayerProfileService.java

PlayerController.java

PlayerProfileController.java

**Player.java**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

package io.datajek.databaserelationships.onetoone;

import javax.persistence.CascadeType;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.JoinColumn;

import javax.persistence.OneToOne;

import com.fasterxml.jackson.annotation.JsonIdentityInfo;

import com.fasterxml.jackson.annotation.JsonManagedReference;

import com.fasterxml.jackson.annotation.ObjectIdGenerators;

@Entity

@JsonIdentityInfo(generator= ObjectIdGenerators.PropertyGenerator.class, property="id")

public class Player {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private int id;

    private String name;

    @OneToOne(cascade=CascadeType.ALL)//, optional = false)

    @JoinColumn(name = "profile\_id", referencedColumnName = "id")

    //@JsonManagedReference

    private PlayerProfile playerProfile;

    public Player( ) {

    }





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/players>

For the code widget given above, use the URL at which the application is running in place of <http://localhost:8080/>. For example, **/players** means <http://localhost8080/players> for local dev environment. If using POSTMAN with code widget above, use the URL shown under the code widget to access **/players**.

## optional attribute:

1. The @OneToOne annotation has an optional attribute. By default the value is true meaning that the association can be null. We can explicitly set it to false for the playerProfile attribute in the Player class:

@OneToOne(cascade=CascadeType.ALL, optional = false)   
@JoinColumn(name = "profile\_id", referencedColumnName = "id")  
private PlayerProfile playerProfile;

If the value of the optional attribute is set to false, then we will get an error when a Player object is added without an associated PlayerProfile object.

The following **POST** request to **\players** now returns a DataIntegrityViolationException as the playerProfile field cannot be left blank.

{  
    "name": "Federer"  
}

However, adding a Player with a nested PlayerProfile object, as shown below, does not result in an error.

{  
    "name": "Djokovic",  
    "playerProfile": {  
        "twitter" : "@DjokerNole"  
    }  
}

When we add the PlayerProfile object first and then add the Player object by using the reference of the PlayerProfile object, JPA throws the detached entity passed to persist error. This error can be removed by changing the cascade type in the Player class to CascadeType.Merge. REASON: save() method calls persist() for new entities and merge() for existing entities. Player is a new entity so persist() is called and the operation is cascaded to PlayerProfile. However, since the PlayerProfile already exists, it needs to be merged, not persisted. When we change the cascade type to MERGE, persist() is not cascaded to PlayerProfile and the exception is avoided. However, if we add a Player object with a nested PlayerProfile object now, we will get the Not-null property references a transient value error.

You can toggle the value of the optional attribute in the code widget given above.

## Pros and cons of bidirectional relationship

Bidirectional relationships are better than unidirectional relationships in terms of performance as both ends of the relationship are aware of any changes.

When using bidirectional relationships, consistency must be ensured. If a Player object references a PlayerProfile object, the same PlayerProfile object must reference back to the Player object. Failure to ensure consistency can lead to unpredictable JPA behavior.

A con of having bidirectional association is that it may make the application vulnerable in terms of security since the referenced side can now be used to access the owning side (we can access the Player object using the PlayerProfile object). Infinite recursion is also an issue when using bidirectional relationships with Jackson, Hibernate JPA, and/or Elasticsearch implementations.

# One-to-Many Unidirectional Relationship

In this lesson, we will cover one-to-many relationships and learn about the orphan removal attribute.

**We'll cover the following**

* [@OneToMany](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B1qjzBxAnKn#@OneToMany)
* [Cascade type](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B1qjzBxAnKn#Cascade-type)
* [Orphan records](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B1qjzBxAnKn#Orphan-records)
  + [orphanRemoval attribute](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B1qjzBxAnKn#orphanRemoval-attribute)

To show the one-to-many relationship, we will model the case where many players can register for a tournament. We will create a **tournament** table and a **registration** table to model this relationship.

Unidirectional one-to-many relationship means that only one side maintains the relationship details. So given a Tournament entity, we can find the Registrations but we cannot find the Tournament details from a Registration entity.

One-to-Many unidirectional relationship

1. To model the one-to-many relationship, create a new package **onetomany.uni** and define a Tournament class with three fields: id, name and location. The id field is the primary key. We can also save other details like the dates in which the tournament takes place, the type of surface on which it will be played, and the number or rounds etc.

package io.datajek.databaserelationships.onetomany.uni;  
  
@Entity  
public class Tournament {  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private int id;  
    private String name;  
    private String location;  
    //getters and setters  
    //constructor  
    //toString method  
}

1. Next, define the Registration class with just one field, id, for now. The id field is the primary key for the table. We will add more fields later.

The Registration class can store information about the registration date, the type of match (single/ doubles) for which the player registers, and the rank assigned to the player (seed) etc.

package io.datajek.databaserelationships.onetomany.uni;  
  
@Entity  
public class Registration {  
    @Id  
    @GeneratedValue(strategy=GenerationType.IDENTITY)  
    private int id;  
    //getters and setters  
    //constructor  
    //toString method  
}

Since a player registers for a tournament, a registration object should be associated with a player object.

1. Now, we will update the Tournament class to show the registrations. Since a tournament can have multiple registrations, we will add a List of Registrations as a new field.

public class Tournament {  
    //...  
    private List<Registration> registrations = new ArrayList<>();  
    //generate getter and setter methods   
    //update constructor & toString()  
}

## @OneToMany

1. The Tournament class has a one-to-many relationship with the Registration class as one tournament can have multiple registrations. This can be modelled by the @OneToMany annotation. In a one-to-many relationship, the primary key of the one side is placed as a foreign key in the many side.

The @JoinColumn annotation shows that this is the owning side of the relationship. **tournament\_id** will be added as a foreign key column in the **registration** table.

@OneToMany  
@JoinColumn(name="tournament\_id")  
private List<Registration> registrations = new ArrayList<>();

@JoinColumn annotation

In the absence of the @JoinColumn annotation, Hibernate creates a join table for the one-to-many relationship containing the primary keys of both the tables.

If the application is run, it creates the database structure shown below. Here **tournament\_id** is the foreign key column. We can verify this using the H2 web console (at <http://localhost:8080/h2-console> with jdbc:h2:mem:testdb as the connection URL).

Table structure

## Cascade type

1. Next, we will choose the cascade type for this relationship. When a tournament is deleted we will delete the associated registrations as well. This can be achieved by choosing CascadeType.ALL.

@OneToMany(cascade=CascadeType.ALL)  
@JoinColumn(name="tournament\_id")  
private List<Registration> registrations = new ArrayList<>();

1. To set up the association between tournament and registration, we will add a method in the Tournament class that assigns a Registration object to a Tournament object.

public void addRegistration(Registration reg) {  
    registrations.add(reg);  
}

1. Now we will create the repository, service and controller classes for Registration and Tournament in the appropriate packages. The repository interfaces are named TournamentRepository and RegistrationRepository and extend the JpaRepository interface.

The REST controllers TournamentController and RegistrationController have a @RequestMapping of **/tournaments** and **/registrations** respectively. The controller classes call methods the in service layer classes, TournamentService and RegistrationService.

All the above mentioned interfaces and classes are shown in the code widget below.

1. We need a **PUT** mapping in the TournamentController class to assign a registration to a tournament. The addRegistration method with **/{id}/registrations/{registration\_id}** mapping adds a registration with registration\_id to a tournament with id as its key.

@PutMapping("/{id}/registrations/{registration\_id}")  
public Tournament addRegistration(@PathVariable int id, @PathVariable int registration\_id) {  
    Registration registration = registrationService.getRegistration(registration\_id);   
    System.out.println(registration);  
    return service.addRegistration(id, registration);  
}

The corresponding service layer method in TournamentService class is shown:

public Tournament addRegistration(int id, Registration registration) {  
    Tournament tournament = repo.findById(id).get();  
    tournament.addRegistration(registration);  
    return repo.save(tournament);  
}

###### /

Registration.java

Tournament.java

RegistrationRepository.java

TournamentRepository.java

RegistrationService.java

TournamentService.java

RegistrationController.java

TournamentController.java

DatabaseRelationshipsApplication.java

**Registration.java**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

package io.datajek.databaserelationships.onetomany.uni;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

@Entity

public class Registration {

    @Id

    @GeneratedValue(strategy=GenerationType.IDENTITY)

    private int id;

    public Registration() {

    }

    public int getId() {

        return id;

    }

    public void setId(int id) {

        this.id = id;

    }

    @Override

    public String toString() {

        return "Registration [id=" + id + "]";

    }

}





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/tournaments>

For the code widget given above, use the URL at which the application is running in place of <http://localhost:8080/>. For example, **/tournaments** means <http://localhost8080/tournaments> for local dev environment. If using POSTMAN with code widget above, use the URL shown under the code widget to access **/tournaments**.

1. To test the application, first add two tournaments using the following **POST** requests to **/tournaments**:

{  
    "name": "Canadian Open",  
    "location": "Toronto"  
 }

{  
    "name": "US Open",  
    "location": "New York City"  
 }

Next, we will add four registrations by sending **POST** request with an empty body to **/registrations**:

{}

Out of the four registrations, we will associate one with the first tournament and three with the second tournament. This can be achieved by sending the following **PUT** requests:

<http://localhost:8080/tournaments/1/registrations/3> <http://localhost:8080/tournaments/2/registrations/1> <http://localhost:8080/tournaments/2/registrations/2> <http://localhost:8080/tournaments/2/registrations/4>

A **GET** request to **/tournaments** shows the tournaments along with their registrations. The same can be verified using the H2 web console.

Creating a Collection from the above mentioned POST and PUT requests can help reduce setup time for subsequent tests.

Tournaments along with registrations

If we delete the tournament with id 2 by sending a **DELETE** request to **/tournaments/2**, the tournament is deleted along with its three registrations. The **registration** table has only one registration left.

After deleting tournament with id 2

## Orphan records

1. An orphan record is a record with a foreign key value that points to a primary key value that no longer exists. Orphan records point to a lack of referential integrity which means that the data in the tables is not in a consistent state.

In our example, the registration record has a foreign key value of tournament\_id. We can remove a registration from a tournament by breaking the association between the two. In such a case, the record in the registration table would become an orphan as it is no longer linked to any entry in the tournament table. The following figure shows an orphan record:

To demonstrate the concept, we will create a method removeRegistration which breaks the association between a Tournament and a Registration object.

public void removeRegistration(Registration reg) {  
    if (registrations != null)  
        registrations.remove(reg);  
}

We will create a new **PUT** mapping of **/tournaments/{id}/remove\_registrations/{registration\_id}** in the TournamentController class. The removeRegistration method removes the registration entity having registraion\_id as its key from the Tournament entity specified using id.

@PutMapping("/{id}/remove\_registrations/{registration\_id}")  
public Tournament removeRegistration(@PathVariable int id, @PathVariable int registration\_id) {  
    Registration registration = registrationService.getRegistration(registration\_id);   
    return service.removeRegistration(id, registration);  
}

Notice, that the controller calls the service class method, removeRegistration, which simply delegates the call to the removeRegistration method of the Tournament class.

Cascade type REMOVE only cascades the delete operation to child records which are linked to the parent. To show how it works, we will create the same scenario as before (with 2 tournaments and 4 registrations by assigning one registration to the first tournament and three registrations to the second tournament).

With the above changes in place, run the application again and create two tournaments and four registrations. Then associate the registrations with the two tournaments as described above.

We will remove one registration from tournament with id 2 by sending a **PUT** request to **/tournaments/2/remove\_registrations/4**. Now the tournament has two registrations left. Note, that we did not delete the registration, but only removed it from the tournament. The registration record is not associated with any tournament and is an orphan record.

The current state of the database is reflected from the response to **GET** requests to **/tournaments** and **/registrations** as shown below:

After removing a registration from tournament 2

Next, delete the tournament by sending a **DELETE** request to **/tournaments/2**. The delete operation is cascaded to the **registration** table and two records associated with the tournament are deleted. If we perform a **GET** on **/registrations**, we can see the orphan record with id 4 is still in the table.

Registration table contains an orphan record

### orphanRemoval attribute

The @OneToMany annotation has an orphanRemoval attribute which can be used to delete records which have been orphaned.

@OneToMany(cascade=CascadeType.ALL, orphanRemoval=true)      
@JoinColumn(name="tournament\_id")  
private List<Registration> registrations = new ArrayList<>();

To test how this attribute differs from CascadeType.REMOVE, we will recreate the same scenario with two tournament and four registration entries and establish one-to-many associations as mentioned above.

Remove registration with id 4 from tournament 2 using a **PUT** request to **/tournaments/2/remove\_registrations/4**. The orphanRemoval attribute triggers a remove operation for the Registration object no longer associated with the Tournament object thereby leaving the database in a consistent state.

Orphan record from registration table removed

Now we can delete tournament with id 2. **GET** request to **/registrations** shows one registration remaining in the table. The registration which was assigned to tournament with id 2 and later removed became an orphan and was removed because we set the orphanRemoval attribute to true.

The difference between orphanRemoval and CascadeType.REMOVE should be clear from the above example. Using cascade type REMOVE only deleted the two registrations associated with the tournament and left the orphaned record in the table.

# One-to-Many Bidirectional Relationship

This lesson shows how to create a one-to-many relationship.

**We'll cover the following**

* [@ManyToOne](https://www.educative.io/courses/guide-spring-5-spring-boot-2/gxggLk6nlBk#@ManyToOne)
* [Cascade type](https://www.educative.io/courses/guide-spring-5-spring-boot-2/gxggLk6nlBk#Cascade-type)
* [@OneToMany](https://www.educative.io/courses/guide-spring-5-spring-boot-2/gxggLk6nlBk#@OneToMany)

In this lesson we will create a bidirectional one-to-many relationship where a Player can have many Registrations.

Let’s add some real life constraints to the model.

* The first one being that every Registration object must be associated with a Player object.
* Secondly, when we delete a Registration object, the associated Player object should not be deleted.

A bidirectional association between Player and Registration means that we can get all the Registration objects if we have a Player object and vice versa, we can get a Player by using the Registration. Compare this to the unidirectional one-to-many relationship, where we could find the Registration objects given a Tournament but we could not find the Tournament from a Registration object.

GET request to /tournaments and /registrations

The inverse of one-to-many relationship is many-to-one, where many registrations map to one player.

1. For the bidirectional relationship example, create a new package named **bi** inside the **onetomany** package. Copy the Registration and Tournament classes from the **onetomany.uni** package and the Player and PlayerProfile classes from the **onetoone** package along with the associated repository, service and controller classes to the [**onetomany.bi**](http://onetomany.bi/) package.
2. We will start with updating the Registration class. To have a bidirectional relationship, we will add a player field in the Registration class. Generate getter and setter methods for the new field and update the constructor and toString methods.

@Entity  
public class Registration {  
    @Id  
    @GeneratedValue(strategy=GenerationType.SEQUENCE)  
    private int id;  
    private Date registrationDate;  
  
    private Player player;  
    //getters and setters  
    //constructor  
    //toString method  
}

## @ManyToOne

1. There is a many-to-one relationship between the Registration and Player classes where many registrations can map to one player. The many side of a many-to-one bidirectional relationship is always the owning side of the relationship.

To model this relationship, we will use the @ManyToOne annotation with @JoinColumn specifying the column that corresponds to the foreign key column in the **registration** table. The **player** table has a column id which will become the foreign key column player\_id in the **registration** table. This is how the Registration knows how to find its Player.

@ManyToOne  
@JoinColumn(name="player\_id", referencedColumnName = "id")  
private Player player;

The @JoinColumn annotation also shows that this is the owning side of the relationship.

## Cascade type

1. Next, we will choose the cascade type for this relationship. If a Registration object is deleted, the associated Player should not be deleted. This means that the delete operation should not be cascaded. Since we have fine grain control over the cascade types, we will list all of them except for REMOVE.

@ManyToOne(cascade={CascadeType.PERSIST, CascadeType.MERGE,   
                    CascadeType.DETACH,  CascadeType.REFRESH}  
@JoinColumn(name="player\_id", referencedColumnName = "id")  
private Player player;

1. Now we will update the Player class to show tournament registrations. Since a player can have multiple registrations, we will add a List of Registrations as a new field to the class.

private List<Registration> registrations = new ArrayList<>();   
//generate getter and setter methods

## @OneToMany

1. The Player class has a one-to-many relationship with the Registration class as one player can register for many tournaments. This can be modelled by the @OneToMany annotation.

Since the many side (Registration) is the owning side of a bidirectional relationship, we will use the mappedBy attribute here (in the Player class) to specify that this is the inverse side of the relationship.

@OneToMany(mappedBy="player", cascade= CascadeType.ALL)  
private List<Registration> registrations = new ArrayList<>();

We are using cascade type ALL here because we want a player’s registrations to be deleted when the player record is deleted.

The player in the mappedBy attribute references the player field in the Registration class. Hibernate looks at the @JoinColumn annotation on the player field to find the foreign key column.

Next, we will add a method to the Player class that sets the bidirectional relationship. In this method, we will add a Registration object to the Player and also update the Registration to reflect that it belongs to this Player.

//set up bidirectional relationship with Registration class  
public void registerPlayer(Registration reg) {  
    //add registration to the list  
    registrations.add(reg);  
    //set the player field in the registration  
    reg.setPlayer(this);  
}

In the PlayerController class, we will add a new **PUT** mapping to associate a Registration with **registration\_id** with a Player having **id** as key as follows:

@PutMapping("/{id}/registrations/{registration\_id}")  
public Player assignRegistration(@PathVariable int id, @PathVariable int registration\_id) {  
   Registration registration = registrationService.getRegistration(registration\_id);   
   return service.assignRegistration(id, registration);  
}

The controller class method invokes the service class method assignRegistration with the player’s id and a Registration object. The method in PlayerService class is shown:

public Player assignRegistration(int id, Registration registration) {  
    Player player = repo.findById(id).get();  
    player.registerPlayer(registration);  
    return repo.save(player);  
}

After setting up the bidirectional relationship between Player and Registration entities, the updated ERD of our database is shown below. The same can be verified from the web console of H2 database (at <http://localhost:8080/h2-console> with jdbc:h2:mem:testdb as the connection URL).

###### /

Player.java

Registration.java

PlayerRepository.java

RegistrationRepository.java

PlayerService.java

RegistrationService.java

PlayerController.java

RegistrationController.java

DatabaseRelationshipsApplication.java

**Player.java**

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

87

    }

    public String getName() {

        return name;

    }

    public void setName(String name) {

        this.name = name;

    }

    public PlayerProfile getPlayerProfile() {

        return playerProfile;

    }

    public void setPlayerProfile(PlayerProfile playerProfile) {

        this.playerProfile = playerProfile;

    }

    public List<Registration> getRegistrations() {

        return registrations;

    }

    public void setRegistrations(List<Registration> registrations) {

        this.registrations = registrations;

    }

    //set up bi-directional relationship with Registration class

    public void registerPlayer(Registration reg) {

        //add registration to the list

        registrations.add(reg);

        //set the player field in the registration

        reg.setPlayer(this);

    }





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/players>

For the code widget given above, use the URL at which the application is running in place of <http://localhost:8080/>. For example, **/players** means <http://localhost8080/players> for local dev environment. If using POSTMAN with code widget above, use the URL shown under the code widget to access **/players**.

To test the application, first create a player object by sending a **POST** request to **/players** as follows:

{  
    "name": "Djokovic",  
    "playerProfile": {  
        "twitter" : "@DjokerNole"  
    }  
}

Next, create three registration entities by sending **POST** requests to **/registrations** with an empty request body:

{}

Right now the Player and Registration objects are not connected as can be seen from the image below:

GET requests to /players and /registrations

To set up a bidirectional mapping, send a **PUT** request to **/players/1/registrations/1**. This will connect the Player with id 1 with the Registration having id 1.

To assign Registration object with id 3 to the same player send another **PUT** request to **/players/1/registrations/3**.

We can confirm the bidirectional association by sending **GET** requests to **/players** and **/registrations**.

Get request to /players and /registrations after creating bidirectional association

To confirm if the cascade is working correctly, delete one registration associated with the player by sending a **DELETE** request to **/registrations/3**. The registrations table has two records left. A **GET** request to **/players** confirms that delete operation is not cascaded and the player exists in the database. Only the registration record is removed from the player. The results can also be verified from the H2 database.

Delete from registration table not cascaded to player table

# Many-to-Many Unidirectional Relationship

In this lesson, we will learn about implementing a many-to-many relationship using a join table.

**We'll cover the following**

* [@ManyToMany](https://www.educative.io/courses/guide-spring-5-spring-boot-2/mE5EyqM4qE0#@ManyToMany)
* [@JoinTable](https://www.educative.io/courses/guide-spring-5-spring-boot-2/mE5EyqM4qE0#@JoinTable)
* [joinColumns and inverseJoinColumns](https://www.educative.io/courses/guide-spring-5-spring-boot-2/mE5EyqM4qE0#joinColumns-and-inverseJoinColumns)
* [Cascade type](https://www.educative.io/courses/guide-spring-5-spring-boot-2/mE5EyqM4qE0#Cascade-type)

Every tournament has some playing categories like singles and doubles for men and/or ladies. In addition, all four grand slam tournaments have the mixed doubles category. Other less known categories are wheelchair tennis and beach tennis. This scenario fits well into the many-to-many relationship where many categories are part of a tournament and many tournaments have the same playing categories.

A tournament has many categories and a category is included in many tournaments.

We have already created the Tournament class. In this lesson, we will create the Category class and then join the two with a many-to-many relationship. In databases, this relationship is modelled using a join table which has the primary keys of both tables in the relationship.

Many-to-many relationship is implemented using a join table.

We will also model two real life constraints.

* The first one is that one playing category should appear only once in the **category** table (we don’t want multiple entries for the same category).
* The second constraint is that when a tournament entry is deleted, the associated playing categories should not be deleted and vice versa.

1. For the many-to-many database relationship example, create a package **manytomany** and copy all the files from the **bi** package inside the **onetomany** package. These include the Player, PlayerProfile, Registration and Tournament classes and the associated repository, service and controller classes.
2. We will begin by creating a new class, Category in the **io.datajek.databaserelationships.manytomany** package. This class has two fields, id and name, where id is the primary key. Since we do not want the same category name to appear more than once, we will impose the unique key constraint using the unique attribute of the @Column annotation.

package io.datajek.databaserelationships.manytomany;  
  
@Entity  
public class Category {  
    @Id   
    @GeneratedValue(strategy=GenerationType.IDENTITY)  
    private int id;  
   
    @Column(unique=true)  
    private String name;  
    //constructors  
    //getters and setters  
}

@Column(unique=true) ensures that the same category name is not entered more than once.

1. For a many-to-many relationship, any side can become the parent/ owner. It depends on how the business rules are defined. Let’s say, we cannot have a tournament without a category attached to it but we can have a category that is not associated with any tournament. Given this scenario, a category can exist on its own but a tournament needs to have one or more categories associated with it. So, Category becomes the owning/parent side and Tournament becomes the referenced/ child side.

In a unidirectional many-to-many relationship, we put the relationship on the child side. So, in the Tournament class we have to put a link to the Category class. Since a tournament can have more than one categories, we will create a List of categories as follows:

@Entity  
public class Tournament {  
    //...  
  
    private List<Category> playingCategories = new ArrayList<>();  
  
    //...  
}

Create a getter and setter method for this field.

## @ManyToMany

1. The @ManyToMany annotation is used to create a many to many relationships between two entities.

@ManyToMany  
private List<Category> playingCategories = new ArrayList<>();

## @JoinTable

1. The many-to-many relationship is different from the relationships that we have seen so far. Here, the foreign keys are stored in a separate table called a join table instead of being placed inside the parent table or the child table. The join table connects two tables and contains the foreign keys of both the tables. The tournament and category tables do not contain the keys of each other, rather the primary keys of both these tables go in the join table.

## joinColumns and inverseJoinColumns

1. joinColumns attribute specifies the column(s) in the owner table that becomes a foreign key in the join table. inverseJoinColumns attribute specifies the foreign key column(s) from the inverse side.

@JoinTable annotation

For a unidirectional relationship, the Category class does not need any information about the tournaments. Here, we are setting the category in the tournament. When a tournament is saved, it creates a join table entry with the appropriate keys.

The addCategory method in the Tournament class sets up the many-to-many relationship:

//set up many-to-many relationship  
public void addCategory(Category category) {  
    playingCategories.add(category);  
}

In the TournamentController we will add a **PUT** mapping **/{id}/categories/{category\_id}** to assign a Category with **category\_id** to a Tournament with **id** as key.

@PutMapping("/{id}/categories/{category\_id}")  
public Tournament addCategory(@PathVariable int id, @PathVariable int category\_id) {  
    Category category = categoryService.getCategory(category\_id);   
    return service.addCategory(id, category);  
}

The corresponding method in the TournamentService class are shown below:

public Tournament addCategory(int id, Category category) {  
    Tournament tournament = repo.findById(id).get();  
    tournament.addCategory(category);  
    return repo.save(tournament);  
}

## Cascade type

1. We will not use cascade type REMOVE as we do not want to delete tournaments when we delete a category. We will also not use cascade type PERSIST, because that will cause an error if we try to add a tournament with nested category values.

@ManyToMany(Cascade = CascadeType.MERGE, CascadeType.DETACH, CascadeType.REFRESH)

The ERD of the project is shown below. The same table structure can be verified from the web console of H2 database (at <http://localhost:8080/h2-console> with jdbc:h2:mem:testdb as the connection URL). Note that the code widget below does not show the implementation of all classes.

###### /

Tournament.java

Category.java

TournamentRepository.java

CategoryRepository.java

TournamentService.java

CategoryService.java

TournamentController.java

CategoryController.java

DatabaseRelationshipsApplication.java

**Tournament.java**

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

import javax.persistence.FetchType;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.JoinColumn;

import javax.persistence.JoinTable;

import javax.persistence.ManyToMany;

import javax.persistence.OneToMany;

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;

@Entity

public class Tournament {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private int id;

    private String name;

    private String location;

    @OneToMany(cascade=CascadeType.ALL, orphanRemoval=true)

    @JoinColumn(name="tournament\_id")

    private List<Registration> registrations = new ArrayList<>();

    @ManyToMany(cascade= {CascadeType.DETACH, CascadeType.MERGE, CascadeType.REFRESH})

    @JoinTable(

            name = "tournament\_categories",

            joinColumns= @JoinColumn(name ="tournament\_id"),  //FK of the owning side

            inverseJoinColumns=@JoinColumn(name="category\_id")  //FK of inverse side

            )

    @JsonIgnoreProperties("tournaments")





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/tournaments>

For the code widget given above, use the URL at which the application is running in place of <http://localhost:8080/>. For example, **/tournaments** means <http://localhost8080/tournaments> for local dev environment. If using POSTMAN with code widget above, use the URL shown under the code widget to access **/tournaments**.

To test the many-to-many relationship, we will add two tournaments by sending **POST** requests to **/tournaments** as follows:

{  
    "name": "Canadian Open",  
    "location": "Toronto"  
}

{  
    "name": "US Open",  
    "location": "New York City"  
}

Next, we will add five categories by sending **POST** requests to **/categories** as follows:

{  
    "name" : "Men's Singles"  
}

{  
    "name" : "Men's Doubles"  
}

{  
    "name" : "Ladies Singles"  
}

{  
    "name" : "Ladies Doubles"  
}

{  
    "name" : "Mixed Doubles"  
}

**GET** request to **/tournaments** and **/categories** has the following response:

Right now, the Tournament and Category objects are not connected. We will assign categories with id 1 and 2 to Toronto Open having id 1 by sending the following **PUT** requests:

**/tournaments/1/categories/1**

**/tournaments/1/categories/2**

Assign all 5 categories to the US Open having id 2 with the following **PUT** requests:

**/tournaments/2/categories/1**

**/tournaments/2/categories/2**

**/tournaments/2/categories/3**

**/tournaments/2/categories/4**

**/tournaments/2/categories/5**

Grouping these requests in a Collection can help reduce the run time.

We can also add a tournament with nested category objects as follows:

{  
    "name":"Western and Southern Open",  
    "location":"Cincinatti",  
    "registrations": null,  
    "playingCategories":[{  
        "id":3  
    },  
    {  
        "id":4  
    }]  
}

A **GET** request to **/tournaments** shows the three tournaments along with their categories.

We can test cascade by deleting a tournament and seeing if the categories also get deleted. Send a **DELETE** request to **/tournaments/1** to delete tournament with id 1. **GET** request to **/tournaments** and **/categories** show that the tournament gets deleted but the categories are not deleted.

The results can also be verified from the H2 database web console.

# Many-to-Many Bidirectional Relationship

Learn how to change the unidirectional many-to-many relationship to a bidirectional relationship.

**We'll cover the following**

* [mappedBy property](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7DQWooWDv2B#mappedBy-property)
* [@JsonIgnoreProperties](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7DQWooWDv2B#@JsonIgnoreProperties)

In a bidirectional relationship, each side has a reference to the other. In our example, the Category class did not have any reference to the Tournament class. Now we will add a reference to the Tournament class so that the relationship can be navigated from both sides. This will have no effect on the underlying database structure. The join table **tournament\_catogories** already has the foreign keys of both the tournament and category tables and it is possible to write SQL queries to get tournaments associated with a category.

For a many-to-many relationship, we can choose any side to be the owner. The relationship is configured in the owner side using the @JoinTable annotation. On the target side we use the mappedBy attribute to specify the name of the field that maps the relationship in the owning side. From the database design point of view, there is no owner of a many-to-many relationship. It would not make any difference to the table structure if we swap the @JoinTable and mappedBy.

1. We will begin by creating a List of tournaments in the Category class along with the getter and setter methods.

package io.datajek.databaserelationships.manytomany;  
  
@Entity  
public class Category {  
    @Id  
    @GeneratedValue(strategy=GenerationType.IDENTITY)  
    private int id;  
   
    @Column(unique = true)  
    private String name;  
  
    private List<Tournament> tournaments = new ArrayList<>();  
    //...  
}

## mappedBy property

1. On the tournaments field created above, use the @ManyToMany annotation with the mappedBy property. This shows the value that is used to map the relationship in the Tournament class.

@ManyToMany(mappedBy= "playingCategories")  
private List<Tournament> tournaments = new ArrayList<>();

1. We will also use the cascade property to cascade all operations except REMOVE because we do not want to delete all associated tournaments, if a category gets deleted.

@ManyToMany(mappedBy= "playingCategories",  
        cascade= {CascadeType.DETACH, CascadeType.MERGE, CascadeType.PERSIST, CascadeType.REFRESH},  
        fetch=FetchType.LAZY)  
    private List<Tournament> tournaments = new ArrayList<>();

1. It is the responsibility of the application to manage a bidirectional relationship. When we add a category to a tournament, we must also add the tournament to that category to preserve the relationship in both directions. Failure to do so may result in unexpected JPA behavior.

We will update the addCategory method in the Tournament class to set up the bidirectional relationship by adding the tournament to the category.

public void addCategory(Category category) {  
   playingCategories.add(category);  
   //set up bidirectional relationship  
   category.getTournaments().add(this);  
}

Similarly, we will update the removeCategory method in the Tournament class to remove the association from both sides.

public void removeCategory(Category category) {  
   if (playingCategories != null)  
          playingCategories.remove(category);  
   //update bidirectional relationship  
   category.getTournaments().remove(this);  
}

## @JsonIgnoreProperties

1. JSON gets into infinite recursion when trying to de-serialize bidirectional relationships. We have seen two ways to solve this issue in the One-to-One Bidirectional Relationship lesson. Here, we will see yet another approach to avoid infinite recursion. We can use the property that we want to ignore with the @JsonIgnoreProperties. This annotation can be used at field level in both the Tournament and Category class.

@JsonIgnoreProperties("tournaments")  
private List<Category> playingCategories = new ArrayList<>();

@JsonIgnoreProperties("playingCategories")  
private List<Tournament> tournaments = new ArrayList<>();

In a many-to-many relationship, there is no owner when it comes to the table structure. This is different from a one-to-many relationship where the many side is always the owning side containing the key of the one side.

###### /

Tournament.java

Category.java

TournamentRepository.java

CategoryRepository.java

TournamentService.java

CategoryService.java

TournamentController.java

CategoryController.java

DatabaseRelationshipsApplication.java

**Tournament.java**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

package io.datajek.databaserelationships.manytomany;

import java.util.ArrayList;

import java.util.List;

import javax.persistence.CascadeType;

import javax.persistence.Entity;

import javax.persistence.FetchType;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.JoinColumn;

import javax.persistence.JoinTable;

import javax.persistence.ManyToMany;

import javax.persistence.OneToMany;

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;

@Entity

public class Tournament {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private int id;

    private String name;

    private String location;

    @OneToMany(cascade=CascadeType.ALL, orphanRemoval=true)

    @JoinColumn(name="tournament\_id")

    private List<Registration> registrations = new ArrayList<>();





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/tournaments>

To test this application, we will add two tournaments and five categories.

To create tournament entries, send **POST** request to **/tournaments** as follows:

{  
    "name": "Canadian Open",  
    "location": "Toronto"  
}

{  
    "name": "US Open",  
    "location": "New York City"  
}

Then, add five categories by sending **POST** requests to **/categories** as follows:

{  
    "name" : "Men's Singles"  
}

{  
    "name" : "Men's Doubles"  
}

{  
    "name" : "Ladies Singles"  
}

{  
    "name" : "Ladies Doubles"  
}

{  
    "name" : "Mixed Doubles"  
}

A **GET** request to **/categories** now shows the tournaments associated with each catogory. This is different from the many-to-many unidirectional relationship, the Category had no information about Tournament.

GET request to /categories in a bidirectional relationship

We can also test the cascade options by deleting a tournament or category and verify the results using the web console of H2 database (at <http://localhost:8080/h2-console> with jdbc:h2:mem:testdb as the connection URL).

# Spring Aspect Orientated Programming (AOP)

**Aspect Oriented Programming**

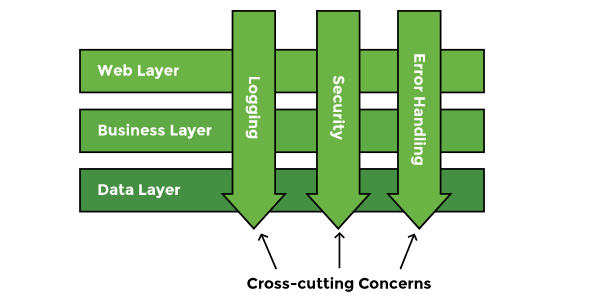
Learn what Aspect Oriented Programming is and its importance. Also set up an example.

**We'll cover the following**

* [What is AOP?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/xlv8Nv2opQB#What-is-AOP?)
* [spring-aop](https://www.educative.io/courses/guide-spring-5-spring-boot-2/xlv8Nv2opQB#spring-aop)

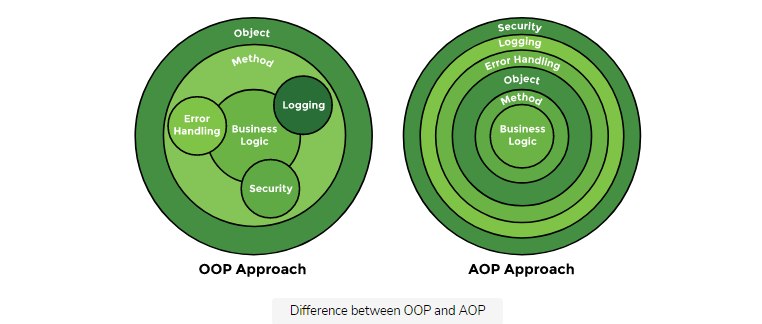
**What is AOP?**

Aspect Oriented Programming (AOP) is the best approach for implementing cross-cutting concerns. Applications are divided into layers like web, business, data, etc. Each layer works independently. There are some concerns that are common across layers. These include security, logging, transaction management, auditing, error handling, performance tracking, etc. These concerns are present in all the layers and are thus called *cross-cutting concerns*…



AOP provides an easy way to add concerns like printing logs or tracking performance of methods across layers. For example, we may want to print logs of methods that provide a certain feature. These methods may belong to the web, business, or data layer. Using the OOP approach, we can call the logger from these methods. The problem with such an approach is that in future, if the logs are not needed anymore, or if they are needed for another group of methods, we will have to make changes to the source code. By following the AOP approach, such changes are easy to maintain. Instead of changing the source code, concerns are implemented separately. Thus, any cross-cutting concern can be added and removed without recompiling the complete code by only changing the config files. The logging functionality can be defined separately and applied to any method belonging to any layer easily.

Difference between OOP and AOP



**spring-aop**

spring-aop is a popular implementation of AOP provided by Spring. It is not as powerful as AspectJ. spring-aop can be used to intercept any calls to the beans managed by Spring. Once method calls are intercepted, cross-cutting concerns can be applied before, after, or around the method logic.

Let’s now understand AOP with an example. We will make a modified version of the movie recommender system example. There are two classes in the business layer implementing two techniques for finding movies to recommend. The data layer contains two classes namely User and Movie.

1. Create a Spring Boot application using the [Spring Initializr](http://start.spring.io/). The **GroupId** is io.datajek.springaop, while the **ArtifactId** is movie-recommender-aop. Right now, we do not need to add any dependency to the project. Once the project has been created and imported in the IDE, we will add the spring-aop dependency to **pom.xml** as follows:

<dependency>  
   <groupId>org.springframework.boot</groupId>  
   <artifactId>spring-boot-starter-aop</artifactId>  
</dependency>

Maven will download the spring-aop and aspectj-core jars when the file is saved. These jars will bring in the AOP functionality to our application.

1. Next, we will create two business classes, FilteringTechnique1 and FilteringTechnique2, and mark them with the @Service annotation. We will create a separate package to keep our business logic in one place. FilteringTechnique1 recommends movies based on their similarity, while FilteringTechnique2 takes the user’s watch history into account.

We will create two classes, Movie and User, in the data layer. These classes will be placed in a separate package, and will contain the @Repository annotation to denote that they belong to the data access layer.

@Repository  
public class Movie {  
    public String getMovieDetails() {  
        //interacts with the Movie repository  
        return "movie details";       
    }  
}

@Repository  
public class User {  
    public String getUserDetails() {  
        //interacts with the User repository  
        return "user details";        
    }  
}

The Movie and User classes have a method which returns some dummy information back. Classes FilteringTechnique1 and FilteringTechnique2 interact with classes in the data layer as follows:

@Service  
public class FilteringTechnique1 {  
    @Autowired  
    private Movie movie;  
   
    public String contentBasedFiltering() {  
        String movieDetails = movie.getMovieDetails();  
        return movieDetails;  
    }  
}

@Service  
public class FilteringTechnique2 {  
    @Autowired  
    private User user;  
   
    public String collaborativeFiltering() {  
        String userDetails =  user.getUserDetails();  
        return userDetails;  
    }  
}

1. We will have the MovieRecommenderAopApplication class implement the CommandLineRunner interface which will enable us to write dynamic code. CommandLineRunner is used when we want to execute some code right after the context is loaded and as soon as the startup completes.

We will autowire the business class objects and use them to call the methods and display the output in the log as follows:

@SpringBootApplication  
public class MovieRecommenderAopApplication implements CommandLineRunner{  
    private Logger logger = LoggerFactory.getLogger(this.getClass());  
   
    @Autowired  
    private FilteringTechnique1 filter1;  
   
    @Autowired  
    private FilteringTechnique2 filter2;  
      
    public static void main(String[] args) {      
         SpringApplication.run(MovieRecommenderAopApplication.class, args);  
    }  
  
    @Override  
    public void run(String... args) throws Exception {  
        logger.info("{}",filter1.contentBasedFiltering());  
        logger.info("{}",filter2.collaborativeFiltering());  
    }  
}

If the code given below is run, the output shows the data being returned from the data layer.

MovieRecommenderAopApplication.java

FilteringTechnique1.java

FilteringTechnique2.java

Movie.java

User.java

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

package io.datajek.springaop.movierecommenderaop;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import io.datajek.springaop.movierecommenderaop.business.FilteringTechnique1;

import io.datajek.springaop.movierecommenderaop.business.FilteringTechnique2;

@SpringBootApplication

 public class MovieRecommenderAopApplication implements CommandLineRunner{

    private Logger logger = LoggerFactory.getLogger(this.getClass());

    @Autowired

    private FilteringTechnique1 filter1;

    @Autowired

    private FilteringTechnique2 filter2;

    public static void main(String[] args) {

        SpringApplication.run(MovieRecommenderAopApplication.class, args);

    }

    @Override

    public void run(String... args) throws Exception {

        logger.info("{}",filter1.contentBasedFiltering());

        logger.info("{}",filter2.collaborativeFiltering());





Run

Save

Reset

Now that the setup is complete, we will start intercepting method calls.

**Terminology**

Before getting our hands dirty with code, let’s take a look at the core concepts of AOP.

**We'll cover the following**

* [Aspect](https://www.educative.io/courses/guide-spring-5-spring-boot-2/3jjq3L6VW0n#Aspect)
* [Pointcut](https://www.educative.io/courses/guide-spring-5-spring-boot-2/3jjq3L6VW0n#Pointcut)
* [Advice](https://www.educative.io/courses/guide-spring-5-spring-boot-2/3jjq3L6VW0n#Advice)
* [Joinpoint](https://www.educative.io/courses/guide-spring-5-spring-boot-2/3jjq3L6VW0n#Joinpoint)
* [Weaving](https://www.educative.io/courses/guide-spring-5-spring-boot-2/3jjq3L6VW0n#Weaving)
* [Weaver](https://www.educative.io/courses/guide-spring-5-spring-boot-2/3jjq3L6VW0n#Weaver)

**Aspect**

An aspect is a Java class that implements cross-cutting concerns. The @Aspect annotation is used to define a class as an aspect. An aspect is a combination of the kinds of methods to intercept and what to do after intercepting them.

Aspects can define functionality for any concern like logging, performance management, or transaction management that cuts across multiple application layers.

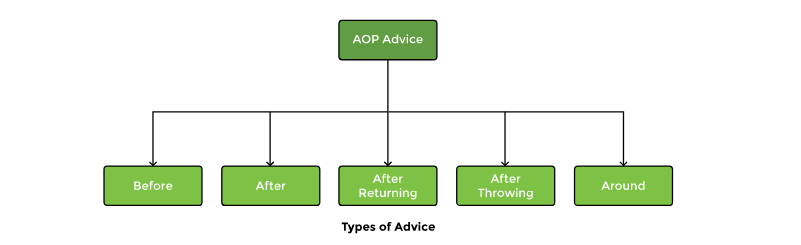
**Pointcut**

Pointcuts are expressions that determine which method calls will be intercepted. These expressions determine whether an advice needs to be executed or not. Pointcuts are defined following a particular syntax.

Pointcuts should be carefully defined, as they determine how many calls will be intercepted.

**Advice**

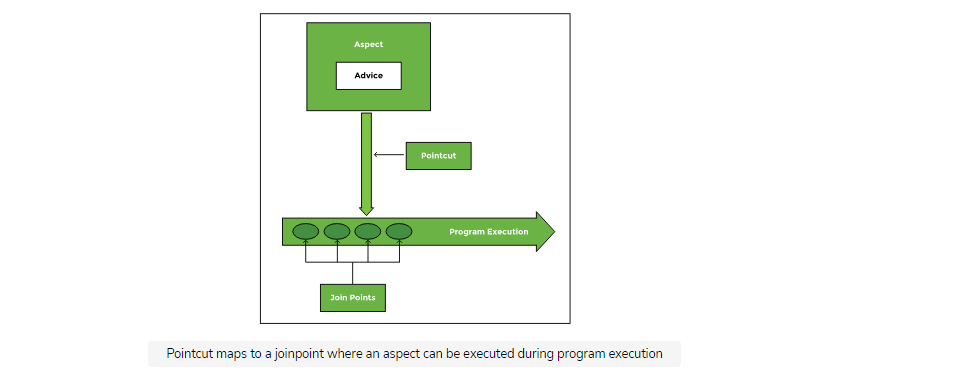
The tasks performed after intercepting a method call are called advices. It is the logic of the methods in a class marked with @Aspect. Advices are basically the methods that get executed when a method calls meets a pointcut. These methods can get executed before, around the time of, or after the execution of the intercepted method call. There are different advice types as shown below.



**Joinpoint**

All method calls that are intercepted are joinpoints. It is a point in the program execution where an aspect can be plugged in. It contains the name of the intercepted method call. The following figure shows the big picture of how AOP works:

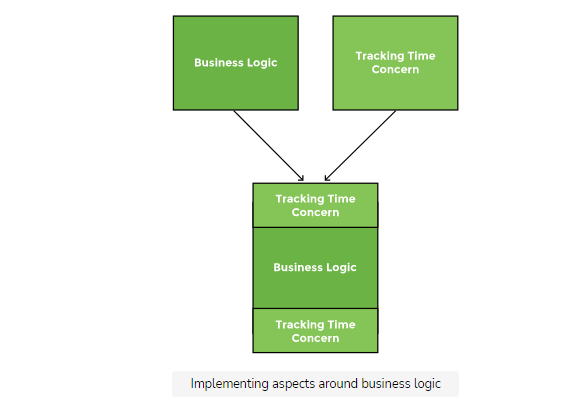
Pointcut maps to a joinpoint where an aspect can be executed during program execution



**Weaving**

The process of implementing AOP around method calls is called weaving. Weaving links an aspect with objects in the application to create an advised object. The aspect is called at the right moment. For example, if we are tracking the execution time of some methods in our application, the weaving process will be like this:

Implementing aspects around business logic



**Weaver**

The framework that ensures that an aspect is invoked at the right time is called a weaver.

**Defining an Aspect**

Learn how to define an aspect and intercept message calls.

**We'll cover the following**

* [Aspect](https://www.educative.io/courses/guide-spring-5-spring-boot-2/R1jp08VWORz#Aspect)
* [Advice](https://www.educative.io/courses/guide-spring-5-spring-boot-2/R1jp08VWORz#Advice)
* [Pointcut expression](https://www.educative.io/courses/guide-spring-5-spring-boot-2/R1jp08VWORz#Pointcut-expression)
* [Joinpoint](https://www.educative.io/courses/guide-spring-5-spring-boot-2/R1jp08VWORz#Joinpoint)

In this lesson, we will define an aspect to intercept all the calls to the business layer and log their output. We will create a number of aspects and place all of them in a separate package.

When we intercept method calls, we have the option to perform tasks before the method is executed as well as afterwards. To define an aspect for a cross-cutting concern, we will perform the following steps:

* Define aspect class.
* Write methods containing the advice to be executed when method calls are intercepted.
* Write pointcut expressions for intercepting method calls.

**Aspect**

1. Suppose we want to intercept method calls to check if the user has access before the method gets executed. Access check is a cross-cutting concern and instead of implementing it in every method, we can write it as an aspect.

We will create a class called AccessCheckAspect and place it in the aspect package. To establish that this is a configuration class, we will use the @Configuration annotation. We will also add the @Aspect annotation to establish that this class is related to AOP.

@Aspect  
@Configuration  
public class AccessCheckAspect {  
  
}

**Advice**

1. The next step is to define a method that contains the logic of the steps that need to be carried out when a method call gets intercepted. We will create a method called userAccess and print a message as follows:

@Aspect  
@Configuration  
public class AccessCheckAspect {  
    private Logger logger = LoggerFactory.getLogger(this.getClass());  
  
    public void userAccess(JoinPoint joinPoint) {  
        logger.info("Intercepted method call");  
    }  
}

**Pointcut expression**

1. User access needs to be checked before a method gets executed. We need the @Before annotation on our method. It ensures that the advice is run before the method is executed.

@Before needs an argument which specifies the method calls that will be intercepted. This is called the pointcut. Pointcuts are defined in the following format:

execution(\* PACKAGE.\*.\*(..))

The pointcut expression starts with a key word called a designator, which tells Spring AOP what to match. execution is the primary designator which matches method execution joinpoints.

* + The first \* in the expression corresponds to the return type. \* means any return type.
  + Then comes the package name followed by class and method names.
  + The first \* after package means any class and the second \* means any method. Instead of \*, we could specify the class name and method name to make the pointcut expression specific.
  + Lastly, parentheses correspond to arguments. (…) means any kind of argument.

Suppose we want to intercept calls to methods belonging to the business package. The pointcut expression, in this case, will be:

@Before("execution(\* io.datajek.springaop.movierecommenderaop.business.\*.\*(..))")

If we use this pointcut expression and run the application, the message in the method will be printed twice, indicating that two method calls have been intercepted.

The userAccess method is invoked before the actual method. This method contains the logic for checking user access, which is not shown.

**Joinpoint**

1. To find out which method calls have been intercepted, we will use a join point as an argument to the method. The joinpoint contains the name of the method that is intercepted. We can use the joinpoint to print the name of the method as follows:

@Before("execution(\* io.datajek.springaop.movierecommenderaop.business.\*.\*(..))")  
public void userAccess(JoinPoint joinPoint) {  
    logger.info("Intercepted call before execution: {}", joinPoint);  
}

MovieRecommenderAopApplication.java

AccessCheckAspect.java

FilteringTechnique1.java

FilteringTechnique2.java

Movie.java

User.java

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

package io.datajek.springaop.movierecommenderaop.aspect;

import org.aspectj.lang.JoinPoint;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.context.annotation.Configuration;

@Aspect

@Configuration

public class AccessCheckAspect {

    private Logger logger = LoggerFactory.getLogger(this.getClass());

    @Before("execution(\* io.datajek.springaop.movierecommenderaop.business.\*.\*(..))")

    public void before(JoinPoint joinPoint) {

        //  logger.info("Intercepted method call");

        logger.info("Intercepted call before execution of: {}", joinPoint);

        //access check logic

    }

}





Run

Save

Reset

It can be seen that the intercepted method calls are FilteringTechnique1.contentBasedFiltering() and FilteringTechnique2.collaborativeFiltering().

The pointcut that we defined is applicable on all methods in the business layer.

**Pointcut Expressions**

Play around with pointcut expressions to gain a deeper understanding of how method calls get intercepted.

**We'll cover the following**

* [Intercepting all method calls in a package](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B8Rm0MkRl9x#Intercepting-all-method-calls-in-a-package)
* [Intercepting all method calls](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B8Rm0MkRl9x#Intercepting-all-method-calls)
* [Intercepting calls using return type](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B8Rm0MkRl9x#Intercepting-calls-using-return-type)
* [Intercepting calls to a specific method](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B8Rm0MkRl9x#Intercepting-calls-to-a-specific-method)
* [Intercepting calls with specific method arguments](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B8Rm0MkRl9x#Intercepting-calls-with-specific-method-arguments)
* [Combining pointcut expressions](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B8Rm0MkRl9x#Combining-pointcut-expressions)

The way pointcuts are defined is important because it decides the method calls that will be intercepted. If we have two packages, we can define which method calls will be intercepted.

If we remove the last part of the package name, all calls will be intercepted.

**Intercepting all method calls in a package**

1. We have the following pointcut expression:

@Before("execution(\* io.datajek.springaop.movierecommenderaop.business.\*.\*(..))")  
public void before(JoinPoint joinPoint) {  
    //intercept a call  
    logger.info("Intercepted call before execution of: {}", joinPoint);  
    //access check logic          
}

This pointcut intercepted calls belonging to the business package. Since we used \* in place of the class and method names, all calls to methods in the business package were intercepted. If we change the package from business to data, only calls to methods in the data package will be intercepted.

**Intercepting all method calls**

1. If we remove the name of the package (business), the pointcut expression becomes:

@Before("execution(\* io.datajek.springaop.movierecommenderaop..\*.\*(..))")

This will intercept all calls in the movierecommenderaop package. The MovieRecommenderAopApplication.run method will also get intercepted because it is in the movierecommenderaop package. All method calls in the business and data layer also get intercepted.

**Intercepting calls using return type**

1. Say, we want to intercept calls to all methods that return a String value. This can be done by specifying String as the return type in the pointcut expression as follows:

@Before("execution(String io.datajek.springaop.movierecommenderaop..\*.\*(..))")

Here we are looking at calls in all subpackages of the movierecommenderaop package and matching the return type of the method calls to String. If this expression is used, four method calls will be intercepted, two belonging to the business package and two from the data package.

**Intercepting calls to a specific method**

1. If we want to intercept calls to all methods that have the word *Filtering* in it, we will use the following pointcut expression:

@Before("execution(String io.datajek.springaop.movierecommenderaop..\*.\*Filtering(..))")

The wildcard \* used in place of the method name will match all methods that have the word *Filtering* in it. Calls to contentBasedFiltering() and collaborativeFiltering() will be intercepted. If we change the word *Filtering* to *Filter*, no method calls will match our criterion.

**Intercepting calls with specific method arguments**

1. Consider the following pointcut expression:

@Before("execution(\* io.datajek.springaop.movierecommenderaop..\*.\*(String))")

This pointcut will match method calls having one parameter of String type. We can modify this expression to match all method calls with String as the first argument as follows:

@Before("execution(\*io.datajek.springaop.movierecommenderaop..\*.\*(String,..))")

**Combining pointcut expressions**

1. The && , || and ! symbols can be used to combine different pointcut expressions.

@Before("execution(\* io.datajek.springaop.movierecommenderaop..\*.\*Filtering(..)) || execution(String io.datajek.springaop.movierecommenderaop..\*.\*(..))")

This pointcut expression will match method calls that have the word *Filtering* in them as well as those that return a String. In all, four methods will be intercepted matching either of the two conditions.

All the pointcut expressions discussed above are provided in the code widget below. You can use them, or create your own pointcuts and play around with them to see which method calls get intercepted.

MovieRecommenderAopApplication.java

AccessCheckAspect.java

FilteringTechnique1.java

FilteringTechnique2.java

Movie.java

User.java

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

package io.datajek.springaop.movierecommenderaop.aspect;

import org.aspectj.lang.JoinPoint;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.context.annotation.Configuration;

@Aspect

@Configuration

public class AccessCheckAspect {

    private Logger logger = LoggerFactory.getLogger(this.getClass());

    @Before("execution(\* io.datajek.springaop.movierecommenderaop..\*.\*Filtering(..)) || execution(String io.datajek.springaop.movierecommenderaop..\*.\*(..))")

    //@Before("execution(String io.datajek.springaop.movierecommenderaop..\*.\*(String))")

    //@Before("execution(\* io.datajek.springaop.movierecommenderaop..\*.\*Filtering(..))")

    //@Before("execution(String io.datajek.springaop.movierecommenderaop..\*.\*(..))")

    //@Before("execution(\* io.datajek.springaop.movierecommenderaop.business.\*.\*(..))")

    public void before(JoinPoint joinPoint) {

        //  logger.info("Intercepted method call");

        logger.info("Intercepted call before execution of: {}", joinPoint);

        //access check logic

    }

}





Run

Save

Reset

**After Aspect**

Learn how to perform a task after an intercepted method call has been executed and what to do in case it throws an exception.

**We'll cover the following**

* [@AfterReturning](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JQkvD3O8YNJ#@AfterReturning)
* [@AfterThrowing](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JQkvD3O8YNJ#@AfterThrowing)
* [@After](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JQkvD3O8YNJ#@After)

Logging is a cross-cutting concern for which aspects can be created. In this lesson, we will create another aspect that will log the values returned *after* the methods have been executed.

1. We will create an aspect called **LoggingAspect** by creating a Java class and mark it with @Aspect and @Configuration annotations.

@Aspect  
@Configuration  
public class LoggingAspect {  
  
}

**@AfterReturning**

1. The LoggingAspect class will have a method LogAfterExecution, which will print a message if the method is successfully executed.

To tell spring-aop that this method needs to be called after the intercepted method call is executed, we will use the @AfterReturning annotation.

@AfterReturning("execution(\* io.datajek.springaop.movierecommenderaop.data.\*.\*(..))")  
public void LogAfterExecution(JoinPoint joinPoint) {  
    logger.info("Method {} complete", joinPoint);  
}

We can get the return values of the method here by using the returning tag. Now that pointcut is not the only argument in the @AfterReturning annotation, we will use tags to differentiate arguments as follows:

@AfterReturning(  
            value = "execution(\* io.datajek.springaop.movierecommenderaop.business.\*.\*(..))",  
            returning = "result")  
public void LogAfterExecution(JoinPoint joinPoint, Object result) {  
    logger.info("Method {} returned with: {}", joinPoint, result);  
}

value contains the pointcut expression and returning contains the value that is returned by the executing method, which is stored in result and passed to the LogAfterExecution method. Note that result is of type Object because different methods will have different return types.

We can use result to log the return value of all intercepted methods. If the application is run, two methods from the business layer are intercepted and the return values are also displayed.

**@AfterThrowing**

1. To intercept an exception, another annotation, @AfterThrowing, is used. We can get the result of the exception using the throwing tag.

We will use this pointcut on a method called LogAfterException as follows:

@AfterThrowing(  
 value = "execution(\* io.datajek.springaop.movierecommenderaop.business.\*.\*(..))",  
 throwing = "exception")  
public void LogAfterException(JoinPoint joinPoint, Object exception) {  
    logger.info("Method {} returned with: {}", joinPoint, exception);  
}

This advice will be executed in case there is an exception in any method call from the business layer.

**@After**

1. The @After annotation is a generic annotation that is used in both scenarios, whether the method execution is successful or results in an exception. The method LogAfterMethod demonstrates the use of this annotation.

@After("execution(\* io.datajek.springaop.movierecommenderaop.business.\*.\*(..))")  
public void LogAfterMethod(JoinPoint joinPoint) {  
    logger.info("After method call {}", joinPoint);  
}

Since @After is a generic annotation, if the application is run, this method also gets executed alongside the LogAfterExecution method marked with the @AfterReturning annotation.

The code widget below contains the aspect created in this lesson as well as the one created in the previous lessons. If you wish to run both aspects, make sure to uncomment the @Aspect annotation from AccessCheckAspect.

MovieRecommenderAopApplication.java

LoggingAspect.java

AccessCheckAspect.java

FilteringTechnique1.java

FilteringTechnique2.java

Movie.java

User.java

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

package io.datajek.springaop.movierecommenderaop.aspect;

import org.aspectj.lang.JoinPoint;

import org.aspectj.lang.annotation.After;

import org.aspectj.lang.annotation.AfterReturning;

import org.aspectj.lang.annotation.AfterThrowing;

import org.aspectj.lang.annotation.Aspect;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.context.annotation.Configuration;

@Aspect

@Configuration

public class LoggingAspect {

    private Logger logger = LoggerFactory.getLogger(this.getClass());

    @AfterReturning(value = "execution(\* io.datajek.springaop.movierecommenderaop.business.\*.\*(..))",

                    returning = "result")

    //@AfterReturning("execution(\* io.datajek.springaop.movierecommenderaop.data.\*.\*(..))")

    public void logAfterExecution(JoinPoint joinPoint, Object result) {

        //logger.info("Method {} complete", joinPoint);

        logger.info("{} returned with: {}", joinPoint, result);

    }

    @AfterThrowing(value = "execution(\* io.datajek.springaop.movierecommenderaop.business.\*.\*(..))",

            throwing = "exception")

        public void logAfterException(JoinPoint joinPoint, Object exception) {

            logger.info("Exception in {} returned with: {}", joinPoint, exception);

    }





Run

Save

Reset

Methods annotated with both @AfterReturning and @After get executed.

**Around Aspect**

Investigate how to wrap an aspect around an intercepted method call with the @Around annotation.

There are aspects that are executed before and after the intercepted method calls. This lesson looks at another type of aspect, the around aspect, which is executed around the intercepted method call. This kind of aspect is useful if we want to perform a task before the intercepted method starts execution and after the method has returned.

A good example of an around aspect is measuring the time taken by the method call to execute. We can note the time when the method call is intercepted, then allow the intercepted method to execute, and note the time after the method returns. Instead of having two separate annotations, @Before and @After, we can accomplish this task using the more advanced @Around annotation.

1. We will create a class called **ExecutionTimeAspect**. To mark it as an aspect and a configuration, we will use the @Aspect and @Configuration annotations as follows:

@Aspect  
@Configuration  
public class ExecutionTimeAspect {  
  
}

1. In the ExecutionTimeAspect class, we will create a method called calculateExecutionTime. The parameter type of this method will be ProceedingJoinPoint instead of JoinPoint, which we have used with methods marked with @Before and @After annotations. ProceedingJoinPoint allows the continuation of the execution. This method will return an Object that contains the values returned after the execution of the intercepted method call. The proceed method of ProceedingJoinPoint should either be surrounded by a **Try Catch** block or should include a throws declaration with the method definition.

The logic of this method is shown below:

public Object calculateExecutionTime(ProceedingJoinPoint joinPoint) throws Throwable {  
    //note start time  
    long startTime = System.currentTimeMillis();  
      
    //allow method call to execute  
    Object returnValue = joinPoint.proceed();  
      
    //time taken = end time - start time  
    long timeTaken = System.currentTimeMillis() - startTime;  
      
    logger.info("Time taken by {} to complete execution is: {}", joinPoint, timeTaken);  
    return returnValue;  
}

The variable startTime notes the time when the call is intercepted. The proceed method allows the intercepted method call to execute. The variable returnValue contains the values returned by the method. After the method call has returned, we calculate the execution time in a variable timeTaken.

1. Now, we will use the @Around annotation to define a pointcut for method calls for which we want the execution time to be tracked. If we want the time of all methods to be tracked, the following pointcut expression will be used:

@Around("execution(\* io.datajek.springaop.movierecommenderaop..\*.\*(..))")

When the application given in the code widget below is executed, it will print the execution time of every method.

MovieRecommenderAopApplication.java

ExecutionTimeAspect.java

LoggingAspect.java

AccessCheckAspect.java

FilteringTechnique1.java

FilteringTechnique2.java

Movie.java

User.java

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

package io.datajek.springaop.movierecommenderaop.aspect;

import org.aspectj.lang.ProceedingJoinPoint;

import org.aspectj.lang.annotation.Around;

import org.aspectj.lang.annotation.Aspect;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.context.annotation.Configuration;

@Aspect

@Configuration

public class ExecutionTimeAspect {

    private Logger logger = LoggerFactory.getLogger(this.getClass());

    @Around("execution(\* io.datajek.springaop.movierecommenderaop..\*.\*(..))")

    public Object calculateExecutionTime(ProceedingJoinPoint joinPoint) throws Throwable {

        //note start time

        long startTime = System.currentTimeMillis();

        //allow method call to execute

        Object returnValue = joinPoint.proceed();

        //time taken = end time - start time

        long timeTaken = System.currentTimeMillis() - startTime;

        logger.info("Time taken by {} to complete execution is: {}", joinPoint, timeTaken);

        return returnValue;

    }

}





Run

Save

Reset

Back

**JoinPoint Configuration File**

This lesson focuses on storing all the pointcut expressions in a separate configuration file, which is considered best practice in AOP.

**We'll cover the following**

* [For a specific layer](https://www.educative.io/courses/guide-spring-5-spring-boot-2/m7q4zgE9nKG#For-a-specific-layer)
* [For multiple layers](https://www.educative.io/courses/guide-spring-5-spring-boot-2/m7q4zgE9nKG#For-multiple-layers)
* [For a bean](https://www.educative.io/courses/guide-spring-5-spring-boot-2/m7q4zgE9nKG#For-a-bean)

Creating aspects to intercept method calls and perform tasks before, after, or around the execution is a repetitive task that repeats the same pointcuts over and over again. In an application with a large number of aspects, this can become cumbersome.

A best practice related to AOP is to have a separate configuration file that defines all the pointcuts. This way, all the pointcut definitions will be in one place. Hence they will be easy to manage. We can then use the definitions in any aspect.

**For a specific layer**

1. We will create a class called JoinPointConfig and define pointcuts using the @pointcut annotation. This annotation will be used on an empty method as follows:

@Pointcut("execution(\* io.datajek.springaop.movierecommenderaop.data.\*.\*(..))")  
public void dataLayerPointcut() {}

Now, we can use the dataLayerPointcut method by providing its fully qualified name wherever we want to intercept execution of methods in the data layer.

1. Previously, we had been using pointcuts as follows:

@AfterReturning("execution(\* io.datajek.springaop.movierecommenderaop.data.\*.\*(..))")  
public void logAfterExecution(JoinPoint joinPoint) {  
    //...  
}

Now, we will use the method that defines this pointcut in the configuration file as follows:

@AfterReturning("io.datajek.springaop.movierecommenderaop.aspect.JoinPointConfig.dataLayerPointcut()")    
public void logAfterExecution(JoinPoint joinPoint) {  
    //...  
}

1. In the same manner, we can create a pointcut configuration for intercepting method calls in the business layer as follows:

@Pointcut("execution(\* io.datajek.springaop.movierecommenderaop.business.\*.\*(..))")  
public void businessLayerPointcut() {}

The fully qualified name of businessLayerPointcut can be used in place of the pointcut definition in AccessCheckAspect without affecting the output as follows:

@Before("io.datajek.springaop.movierecommenderaop.aspect.JoinPointConfig.businessLayerPointcut()")  
public void before(JoinPoint joinPoint) {  
    //. . .  
}

**For multiple layers**

1. We can also combine pointcuts using the AND (&&), (OR) ||, and (NOT) ! operators. The method allLayerPointcut will intercept calls belonging to either the business layer or the data layer.

@Pointcut("io.datajek.springaop.movierecommenderaop.aspect.JoinPointConfig.dataLayerPointcut() || "  
      + "io.datajek.springaop.movierecommenderaop.aspect.JoinPointConfig.businessLayerPointcut()")  
public void allLayersPointcut() {}

When this pointcut is used in **AccessCheckAspect**, four method calls are intercepted.

**For a bean**

1. We can also define a pointcut to intercept calls belonging to a particular bean. Say we want to log the execution of all methods belonging to beans that have the word *Movie* in their name. We can define a pointcut as follows:

@Pointcut("bean(movie\*)")  
    public void movieBeanPointcut() {}

When this pointcut is used in AccessCheckAspect, it will intercept calls from the Movie and MovieRecommenderAopApplication beans.

MovieRecommenderAopApplication.java

JoinPointConfig.java

ExecutionTimeAspect.java

LoggingAspect.java

AccessCheckAspect.java

FilteringTechnique1.java

FilteringTechnique2.java

Movie.java

User.java

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

package io.datajek.springaop.movierecommenderaop.aspect;

import org.aspectj.lang.annotation.Pointcut;

public class JoinPointConfig {

    @Pointcut("execution(\* io.datajek.springaop.movierecommenderaop.data.\*.\*(..))")

    public void dataLayerPointcut() {}

    @Pointcut("execution(\* io.datajek.springaop.movierecommenderaop.business.\*.\*(..))")

    public void businessLayerPointcut() {}

    //to intercept method calls for both layers:

    @Pointcut("io.datajek.springaop.movierecommenderaop.aspect.JoinPointConfig.dataLayerPointcut() || "

            + "io.datajek.springaop.movierecommenderaop.aspect.JoinPointConfig.businessLayerPointcut()")

    public void allLayersPointcut() {}

    //for a particular bean

    @Pointcut("bean(movie\*)")

    public void movieBeanPointcut() {}

}





Run

Save

Reset

Back

**Defining a Custom Annotation for Aspects**

Learn how to create your own annotation and use it in aspects to intercept method calls.

An aspect can be used to track the execution time of a method which intercepts calls to a particular layer. Another approach could be to define a custom annotation on any method for which we want to track the execution time. spring-aop allows us to create our own annotations and define aspects to implement them.

1. Suppose we want to call our annotation @MeasureTime. We will create this annotation in the same folder as the other aspects. This creates an interface:

package io.datajek.springaop.movierecommenderaop.aspect;  
  
public @interface MeasureTime {  
  
}

1. We will restrict the use of this annotation to methods only. This can be achieved using the @Target annotation, with ElementType set to METHOD. Other options include class, package, field, etc.

@Target(ElementType.METHOD)

We would also like the annotation information to be available at runtime. We will use the @Retention annotation to define a retention policy as follows:

@Retention(RetentionPolicy.RUNTIME)

1. Now that we have defined our annotation, we can add it to the JoinPointConfig file and create a pointcut as follows:

@Pointcut("@annotation(io.datajek.springaop.movierecommenderaop.aspect.MeasureTime)")  
public void measureTimeAnnotation() {}

We can now use this pointcut to calculate the execution time of only chosen methods. Previously, the ExecutionTimeAspect looked like this:

public class ExecutionTimeAspect {  
    private Logger logger = LoggerFactory.getLogger(this.getClass());  
  
    @Around("io.datajek.springaop.movierecommenderaop.aspect.JoinPointConfig.businessLayerPointcut()")  
    public Object calculateExecutionTime(ProceedingJoinPoint joinPoint) throws Throwable {  
    //...  
    }  
}

Instead of tracking the time of business layer methods, we will intercept methods using the @MeasureTime annotation:

public class ExecutionTimeAspect {  
    private Logger logger = LoggerFactory.getLogger(this.getClass());  
    @Around("io.datajek.springaop.movierecommenderaop.aspect.JoinPointConfig.measureTimeAnnotation()")  
    public Object calculateExecutionTime(ProceedingJoinPoint joinPoint) throws Throwable {  
        //...  
    }  
}

Since we haven’t used the annotation anywhere in the code yet, no call will be intercepted. Suppose we want to track the time for the contentBasedFiltering method of the FilteringTechnique1 class and the getMovieDetails method of the Movie class. We will use our custom annotation on these methods:

@MeasureTime  
public String contentBasedFiltering() {  
    String movieDetails = movie.getMovieDetails();  
    return movieDetails;  
}

@MeasureTime  
public String getMovieDetails() {  
    return "movie details";       
}

If the application is run after adding @MeasureTime, calls to both these methods will be intercepted and the execution time will be measured.

MovieRecommenderAopApplication.java

MeasureTime.java

JoinPointConfig.java

AccessCheckAspect.java

LoggingAspect.java

ExecutionTimeAspect.java

FilteringTechnique1.java

FilteringTechnique2.java

Movie.java

User.java

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

package io.datajek.springaop.movierecommenderaop.aspect;

import org.aspectj.lang.ProceedingJoinPoint;

import org.aspectj.lang.annotation.Around;

import org.aspectj.lang.annotation.Aspect;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.context.annotation.Configuration;

@Aspect

@Configuration

public class ExecutionTimeAspect {

    private Logger logger = LoggerFactory.getLogger(this.getClass());

    @Around("io.datajek.springaop.movierecommenderaop.aspect.JoinPointConfig.measureTimeAnnotation()")

    public Object calculateExecutionTime(ProceedingJoinPoint joinPoint) throws Throwable {

        //note start time

        long startTime = System.currentTimeMillis();

        //allow method call to execute

        Object returnValue = joinPoint.proceed();

        //time taken = end time - start time

        long timeTaken = System.currentTimeMillis() - startTime;

        logger.info("\n\n>>Time taken by {} \nto complete execution is: {}\n", joinPoint, timeTaken);

        return returnValue;

    }

}





Run

Save

Reset

Back

# Spring MVC

**Setting Up a Web Application**

The aim of this lesson is to create a simple web application that runs using the Tomcat server.

**We'll cover the following**

* [Creating a simple Maven project](https://www.educative.io/courses/guide-spring-5-spring-boot-2/g20z1QoV2Aj#Creating-a-simple-Maven-project)
* [Java EE Web API dependency](https://www.educative.io/courses/guide-spring-5-spring-boot-2/g20z1QoV2Aj#Java-EE-Web-API-dependency)
* [Plugins for Maven compiler and Tomcat](https://www.educative.io/courses/guide-spring-5-spring-boot-2/g20z1QoV2Aj#Plugins-for-Maven-compiler-and-Tomcat)
* [web.xml file](https://www.educative.io/courses/guide-spring-5-spring-boot-2/g20z1QoV2Aj#web.xml-file)

**Creating a simple Maven project**

We will create a web application using Maven as it makes development easier. Maven is a Java build tool which manages all the dependencies of an application. Create a simple Maven project using the IDE of your choice. Maven provides a set of sample projects which are called *archetypes* with predefined configuration.

We will skip the archetype selection for this project as we want to configure the application ourselves. The name of a Maven project has two parts, group Id and artifact Id. We have supplied the following values:

Group id: io.datajek.springmvc

Artifact id: tennis-player-web

Packaging: war (web archive)

The packaging type is set to WAR because we are creating a web project. A WAR file contains servlet, JSP, XML, CSS, HTML and JS files that can be deployed on a servlet container while a JAR file contains Java classes and associated metadata as a single file.

Once the project is created, we can see a hierarchy of folders.

* The source code is placed in **src/main/java**.
* Any resources like a property file or XML file goes in **src/main/resources**.
* All test code is placed in **src/test/java**.
* Any resources for the test code reside in **src/test/resources**.

**Java EE Web API dependency**

The **pom.xml** shows the basic information about the group Id and artifact Id that we provided. To run a web application, we need a number of dependencies or jars. Maven downloads those jars, saving us the time to manually download them. We will add the Java EE Web API dependency to the **pom.xml** file as follows:

<dependencies>  
 <dependency>  
     <groupId>javax</groupId>  
     <artifactId>javaee-web-api</artifactId>  
    <version>6.0</version>  
    <scope>provided</scope>  
 </dependency>  
</dependencies>

We will run a Java EE 6 application where servlets extend the **HttpServlet** class. We need to provide the jar to the compiler where **HttpServlet** is defined. The **javaee-web-api** dependency added above downloads the API containing the **HttpServlet** among other jars.

**Plugins for Maven compiler and Tomcat**

To compile the application, we will add the **maven-complier-plugin**. This will take care of compiling the Java classes and building the jars and wars using version 1.8. To be able to run the web application in Tomcat, we will add the **tomcat7-maven-plugin**. This plugin downloads Tomcat and runs the web application in it. The plugins are added to the **pom.xml** file as shown below:

<build>  
  <pluginManagement>  
    <plugins>  
      <plugin>    
        <groupId>org.apache.maven.plugins</groupId>  
        <artifactId>maven-compiler-plugin</artifactId>  
        <version>3.2</version>  
        <configuration>  
            <verbose>true</verbose>  
            <source>1.8</source>  
            <target>1.8</target>  
            <showWarnings>true</showWarnings>  
        </configuration>  
      </plugin>  
      <plugin>  
        <groupId>org.apache.tomcat.maven</groupId>  
        <artifactId>tomcat7-maven-plugin</artifactId>  
        <version>2.2</version>  
        <configuration>  
            <path>/</path>  
            <contextReloadable>true</contextReloadable>  
        </configuration>  
      </plugin>  
    </plugins>  
  </pluginManagement>  
</build>

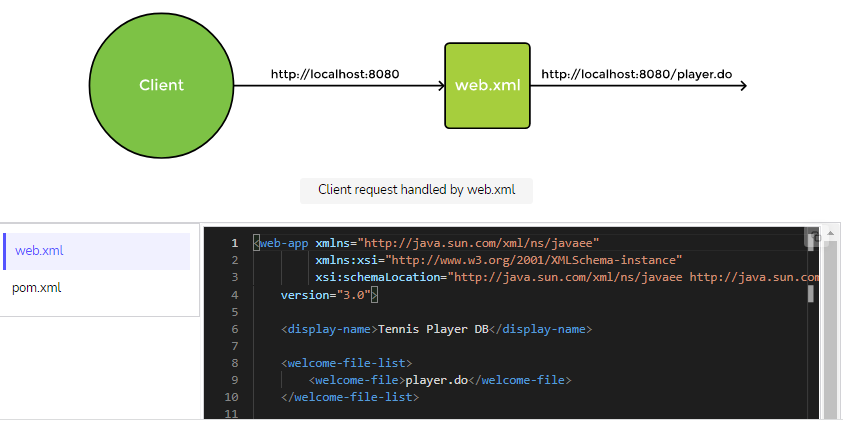
**web.xml file**

**web.xml** is the starting point for any web application. This is where Tomcat, or any other Java EE implementation server, starts looking for servlets. We will create this file in **src/main/webapp/WEB-INF/** folder. **web.xml** file contains header information for the application (shown in the widget below). After the metadata, we will define the landing page for our application using the <welcome-file> tag. A welcome file is automatically invoked by the server if no other file name is specified. We will call our welcome file, **[player.do](http://player.do/" \t "_blank)**.The following line defines a redirection for **localhost:8080** as **localhost:8080/player.do**:

<welcome-file-list>  
    <welcome-file>player.do</welcome-file>  
</welcome-file-list>

The complete **web.xml** file is shown in the code widget at the end of the lesson.

The figure below shows how an HTTP request from the client is handled by **web.xml**.



Client request handled by web.xml

web.xml

pom.xml

1

2

3

4

5

6

7

8

9

10

11

12

<web-app xmlns="http://java.sun.com/xml/ns/javaee"

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

         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd"

    version="3.0">

    <display-name>Tennis Player DB</display-name>

    <welcome-file-list>

        <welcome-file>player.do</welcome-file>

    </welcome-file-list>

</web-app>

# Creating a Servlet

Let’s create a servlet to map requests coming to localhost:8080.

**We'll cover the following**

* [HttpServlet class](https://www.educative.io/courses/guide-spring-5-spring-boot-2/BnpJloJOp3J#HttpServlet-class)
* [Mapping URL to servlet](https://www.educative.io/courses/guide-spring-5-spring-boot-2/BnpJloJOp3J#Mapping-URL-to-servlet)
* [Handling a GET request](https://www.educative.io/courses/guide-spring-5-spring-boot-2/BnpJloJOp3J#Handling-a-GET-request)
* [How to run the web application](https://www.educative.io/courses/guide-spring-5-spring-boot-2/BnpJloJOp3J#How-to-run-the-web-application)

A servlet is a Java class which can take in a request and return a response. The input to the servlet is a request and the output is a response. In a web application, we are talking about HTTP requests and HTTP responses. When something is typed in the address bar of a browser, it sends a request to the web server. The web server returns a response based on the request, and the browser displays the page.

So far, we have created a **web.xml** file that defines a welcome file, [**player.do**](http://player.do/), for our web application. When http://localhost:8080 is typed in the browser, the browser creates a GET request and sends it to http://localhost:8080/player.do. We want this request to be mapped to a servlet which sends some response back to the browser.

## HttpServlet class

1. We will create a servlet class in **src/main/java** and call it PlayerServlet. A servlet class extends the HttpServlet class defined in javax.servlet.http.HttpServlet.

import javax.servlet.http.HttpServlet  
  
public class PlayerServlet extends HttpServlet {  
  
}

## Mapping URL to servlet

1. The next step is to define a URL that will be used to access the servlet. We will assign /player.do to PlayerServlet. This can be done using the @WebServlet annotation. The urlPatterns property can be used to assign /player.do to our servlet as follows:

import javax.servlet.annotation.WebServlet;  
  
@WebServlet(urlPatterns = "/player.do")  
public class PlayerServlet extends HttpServlet {  
  
}

Now the player servlet can be called when http://localhost:8080/player.do is typed.

Client request mapped to PlayerServlet

Spring Boot offers another annotation, @ServletComponentScanto register classes annotated with @WebServlet, @WebFilter and @WebListener.

In case the embedded server fails to scan the servlet defined using @WebServlet, then @ServletComponentScan can be used on the application class.

## Handling a GET request

1. Our servlet class will handle HTTP requests and respond with an HTTP response. The HttpServlet class has several methods to handle requests. We will write the implementation of the doGet method. This method has two parameters, HttpServletRequest and HttpServletResponse.

import javax.servlet.http.HttpServletRequest;  
import javax.servlet.http.HttpServletResponse;  
  
@WebServlet(urlPatterns = "/player.do")  
public class PlayerServlet extends HttpServlet {  
  
    @Override  
    protected void doGet(HttpServletRequest request, HttpServletResponse response)   
                                                                throws IOException {  
        //...  
    }  
}

This method defines the response to a **GET** request for http://localhost:8080. The input to this method is request, and the output is response. We will read the input provided by the client from the request and put information to be shown to the client in the response.

First, we show how to send a simple HTML response back using the PrintWriter object in the code snippet below:

protected void doGet(HttpServletRequest request, HttpServletResponse response)   
                                                            throws IOException {  
    PrintWriter out = response.getWriter();  
    out.println("<html>");  
    out.println("<head>" +  
             "<title>Player DB</title>" +  
             "</head>");  
    out.println("<body>" +  
             "<H2>Welcome to the Tennis Players database!</H2>" +  
             "</body>");  
    out.println("</html>");  
}

The println method of the PrintWriter object is used to define HTML, which sets the title of the page and displays some text in the page body.

The image below shows what we have achieved so far. The PlayerServlet gets an HTTP request and sends an HTTP response back. The request/response cycle is shown below:

PlayerServlet receives HTTP request and sends HTTP response back

## How to run the web application

1. We have modified the **pom.xml** file, added a **web.xml** file, and created a servlet. To complete the configuration of our web application and remove any compilation errors, update the project as **Maven->Update Project** in your IDE.

Try the code given below:

###### /

PlayerServlet.java

web.xml

**PlayerServlet.java**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

package io.datajek.jee;

import java.io.IOException;

import java.io.PrintWriter;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

@WebServlet(urlPatterns = "/player.do")

public class PlayerServlet extends HttpServlet {

    @Override

    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException {

        PrintWriter writer = response.getWriter();

        writer.println("<html>");

        writer.println("<head>" +

                       "<title>Player DB</title>" +

                       "</head>");

        writer.println("<body>" +

                       "<H2>Welcome to the Tennis Players database!</H2>" +

                       "</body>");

        writer.println("</html>");

    }

}





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/player.do>

In your IDE, to run the application as Maven Build, specify **tomcat7:run** as goal. After Tomcat starts, you can see Running war on [*http://localhost:8080/*](http://localhost:8080/).

Type [**http://localhost:8080/player.do**](http://localhost:8080/player.do) in the browser. On this link, the PlayerServlet opens up and the contents returned by the doGet method are displayed.

Be sure to kill the server before attempting to re-run the application to avoid the Address already in use exception.

# Using a JSP File

Instead of writing static HTML content in a servlet, let’s learn how to generate dynamic content using a JSP file.

**We'll cover the following**

* [Creating a JSP file](https://www.educative.io/courses/guide-spring-5-spring-boot-2/3wAnP8pwLOR#Creating-a-JSP-file)
* [Redirecting GET request to JSP file](https://www.educative.io/courses/guide-spring-5-spring-boot-2/3wAnP8pwLOR#Redirecting-GET-request-to-JSP-file)

Writing HTML in Java is not good practice. We have to use the println method for every line of HTML code. A better approach is to use **Java Server Pages (JSP)** to write HTML content to the browser. A JSP is also compiled as a servlet so there is no performance gain in using JSPs. Rather, it is ease to use and able to send dynamic content to the browser.

## Creating a JSP file

1. Typically, JSPs reside in the views folder in **src/main/webapp/WEB-INF**. Right now, we only have **webapp** in the project hierarchy. Inside the **webapp** folder, we will create a folder **WEB-INF** which will have a subfolder, **views**.

In the **views** folder, we will create a JSP file and call it **welcome.jsp**. By default, this generates an HTML 5 JSP. This file provides the basic HTML structure of a webpage.

<!DOCTYPE html>  
<html>  
<head>  
<meta charset="ISO-8859-1">  
<title>Insert title here</title>  
</head>  
<body>  
  
</body>  
</html>

We can now provide the title and content for the body of the page:

<html>  
<head>  
<meta charset="ISO-8859-1">  
<title>Tennis Player DB</title>  
</head>  
<body>  
<h2>Welcome to the tennis player database!</h2>  
</body>  
</html>

Now that the JSP is ready, we want a GET request to the PlayerServlet to redirect to this JSP.

## Redirecting GET request to JSP file

1. To call this JSP file in the doGet method, we will use the getRequestDispatcher method, provide the path of the file, and then use the forward method to forward the request and response to the JSP as shown in the snippet below.

The path of the JSP file starts from the WEB-INF folder.

@Override  
protected void doGet(HttpServletRequest request, HttpServletResponse response)   
                     throws IOException, ServletException {  
    request.getRequestDispatcher("/WEB-INF/views/welcome.jsp").forward(request, response);  
}

This will load the welcome JSP when http://localhost:8080 is typed in the browser.

The request response cycle is shown below. The client sends a request (http://localhost:8080/player.do) to the web server (Tomcat). The web server maps the request to a servlet (PlayerServlet) and retrieves a page from the files (welcome.jsp). This page is sent as a response to the client. The browser interprets the response and displays the web page.

###### /

PlayerServlet.java

web.xml

welcome.jsp

**PlayerServlet.java**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

package io.datajek.jee;

import java.io.IOException;

import java.io.PrintWriter;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

@WebServlet(urlPatterns = "/player.do")

public class PlayerServlet extends HttpServlet {

    @Override

    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException, ServletException {

        request.getRequestDispatcher("/WEB-INF/views/welcome.jsp").forward(request, response);

    }

}





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/player.do>

The content of the JSP file is a bit different from the HTML we used in the previous lesson. This is to eliminate any doubt that the content is in fact coming from the JSP.

# Passing Parameters with GET request

Let’s explore how to pass parameters from the browser to the servlet with a GET request.

**We'll cover the following**

* [Passing a parameter in URL](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVjDlgRXBp3#Passing-a-parameter-in-URL)
* [Receiving parameter in servlet](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVjDlgRXBp3#Receiving-parameter-in-servlet)
* [Passing the attribute to JSP](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVjDlgRXBp3#Passing-the-attribute-to-JSP)
* [Displaying the attribute in JSP](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVjDlgRXBp3#Displaying-the-attribute-in-JSP)

A parameter is something that is passed between the browser and web server.

## Passing a parameter in URL

1. Suppose we want to pass the name of a tennis player as a parameter. The way to pass a parameter is to put a question mark at the end of the URL and to provide the name of the parameter. The **?** separates the URL from the parameter:

http://localhost:8080?name=sampras

Here, we are passing a parameter called name. The value of the parameter is sampras.

To pass multiple parameters, the **&** symbol is used as follows:

http://localhost:8080/?fname=novak&lname=djokovic

The above snippet shows how to pass two parameters, fname and lname, in the URL string.

## Receiving parameter in servlet

1. To receive the parameter passed from the browser in the doGet method of the PlayerServlet class, we will use the getParameter method and provide the name of the parameter that we wish to receive.

String playerName = request.getParameter("name");

We are storing the parameter in the String variable playerName.

## Passing the attribute to JSP

1. Now that our servlet has received the parameter, the next step is to forward it to the JSP. A parameter is what is passed between the browser and the web server. An attribute is what gets passed from the web server to the JSP.

We will use the setAttribute method to pass the received parameter as an attribute to the JSP. The method has two arguments: the name of the attribute and its value.

request.setAttribute("name", playerName);  
request.getRequestDispatcher("/WEB-INF/views/welcome.jsp").forward(request, response);

Anything that is set as the request attribute will be available to the JSP. In the request, we are setting the parameter name as an attribute and passing the variable playerName as the value of the attribute. After setting the attribute, the request is forwarded to the JSP and the attribute will be available for use in welcome.jsp.

## Displaying the attribute in JSP

1. To access the attribute in the JSP, we will use its name (name). The value of the attribute can be displayed in expression language as follows:

<body>  
<h2>Welcome to the tennis player database!</h2>  
<h3>Player name: <i>${name}</i> </h3>  
  
</body>

${attribute\_name} is expression language syntax in which the attribute\_name is replaced by the value of the attribute. Expression language is used to dynamically pick up content from the request.

When the code given below is run, you will see that no player name is displayed. To pass a name as a parameter in the URL, open the link for the app given below the code and use the syntax explained in point #1 for passing parameters. For example, http://localhost:8080/player.do?name=novak.

###### /

PlayerServlet.java

web.xml

welcome.jsp

**PlayerServlet.java**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

package io.datajek.jee;

import java.io.IOException;

import java.io.PrintWriter;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

@WebServlet(urlPatterns = "/player.do")

public class PlayerServlet extends HttpServlet {

    @Override

    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException, ServletException {

        String playerName = request.getParameter("name");

        System.out.println(playerName);

        request.setAttribute("name", playerName);

        request.getRequestDispatcher("/WEB-INF/views/welcome.jsp").forward(request, response);

    }

}





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/player.do?name=sampras>

In a GET request, the only way to pass parameters is through the URL, which poses a security hazard. Passing sensitive information as a parameter is not recommended as the request passes through a number of routers and any router can access the information in the URL string, thus making it insecure.

# Scriptlets and Scriptlet Expressions

Learn about scriptlet, scriptlet expression, and expression language.

**We'll cover the following**

* [Scriptlet](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JE1xNgA5BlD#Scriptlet)
* [Scriptlet expression](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JE1xNgA5BlD#Scriptlet-expression)

A JSP is compiled into a servlet, which means that anything that can be done in a servlet can also be done in a JSP. In PlayerServlet, we wrote Java code, while in welcome.jsp, we wrote HTML. Just as HTML can be written in the PlayerServlet class, Java code can be written in welcome.jsp. Java code in a JSP is called a scriptlet.

A disclaimer before we show how to write a scriptlet: this is not recommended practice and should be avoided.

## Scriptlet

1. In a JSP, anything written between the <% %> tags is a scriptlet. We can print something on the console from welcome.jsp as follows:

<body>  
<%  
System.out.println("Hello from JSP");  
%>  
  
</body>

When the JSP is executed, this message will be displayed on the console.

1. Any Java code can be written in a scriptlet. We can output the name of the parameter using the request.getParameter method, use loops, or get the current system date and time. Let’s say we want to print the date in the JSP. Import java.time.LocalDate in the JSP using @page as shown below:

<%@page import="java.time.LocalDate"%>

Now we can access the now method to get the date as follows:

<%  
LocalDate currentDate = LocalDate.now();  
%>

## Scriptlet expression

1. To show the value of a Java variable in HTML, we will use a scriptlet expression. A scriptlet expression uses the <%= variable\_name %> syntax. We can show the currentDate variable in HTML as follows:

<p>Current Date: <%= currentDate %> </p>  
<p>Today is <%= currentDate.getDayOfWeek() %> </p>

We can also call methods on the variable in a scriptlet expression. The second line, in the code snippet above calls the getDayOfWeek method on the currentDate variable.

###### /

PlayerServlet.java

web.xml

welcome.jsp

**welcome.jsp**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

<%@page import="java.time.LocalDate"%>

<%@ page language="java" contentType="text/html; charset=ISO-8859-1"

    pageEncoding="ISO-8859-1"%>

<!DOCTYPE html>

<html>

<head>

<meta charset="ISO-8859-1">

<title>Tennis Player DB</title>

</head>

<body>

<%

System.out.println("Hello from JSP");

LocalDate currentDate = LocalDate.now();

%>

<p><i>Current Date: <%= currentDate %> </i></p>

<p><i>Today is <%= currentDate.getDayOfWeek() %> </i></p>

<h2>Welcome to the tennis player database!</h2>

<h3>Player name: <i>${name}</i> </h3>

</body>

</html>





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/player.do>

To display a player name, it needs to be passed in the URL. Open the link for the app and pass a parameter as demonstrated in the last lesson.

As we have demonstrated, it is possible to write Java code in a JSP, but this is not recommended. A JSP is a view and views should only be used to display information. When we start writing Java code in a JSP, there is a chance that we may write business logic there. However, business logic belongs in the business layer and should be written in a Java class. This is why using scriptlets and scriptlet expressions should be avoided.

# POST request

Learn about the POST request and create a form to pass parameters to the web server.

**We'll cover the following**

* [Creating a form](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JPPBWZy8kLv#Creating-a-form)
* [Text box input element](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JPPBWZy8kLv#Text-box-input-element)
* [Submit button](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JPPBWZy8kLv#Submit-button)
* [Creating a POST request](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JPPBWZy8kLv#Creating-a-POST-request)
* [Handling a POST request](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JPPBWZy8kLv#Handling-a-POST-request)
* [Forwarding the parameter as attribute](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JPPBWZy8kLv#Forwarding-the-parameter-as-attribute)
* [Calling a business service method](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JPPBWZy8kLv#Calling-a-business-service-method)

In a GET request, parameters are passed as a query string in the URL. The routers and ISPs can see the parameters and read values as they are passed in a GET request. POST request is another type of HTTP request which is more secure. In this lesson, we will pass parameters through a form using a POST request. The POST request is not completely secure, but it is better than a GET request. Name of a player can be passed as part of the URL as well as part of the form. Now we will pass it through a form.

## Creating a form

1. The first step is to create a form which will have fields for input parameters. We will create the form in welcome.jsp. Forms can be created using the <form> tag. We need to specify who will handle the information in the form. Since we want the PlayerServlet to handle this request, we will specify player.do as action. In our application, any request to player.do is handled by the PlayerServlet:

<form action="/player.do">  
  
</form>

## Text box input element

1. Next, we will create an input element of text type and specify the name of the parameter.

<form action="/player.do">  
    <input type="text" name="name"/>  
</form>>

This will create a text box and the value entered in the text box is stored in the name parameter.

## Submit button

1. To pass the parameter to the web server, we need a submit button. To create a submit button, another input element with type as submit is needed. The code for the form, after adding a submit button, is shown below:

<form action="/player.do">  
  <b>Player name:</b> <input type="text" name="name"/>  
                      <input type="submit" value="Enter"/>  
</form>>

By default, the text on the button is Submit. value can be used to show custom text. We have chosen the word Enter to be shown on the submit button.

## Creating a POST request

1. If you type a name in the text box and press Enter, the parameter appears in the URL. This shows that even though we are using a form, the request is still a GET request. To change it to a POST request, we will have to specify the method in the form element.

<form action="/player.do" method="POST">

This will send the parameter as part of form data instead of in the URL.

## Handling a POST request

1. We will handle the POST request created above in the PlayerServlet using the doPost method. A POST request without a handler results in a 405 error that says, “HTTP method POST is not supported by this URL”.

Right now, the PlayerServlet has a doGet method which gets the name parameter, sets it as an attribute, and then forwards it to the JSP. This functionality is not needed anymore. We only want the doGet method redirect to welcome.jsp. This is the JSP that contains the form created in the steps above.

@Override  
protected void doGet(HttpServletRequest request, HttpServletResponse response)   
                                                    throws IOException, ServletException {  
    request.getRequestDispatcher("/WEB-INF/views/welcome.jsp").forward(request, response);  
}

The above doGet method redirects a request for http://localhost:8080 to welcome.jsp where the user can enter a player name using a form. We will create a doPost method to handle the POST request from the form. The doPost method signature is similar to the doGet method. When the user enters a player name in the form and presses Enter, we want the user to be directed to a new page called info.jsp.

@Override  
protected void doPost(HttpServletRequest request, HttpServletResponse response)   
                                                 throws IOException, ServletException {  
    request.getRequestDispatcher("/WEB-INF/views/info.jsp").forward(request, response);  
}

1. The doPost method redirects to info.jsp, which does not exist currently. We will create a simple JSP in WEB-INF/views as follows:

<body>  
<h3>Searching for player... </h3>  
</body>

For now, this JSP displays some text.

The image below shows what we have achieved so far. When a client types http://localhost:8080, it generates a GET request that is redirected by our servlet to welcome.jsp. This page shows a form with a submit button. When the button is clicked, it generates a POST request to our servlet, which is then redirected to info.jsp.

GET and POST requests

## Forwarding the parameter as attribute

1. If we want to forward the parameter name from the form to info.jsp, we can use getParameter and setAttribute methods on the request as follows:

protected void doPost(HttpServletRequest request, HttpServletResponse response)   
                                                throws IOException, ServletException {  
    String playerName = request.getParameter("name");  
    request.setAttribute("name", playerName);  
    request.getRequestDispatcher("/WEB-INF/views/info.jsp").forward(request, response);  
}

Here we are getting the parameter sent from the form in welcome.jsp and storing it in a variable playerName. Then we are setting a request attribute by the name name and providing its value as playerName. Lastly, we are forwarding the request to info.jsp. Now the attribute name is available to the JSP, we will use EL to display the attribute in info.jsp as follows:

<body>  
    <h3>Searching for ${name}... </h3>  
</body>

###### /

PlayerServlet.java

web.xml

welcome.jsp

info.jsp

**info.jsp**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

<%@ page language="java" contentType="text/html; charset=ISO-8859-1"

    pageEncoding="ISO-8859-1"%>

<!DOCTYPE html>

<html>

<head>

<meta charset="ISO-8859-1">

<title>Tennis Player DB</title>

</head>

<body>

<h3>Searching for <i> ${name} </i>... </h3>

</body>

</html>





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/player.do>

## Calling a business service method

1. Now that we have a form for taking the player name as input, let’s add some functionality to return information about the player chosen by the user. We will create a Player class with name, nationality, date of birth, and number of titles as members. We will create getters and a constructor for initializing a Player object.

public class Player {     
    private int id;  
    private String name;  
    private String nationality;  
    private Date birthDate;  
    private int titles;  
   
    //constructor  
  
    //getter methods  
}

1. Next, we will create a PlayerService class. For starters, we can create a static list of players in the PlayerService class instead of fetching the values from the database. We are hardcoding a list of tennis players as follows:

private List<Player> players = Arrays.asList(  
     new Player(1, "Djokovic", "Serbia", new Date(1987-05-22), 81),  
     new Player(2, "Monfils", "France", new Date(1986-07-01), 10),  
     new Player(3, "Isner", "USA", new Date(1985-04-26), 15)  
     );

The PlayerService class has a method to return a player based on the name. The method is called getPlayerByName, and it contains a lambda expression that filters the List of Players and checks if the name is equal to the one provided by the user. If it finds a match, it returns the Player object. We are returning the first Player matching the criterion.

public Player getPlayerByName(String name) {  
    return players.stream().filter(p -> p.getName().equals(name)).findFirst().get();  
}

1. Now we can call the above method in the PlayerServlet class to get details of a player by passing the input parameter playerName. To access the method getPlayerByName, we need an object of the PlayerService class. The modified doPost method is shown below:

private PlayerService service = new PlayerService();  
  
@Override     
protected void doPost(HttpServletRequest request, HttpServletResponse response)   
                                           throws IOException, ServletException {  
    String playerName = request.getParameter("name");  
    Player player = service.getPlayerByName(playerName);  
    request.setAttribute("name", playerName);  
    request.setAttribute("country", player.getNationality());  
    request.setAttribute("dob", player.getBirthDate());  
    request.setAttribute("titles", player.getTitles());  
    request.getRequestDispatcher("/WEB-INF/views/info.jsp").forward(request, response);  
}  
  
public Player getPlayerByName(String name) {  
   return players.stream().filter(p -> p.getName().equals(name)).findFirst().get();  
}

Here, the parameter name is being stored in a variable playerName. It is then passed as an argument to the getPlayerByName method. This method returns a Player object. Next, we set the name of the player, his nationality, birthdate, and number of titles as attributes and forward this request to info.jsp.

The purpose of the above code is just to show how the web server can call a business service. This functionality has been shown in the code widget below. The user can choose a player (from the hardcoded list from step 9) and the web server responds with his details.

###### /

PlayerServlet.java

PlayerService.java

Player.java

web.xml

welcome.jsp

info.jsp

**info.jsp**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

<%@ page language="java" contentType="text/html; charset=ISO-8859-1"

    pageEncoding="ISO-8859-1"%>

<!DOCTYPE html>

<html>

<head>

<meta charset="ISO-8859-1">

<title>Tennis Player DB</title>

</head>

<body>

<h3>Player info: </h3>

<p> Name: ${name}</p>

<p> DoB: ${dob}</p>

<p> Country: ${country}</p>

<p> Titles: ${titles}</p>

</body>

</html>





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/player.do>

**Spring MVC Architecture**

Learn about the Spring MVC architecture and how it differs from the Servlets and JSP model.

**We'll cover the following**

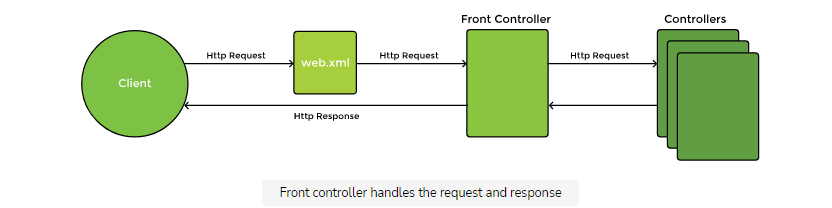
* [Spring MVC request flow](https://www.educative.io/courses/guide-spring-5-spring-boot-2/3wLlXEJEYGQ#Spring-MVC-request-flow)
* [Dispatcher servlet](https://www.educative.io/courses/guide-spring-5-spring-boot-2/3wLlXEJEYGQ#Dispatcher-servlet)
* [Advantages of Spring MVC](https://www.educative.io/courses/guide-spring-5-spring-boot-2/3wLlXEJEYGQ#Advantages-of-Spring-MVC)

Spring MVC is a framework for building web applications in Java which is based on the MVC design pattern. It uses the features of the core spring framework like Inversion of Control (IoC) and dependency injection.

**Spring MVC request flow**

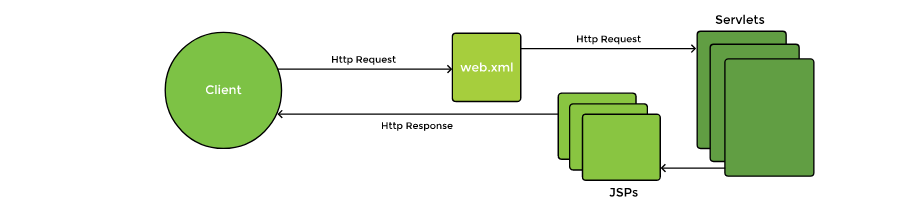
In a nutshell, a request coming from the client is handled by the Spring MVC front controller which delegates the request to another controller. The controller creates a model and based on the user request, populates the model. Then it returns the model to the front controller. Now, the front controller passes this model to the view template which takes the data and renders the response to the client.

MVC architecture has a **front controller** that manages all the other controllers. The developer creates the controllers for the application and Spring MVC provides the front controller. The job of the front controller is to handle all requests coming from the client. The **web.xml** file, which is the entry point of a web application, contains the servlet mapping of the front controller such that all requests are forwarded to it. It is the job of the front controller to call another controller based on the request that it receives.



Front controller handles the request and response

Compare this with the Java Servlet model, where **web.xml** maps each incoming request to a servlet and the servlet sends a response back to the client.

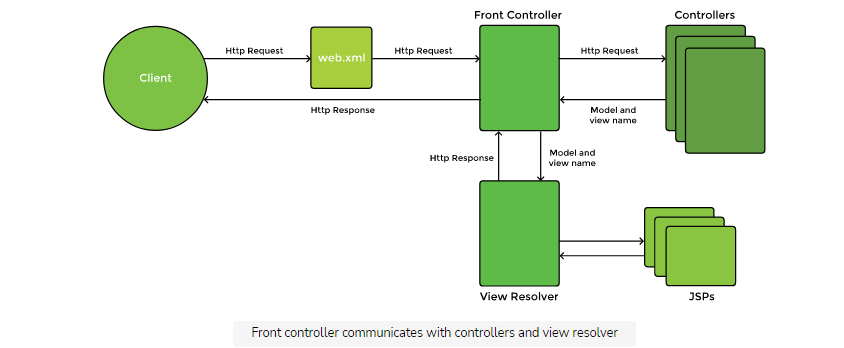


**Dispatcher servlet**

The front controller is called the **DispatcherServlet**. In order to know which controller to call, the dispatcher servlet needs a configuration file which contains information about where the controllers are placed. The @Controller annotation is used on all controllers and they are placed in the same package.

A request from the client goes to the **web.xml** as usual. In MVC architecture, **web.xml** forwards it to the dispatcher servlet. The dispatcher servlet checks its configuration file to find out which controller to invoke.

After the controller is executed, it sends two things back to the dispatcher servlet; the data or result (model) and the name of the page (view) to show to the client. The dispatcher servlet calls the view resolver to find the appropriate page, populates the page with the results and shows it to the client.



Front controller communicates with controllers and view resolver

As the above image shows, the dispatcher servlet manages everything in MVC architecture. The client need not know about the other controllers in the application. It only interacts with the dispatcher servlet.

**Advantages of Spring MVC**

Spring MVC offers many advantages. Some of them are listed below:

* Spring MVC provides a clear separation of concerns between the controllers, model and view objects.
* Spring MVC is flexible and does not force the developer to use JSP as the view technology. The developer is free to use Velocity, Thymeleaf or even design his own view mechanism and seamlessly integrate it. The controllers are not dependent on the view technology and can work independently even if the view technology is changed.
* Spring MVC application is testable as controllers are configured like other objects. The framework also supports Test Driven Development (TDD).
* The Spring tag library provides data binding and validation facility.
* Spring MVC uses WebApplicationContext to scope the beans to an HTTP Request or an HTTP Session. This helps in tracing the state of the web request.

**Spring MVC Components**

This lessons describes the three components of the Spring MVC architecture namely Model, View and Controller.

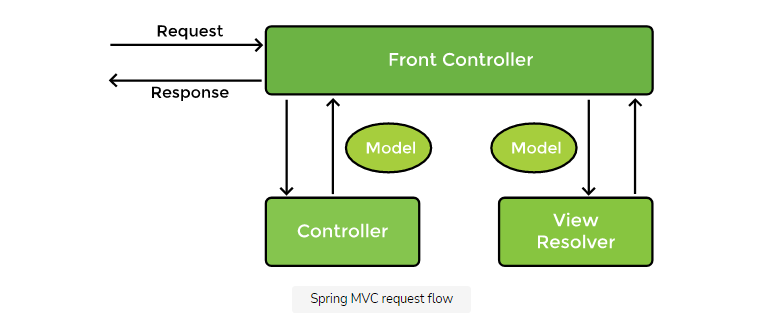
**We'll cover the following**

* [Spring MVC request flow](https://www.educative.io/courses/guide-spring-5-spring-boot-2/x1BvDxNEjLP#Spring-MVC-request-flow)
* [Controller](https://www.educative.io/courses/guide-spring-5-spring-boot-2/x1BvDxNEjLP#Controller)
* [Model](https://www.educative.io/courses/guide-spring-5-spring-boot-2/x1BvDxNEjLP#Model)
* [View template](https://www.educative.io/courses/guide-spring-5-spring-boot-2/x1BvDxNEjLP#View-template)

**Spring MVC request flow**

All incoming requests to a Spring MVC application are handled by the front controller called the DispatcherServlet. This front controller is part of the Spring JAR files and the developer is not required to implement it.

The front controller delegates an incoming request to other components of the application. The controllers contain the business logic, model contains the data, and views are the pages that render that data. The controller class populates the model and sends it to the front controller. The front controller forwards it to the view resolver which resolves the view that renders the model to the client. The big picture of the process is shown below:



Spring MVC request flow

**Controller**

The front controller maps the incoming request to a controller. Controllers contain the business logic of the application. They also handle requests and perform marshalling/ unmarshalling. The function of the controller is to handle a user request. The incoming request may be handled in different ways like reading form data, processing it in some way, storing data in a database, or retrieving data from a web service etc. The controller places the data in the model which, simply put, is a container for the data. The controller returns the model (containing data) back to the front controller.

**Model**

Model is a container for data. It is used to transfer data from one part of the Spring MVC application to another.

Controller populates the model with data from a form or a database or a web service. As can be seen from the Spring MVC request flow diagram, the controller passes the model to the front controller which forwards it to the view resolver. The view template displays the data in the model.

**View template**

Spring MVC supports a number of view templates. The most commonly used are JSP or JSTL (JSP Standard Tag Library). Other view templates like Thymeleaf, Groovy, Velocity, and FreeMarker etc., can also be plugged in.

View template receives the model containing data. It reads the model and displays the data. If, say, the model contains a list of players, the view template can create a table to display that list. In most cases a view template is a JSP page that provides data to the user.

**Configuring a Spring MVC Application - Part 1**

Learn how to configure a Spring MVC application.

**We'll cover the following**

* [Spring MVC dependency](https://www.educative.io/courses/guide-spring-5-spring-boot-2/3jlOw6NjYzx#Spring-MVC-dependency)
* [Define the dispatcher servlet](https://www.educative.io/courses/guide-spring-5-spring-boot-2/3jlOw6NjYzx#Define-the-dispatcher-servlet)
* [URL mapping of the dispatcher servlet](https://www.educative.io/courses/guide-spring-5-spring-boot-2/3jlOw6NjYzx#URL-mapping-of-the-dispatcher-servlet)

In the Spring MVC architecture, any request coming from the browser would go to the front controller (dispatcher servlet), which sends it to a controller or a JSP. To migrate the web application, we will add the Spring MVC dependency, configure the dispatcher servlet in **web.xml** and create a configuration file for the dispatcher servlet.

**Spring MVC dependency**

1. To use Spring MVC functionality in our application, add the following dependency to the **pom.xml** file:

<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-webmvc</artifactId>  
 <version>5.2.12.RELEASE</version>  
</dependency>

We are using version 5.2.12 as this is the latest version available at the time of writing this lesson. After saving the file, Maven downloads a number of dependencies which can be seen in the **Maven Dependencies** folder. By adding just the spring-webmvc dependency, Maven downloads transitive dependencies as well. These are the dependencies that spring-webmvc internally needs.

Note: If using Java 9, additional dependency **javax.xml.bind** is also needed, as this module has been deprecated.

<dependency>  
 <groupId>javax.xml.bind</groupId>  
 <artifactId>jaxb-api</artifactId>  
 <version>2.3.0</version>  
</dependency>

Next, we will create the configuration files for the Spring MVC application, namely **web.xml** and **player-servlet.xml**. These files will be placed in the **WEB-INF** directory.

**Define the dispatcher servlet**

1. A client request first goes to the front controller and then the front controller sends it to a servlet or JSP. The front controller is called a dispatcher servlet in MVC architecture. Maven downloaded the jar for dispatcher servlet when we added the spring-webmvc dependency.

To configure the dispatcher servlet, we will make some modifications to the **WEB-INF/web.xml** file. We will add an entry for the dispatcher servlet using the <servlet> tag and provide its name and class:

<servlet>  
 <servlet-name>dispatcher</servlet-name>  
 <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  
</servlet>

* + The name of our servlet is **dispatcher**. It can have any other name too.
  + The dispatcher servlet is part of the core Spring framework and is defined in **org.springframework.web.servlet.DispatcherServlet**.

1. The dispatcher servlet needs a configuration. We will provide the name and path of the configuration file using the <init-param> tag. The configuration file for the servlet is called **player-servlet** and it can be found in the **/WEB-INF/** folder.

<init-param>  
 <param-name>contextConfigLocation</param-name>  
 <param-value>/WEB-INF/player-servlet.xml</param-value>  
</init-param>  
<load-on-startup>1</load-on-startup>

This file can have any name. Here we are using **player-servlet.xml**. It will be loaded when the application starts.

**URL mapping of the dispatcher servlet**

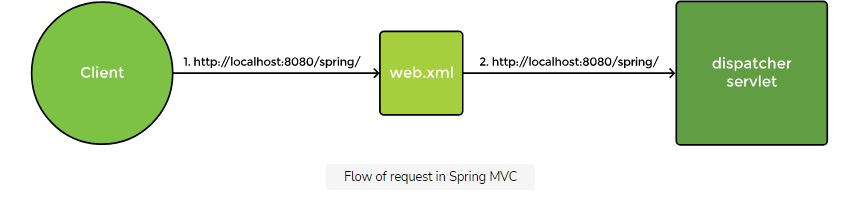
1. Every incoming request in Spring MVC application is handled by the dispatcher servlet. The <servlet-mapping> tag tells Spring which URL pattern should be passed off to the dispatcher servlet defined in step 2.

<servlet-mapping>  
 <servlet-name>dispatcher</servlet-name>  
 <url-pattern>/spring/\*</url-pattern>  
</servlet-mapping>

The servlet name provided here should match the name that we provided in step 2.

**/spring/\*** means that all requests starting with [**http://localhost:8080/spring/**](http://localhost:8080/spring/) should be mapped to the dispatcher servlet. We could also use “**/**” as the <url-pattern> to map all web requests starting with [**http://localhost:8080/**](http://localhost:8080/) to the dispatcher servlet.

The **web.xml** file will receive incoming requests and forward all requests starting with [**http://localhost:8080/spring/**](http://localhost:8080/spring/) to the dispatcher servlet.



The flow of request in Spring MVC is depicted below:

Flow of request in Spring MVC

The **pom.xml** and **web.xml** files are shown below:

web.xml

pom.xml

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

<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>

  <groupId>io.datajek.springmvc</groupId>

  <artifactId>tennis-player-web</artifactId>

  <version>0.0.1-SNAPSHOT</version>

  <packaging>war</packaging>

  <dependencies>

    <dependency>

        <groupId>javax</groupId>

        <artifactId>javaee-web-api</artifactId>

        <version>6.0</version>

        <scope>provided</scope>

    </dependency>

    <!-- For spring MVC -->

    <dependency>

        <groupId>org.springframework</groupId>

        <artifactId>spring-webmvc</artifactId>

        <version>5.2.12.RELEASE</version>

    </dependency>

  </dependencies>

    <build>

        <pluginManagement>

            <plugins>

                <plugin>

                    <groupId>org.apache.maven.plugins</groupId>

                    <artifactId>maven-compiler-plugin</artifactId>





We are half way through the configuration. In the next lesson, we will create the config file of the dispatcher servlet.

**Configuring a Spring MVC Application - Part 2**

..cont

**We'll cover the following**

* [Dispatcher servlet config file](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7D5GjrA5AoO#Dispatcher-servlet-config-file)
* [Component scanning](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7D5GjrA5AoO#Component-scanning)
* [Validation and type conversion](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7D5GjrA5AoO#Validation-and-type-conversion)
* [View resolver](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7D5GjrA5AoO#View-resolver)
* [Creating actual view name from logical name](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7D5GjrA5AoO#Creating-actual-view-name-from-logical-name)

In the last lesson, we defined the front controller called dispatcher for our Spring MVC application. We also mapped all requests starting with [**http://localhost:8080/spring/**](http://localhost:8080/spring/) to the dispatcher servlet. This servlet will map the incoming requests to different controllers as well as handle the response being sent back to the client. For this purpose, it needs a configuration file.

**Dispatcher servlet config file**

We will create the dispatcher servlet configuration file called **player-servlet.xml** in the **WEB-INF** directory because this is how we configured it in **web.xml** file, in the previous lesson:

<init-param>  
    <param-name>contextConfigLocation</param-name>  
    <param-value>/WEB-INF/player-servlet.xml</param-value>  
</init-param>

The dispatcher servlet config file will contain the following:

* Component scanning support
* Data validation and type conversion support
* View resolver configuration

1. Right click on the **WEB-INF** directory and create an XML file called **player-servlet.xml**. The file contains metadata defining namespaces as follows:

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:context="http://www.springframework.org/schema/context"  
 xmlns:mvc="http://www.springframework.org/schema/mvc"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans   
                    http://www.springframework.org/schema/beans/spring-beans.xsd  
                    http://www.springframework.org/schema/mvc   
                    http://www.springframework.org/schema/mvc/spring-mvc.xsd  
                    http://www.springframework.org/schema/context   
                    http://www.springframework.org/schema/context/spring-context.xsd">  
  
</beans>

**Component scanning**

1. In the config file, we can either define the beans using the <bean> tag or enable annotation based configuration and provide the path of the base package for component scanning. By enabling component scan, any Spring beans marked with @Component or its specialized annotations (like @Controller) will be added to the Spring context.

We will use the <context: component-scan> tag to scan the base package for any Spring beans as follows:

<context:component-scan base-package="io.datajek"/>

All controllers in our application will be defined in the package **io.datajek**, so this will be the base package for the component scan.

**Validation and type conversion**

1. Spring MVC provides support for JSR 303 Bean Validation API to validate form data. Spring MVC can also perform conversion of form data and format data based on the business logic. To get this support, we will use the <mvc:annotation-driven/> tag as follows:

<context:component-scan base-package="io.datajek"/>  
<mvc:annotation-driven/>

The <mvc:annotation-driven/> tag is used to register several Spring MVC infrastructure beans that dispatch requests to controllers using the @RequestMapping annotation, scan controllers for MVC annotations like @RequestParam, @ModelAttribute, and @InitBinder etc., configure the Web Data Binder and enable JSR 303 validation.

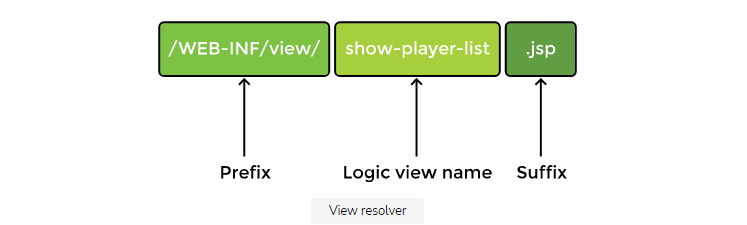
**View resolver**

1. The Spring MVC view resolver maps a view name to the actual view. It provides a layer of abstraction between the implementation and the view technology. Spring framework provides a number of built-in view resolvers. The most commonly used is **InternalResourceViewResolver**.

The following lines of code tell Spring where the pages are located and how to display them:

<bean  
     class ="org.springframework.web.servlet.view.InternalResourceViewResolver">  
     <property name="prefix" value="/WEB-INF/views/" />  
     <property name="suffix" value=".jsp" />  
</bean>

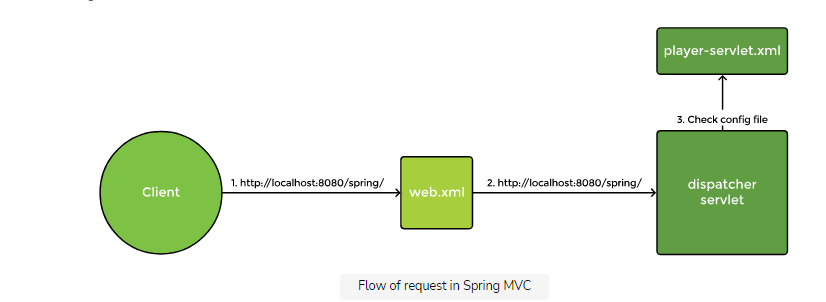
For a given view name, the view resolver will attach the given prefix and suffix. This will create the actual path of the file that is to be rendered as the response. For example, the application returned a view name **show-player-list**. Spring MVC view resolver will resolve it to **/WEB-INF/views/show-player-list.jsp** which is the actual path of the view page that will be rendered.



View resolver

**Creating actual view name from logical name**

The flow of a request in MVC is as follows:

1. **web.xml** receives all requests.
2. Anything starting with **/spring** will go to the front controller (dispatcher servlet).
3. **player-servlet.xml** is the configuration file of the front controller. It specifies where to perform a component scan to look for controllers.
4. 

Flow of request in Spring MVC

The dispatcher servlet config file is shown below:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

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

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

    xmlns:context="http://www.springframework.org/schema/context"

    xmlns:mvc="http://www.springframework.org/schema/mvc"

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

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

                        http://www.springframework.org/schema/beans/spring-beans.xsd

                        http://www.springframework.org/schema/mvc

                        http://www.springframework.org/schema/mvc/spring-mvc.xsd

                        http://www.springframework.org/schema/context

                        http://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="io.datajek"/>

    <mvc:annotation-driven/>

    <bean class = "org.springframework.web.servlet.view.InternalResourceViewResolver">

        <property name="prefix" value="/WEB-INF/views/" />

        <property name="suffix" value=".jsp" />

    </bean>

</beans>





Back

Configuring a Spring MVC Application - Part 1

Next

# Controller

This lesson demonstrates how to create a Controller class.

**We'll cover the following**

* [@Controller annotation](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qArEwA9mp23#@Controller-annotation)
* [@RequestMapping annotation](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qArEwA9mp23#@RequestMapping-annotation)
  + [@RequestMapping at method level](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qArEwA9mp23#@RequestMapping-at-method-level)
  + [@RequestMapping at class level](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qArEwA9mp23#@RequestMapping-at-class-level)
* [@ResponseBody annotation](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qArEwA9mp23#@ResponseBody-annotation)

Controllers are Java classes that map a URL and HTTP request to a method such that when a user goes to an address and sends a request, the controller sends back the appropriate response. Every controller is annotated with information about the URL that it handles and the HTTP methods that it supports.

A typical web application has a number of controllers, so when a request comes, Spring MVC examines the list of controllers to find a match for the URL and the incoming HTTP request. It then maps the request to a particular method of a controller. The controller returns a response which Spring MVC converts to an appropriate format. For a REST controller, the response is automatically converted to JSON.

We will now show how to create a controller for our application. The **TennisController** will handle a web request and display a response in the browser.

## @Controller annotation

1. Create a class, TennisController in the package **io.datajek.springmvc**. We have created a separate package to keep the MVC related code separate from the servlet code. When we created a servlet, it extended the **HttpServlet** class. In Spring MVC, we use the @Controller annotation to notify that the class is a controller. @Controller is a specialized form of @Component that supports web MVC. So, when Spring does a component scan, it also picks up classes marked with @Controller. By using this annotation, Spring will know that our class will handle web requests.

import   
org.springframework.stereotype.Controller;  
  
@Controller  
public class TennisController {  
  
}

1. Next, we will create a method named welcome in our controller class. A controller method can have any name. The return type is String which is used to return the name of the view or a String response that we will show to the client.

@Controller  
public class TennisController {  
  
    public String welcome() {  
        //...  
    }  
}

A controller method can have any number of parameters like request parameters, session objects, model objects etc. The return type can also vary.

## @RequestMapping annotation

### @RequestMapping at method level

The next step is to map a URL to the controller method. We want the URL **“/”** to be mapped to our method. So, when a user types [**http://localhost:8080/spring/**](http://localhost:8080/spring/), we want the welcome method to execute and display a welcome message.

To map a request from **“/”** to the welcome method, the @RequestMapping annotation is used as follows:

import org.springframework.web.bind.annotation.RequestMapping;  
  
@Controller  
public class TennisController {  
  
 @RequestMapping(value = "/")  
 public String welcome() {  
     return "Welcome to the Tennis Player database from Spring MVC!";  
}

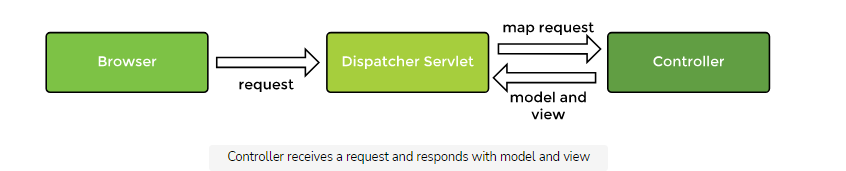
Now, the welcome method will execute when a request for **“/”** is received and will return a string message.

We can have multiple URLs that map to the same controller meaning that a controller can have multiple methods for mapping different URLs and HTTP requests.

### @RequestMapping at class level

The @RequestMapping at method level is used to map a URL to a controller method. This annotation can also be used on classes where it serves as a parent mapping for the controller. All the request mappings on the controller methods are relative to the parent mapping. It is similar to the folder directory structure.

## @ResponseBody annotation

1. When a request comes to the dispatcher servlet, it maps the request to a controller. The controller executes and sends a response back to the dispatcher servlet. That response is usually the name of a view to be shown to the user along with the model.
2. 

Controller receives a request and responds with model and view

The welcome method created above does not return the name of a view. Rather, it returns a String message to be shown to the user. Since the dispatcher servlet is expecting a view name, we need to notify it that the return value is to be shown to the user as it is. Otherwise, the dispatcher servlet will search for a view with the String response returned by the welcome method.

The @ResponseBody annotation tells the dispatcher servlet that the response coming back is not the name of a view. Rather, it is the response that we want to send back to the user. Without using this annotation, if we run the application and type [**http://localhost:8080/spring/**](http://localhost:8080/spring/), we will get a 404 error which indicates that the page is not found because the dispatcher servlet by default searches for a URL mapping. With this annotation, the string **“Welcome to the Tennis Player database from Spring MVC!”** will be displayed as output.

The code in the widget below shows the response when a request to **“/”** is made.

In your IDE, run the application as Maven Build and specify **tomcat7:run** as goal. After Tomcat starts, you can see Running war on [*http://localhost8080/*](http://localhost8080/).

Type [**http://localhost8080/spring/**](http://localhost8080/spring/) in the browser. On this link, the contents returned by the welcome method of the TennisController are displayed.

Be sure to kill the server before attempting to re-run the application to avoid the Address already in use exception.

###### /

web.xml

player-servlet.xml

TennisController.java

**TennisController.java**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

package io.datajek.springmvc;

import org.springframework.stereotype.Controller;

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

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

@Controller

public class TennisController {

    @RequestMapping(value = "/")

    @ResponseBody

    public String welcome() {

        return "Welcome to the Tennis Player database from Spring MVC!";

    }

}





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/spring/>

In this lesson, we created a controller to map a URL to a method and return some response. The flow of a request in MVC is as follows:

1. **web.xml** receives all requests.
2. Anything starting with **/spring** will go to the front controller (dispatcher servlet) as specified in the **web.xml** file.
3. **player-servlet.xml** is the configuration file of the dispatcher servlet. It specifies where to perform a component scan to look for controllers.
4. The dispatcher servlet looks at the next part of the URL **“/”** and looks for a controller that is annotated with that particular URL.
5. When it finds a controller method (welcome) with that mapping, the method is executed.
6. The controller returns a response which can either be a model or a view to be rendered. In this lesson, it is a text message to be shown to the user.

# View

Learn how to create a view template and resolve a view name in this lesson.

**We'll cover the following**

* [Redirecting to a view](https://www.educative.io/courses/guide-spring-5-spring-boot-2/gkLM1Bv3pnr#Redirecting-to-a-view)
* [Resolving name of the view](https://www.educative.io/courses/guide-spring-5-spring-boot-2/gkLM1Bv3pnr#Resolving-name-of-the-view)
* [Developing the view page](https://www.educative.io/courses/guide-spring-5-spring-boot-2/gkLM1Bv3pnr#Developing-the-view-page)

When a controller or handler returns a response back to the dispatcher servlet, it is either shown as it is to the user (if the @ResponseBody annotation is present), or a view is called.

In real world applications, we do not return String responses back to the user. We redirect the user to a view or JSP.

## Redirecting to a view

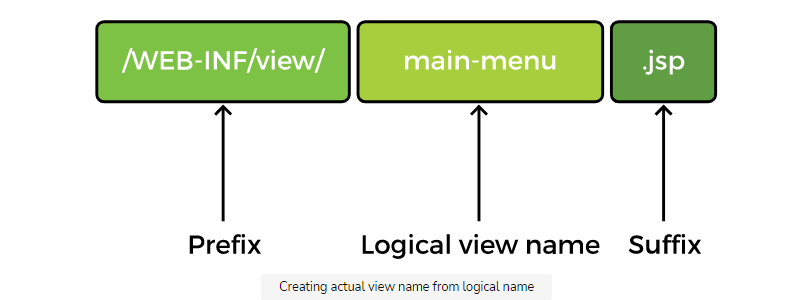
1. Say, we want a request to **“/”** to be redirected to **main-menu.jsp**. We can return the view name as follows:

@RequestMapping(value = "/")  
public String welcome() {  
    return "main-menu";  
}

The return value **main-menu** will be treated as the name of a view now, because we have removed the @ResponseBody annotation from the method.

## Resolving name of the view

1. In Spring, we only specify the name of the JSP and not the extension. Spring will use information from its config file and find the view page. Based on the information in the config file, it will look in the prefix directory, use the view name and then append the suffix to resolve the logical name returned by the method to an actual name.



Creating actual view name from logical name

Recall, that we created a bean of InternalResourceViewResolver class in the **player-servlet.xml** file, which had information to append a String as prefix and suffix to construct the complete location of the view as follows:

<bean  
    class = "org.springframework.web.servlet.view.InternalResourceViewResolver">  
        <property name="prefix">  
        <value>/WEB-INF/views/</value>  
        </property>  
  
        <property name="suffix">  
        <value>.jsp</value>  
        </property>  
</bean>

From the welcome method, we are returning a String **main-menu**. The view resolver will translate **main-menu** to **/WEB-INF/views/main-menu.jsp** by prepending **/WEB-INF/views/** and appending **.jsp** to the name of the view.

## Developing the view page

1. Now, we will create a JSP page **main-menu** in the **WEB-INF/views** directory. This page will display a text message:

<!DOCTYPE html>  
<html>  
<head>  
    <title>Main Menu</title>  
</head>  
<body>  
    <h2> Spring MVC - Tennis Player Database </h2>  
</body>  
</html>

###### /

main-menu.jsp

player-servlet.xml

TennisController.java

**player-servlet.xml**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

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

    xmlns:context="http://www.springframework.org/schema/context"

    xmlns:mvc="http://www.springframework.org/schema/mvc"

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

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

    http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc.xsd

    http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="io.datajek" />

    <mvc:annotation-driven />

    <bean

        class = "org.springframework.web.servlet.view.InternalResourceViewResolver">

        <property name="prefix">

        <value>/WEB-INF/views/</value>

        </property>

        <property name="suffix">

        <value>.jsp</value>

        </property>

    </bean>

</beans>





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/spring/>

This application is created with the spring-webmvc dependency. If Spring Boot is used to create the web application (with spring-boot-starter-web dependency), an addition dependency, tomcat-embed-jasper, is required to display the JSP pages.

1. Client sends a request to the application which is received by **web.xml**.
2. **web.xml** forwards all requests to the dispatcher servlet.
3. Dispatcher servlet checks its config file, **player-servlet.xml**, for controllers.
4. Based on the controller mappings, **“/”** is mapped to the welcome method in the **TennisController**.
5. The controller method returns a String response.
6. Since the welcome method of the controller does not have the @ResponseBody annotation, the dispatcher servlet treats the return value as a view name and forwards it to the view resolver configured in **player-servlet.xml** file.
7. The view resolver resolves the String **main-menu** to **/WEB-INF/views/main-menu.jsp** and returns the view/JSP to the dispatcher servlet.
8. Dispatcher servlet renders the view in the browser.

# HTML Forms

Learn how to read and process form data in Spring MVC application.

**We'll cover the following**

* [Controller class](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JQGrV5OlX4P#Controller-class)
* [showForm method](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JQGrV5OlX4P#showForm-method)
* [HTML form](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JQGrV5OlX4P#HTML-form)
* [processForm method](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JQGrV5OlX4P#processForm-method)
* [Player details page](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JQGrV5OlX4P#Player-details-page)

In this lesson, we will set up a form, which will prompt the user to enter a player’s name. When the user clicks the Submit button, our Spring MVC application will show a page containing the name entered by the user. Later, we will add functionality to show details of the player entered by the user.

A controller can have multiple request mappings. We will create a PlayerController to handle different request mappings. This controller will map the URL **showPlayerForm** to a method that displays the form and **processPlayerForm** to a method that directs the user to the player details page.

## Controller class

1. The first step is to create a class, PlayerController. The @Controller annotation tells Spring that this class will handle web requests.

@Controller  
public class PlayerController {  
    //method to handle /showPlayerForm  
  
    //method to handle /processPlayerForm  
}

## showForm method

1. First, we need a method that will show the form to the user. The **/showPlayerForm** URL maps to this method. We will call this method showForm.

@RequestMapping("/showPlayerForm")  
public String showForm () {  
    return "search-player-form";  
}

Spring will call the view resolver which will resolve **search-player-form** to **/WEB-INF/views/search-player-form.jsp**.

## HTML form

1. The showForm method created in the previous step, returns the **search-player-form** page. We will create this page in **WEB-INF/views** directory as follows:

<!DOCTYPE html>  
<html>  
<head>  
<title>Player Form</title>  
</head>  
<body>  
<h2>Player Form</h2>  
<br>  
    <form action = "processPlayerForm" method ="GET">  
 Enter player name: <input type = "text" name = "playerName" />  
                    <input type = "submit" value = "Submit"/>     
    </form>  
</body>  
</html>

The <form> element sends the data to the path **processPlayerForm**. It has a text box and a submit button.

If the application is run now and the URL **/showPlayerForm** is entered, the dispatcher servlet maps this request to **PlayerController** which returns **search-player-form**. After resolving the view name, the dispatcher servlet renders the view **search-player-form.jsp** in the browser. If we enter a name in the form and then press **Submit**, we encounter the 404 error because the dispatcher servlet is unbale to find a mapping for **/processPlayerForm**.

###### /

search-player-form.jsp

PlayerController.java

**PlayerController.java**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

package io.datajek.springmvc;

import org.springframework.stereotype.Controller;

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

@Controller

public class PlayerController {

    //method to handle /showPlayerForm

    @RequestMapping("/showPlayerForm")

    public String showForm () {

        return "search-player-form";

    }

    //method to handle /processPlayerForm

}





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/spring/showPlayerForm>

## processForm method

1. When the user clicks the submit button on the form, a GET request is generated for **/processPlayerForm**. We will now implement a method in the PlayerController to handle that request.

When the user enters a player name, we want him to be redirected to a page containing the name entered in the form. We will create a method processForm to return player-detail view as follows:

@RequestMapping("/processPlayerForm")  
public String processForm () {  
    return "player-detail";  
}

## Player details page

1. The last step is to create the **player-detail** page. Since we have configured the view resolver to search for pages in the **WEB-INF/views** folder, we will create a JSP file called **player-detail** in that directory. The details page will display the name entered in the **search-player-form**. To access the value entered in the form, we will use JSP expression language syntax, ${expression} as follows:

<!DOCTYPE html>  
<html>  
<head>  
<title>Player Details</title>  
</head>  
<body>  
 <h2>Player info</h2>  
 <hr>  
  
 <br><br>  
 Player name : ${param.playerName}  
  
</body>  
</html>

param is a predefined variable in EL which maps a request parameter to a value. It is similar to request.getParameter method. Here, we are accessing the player name using the text input element playerName from step 3 above.

The default value of isELIgnored attribute, which disables the evaluation of EL expressions, is true. It can be set to false by including the taglib reference <%@ page isELIgnored = "false" %>.

1. Instead of having to type the URL for the player form **(/spring/showPlayerForm)** in the address bar, we can add a link to the form in **main-menu.jsp** as follows:

<!DOCTYPE html>  
<html>  
<head>  
 <title>Main Menu</title>  
</head>  
<body>  
 <h2> Spring MVC - Tennis Player Database</h2>  
   
 <hr><br>  
 <a href="showPlayerForm"> Search Player</a>  
</body>  
</html>

Now the player form will be displayed on the home page of our application as a hyperlink.

The Spring MVC application can be accessed at [**http://localhost:8080/spring/**](http://localhost:8080/spring/)

###### /

player-detail.jsp

main-menu.jsp

search-player-form.jsp

PlayerController.java

**player-detail.jsp**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

<%@ page language="java" contentType="text/html; charset=ISO-8859-1"

    pageEncoding="ISO-8859-1"%>

<!DOCTYPE html>

<html>

<head>

<meta charset="ISO-8859-1">

<title>Player Details</title>

</head>

<body>

<h2>Player info</h2>

<hr>

<br><br>

Player name :  ${param.playerName}

</body>

</html>





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/spring/>

# Model

Learn how to pass data between controllers and views.

**We'll cover the following**

* [Passing Model object to controller](https://www.educative.io/courses/guide-spring-5-spring-boot-2/Y5j0DXw8z1W#Passing-Model-object-to-controller)
* [Read and process form data](https://www.educative.io/courses/guide-spring-5-spring-boot-2/Y5j0DXw8z1W#Read-and-process-form-data)
* [Add data to model with addAttribute](https://www.educative.io/courses/guide-spring-5-spring-boot-2/Y5j0DXw8z1W#Add-data-to-model-with-addAttribute)
* [Accessing model data in view](https://www.educative.io/courses/guide-spring-5-spring-boot-2/Y5j0DXw8z1W#Accessing-model-data-in-view)

Model is a container used for passing data between different components in our application. The controller class can add data to the model in the form of strings, numbers, or objects etc. The view page is able to access data placed in the model.

In this lesson, we will first pass the model to our controller class, then add some data to it and finally show how the data in the model is accessed in the view page.

Passing data from controller to view

We have created a form in **search-player-form.jsp** that takes a player name as input. The code for the form is reproduced below:

<form action = "processPlayerForm" method ="GET">  
Enter player name: <input type = "text" name = "playerName" />  
                    <input type = "submit" value = "Submit"/>      
</form>

When **Submit** is clicked, a GET request invokes the PlayerController because **/processPlayerForm** maps to it. The method created in the last lesson for processing form data is reproduced below:

@RequestMapping("/processPlayerForm")  
public String processForm () {  
    return "player-detail";  
}

## Passing Model object to controller

1. A controller method can have any number of parameters. Right now, we are not passing any parameters, but for reading form data, we need HttpServletRequest and to add data to the model we need the Model object in the method signature.

@RequestMapping("/processPlayerForm")  
public String processForm(HttpServletRequest request, Model model) {  
    return "player-detail";  
}

The model being passed as a parameter to the processForm method is empty. We will add data to it in the next step.

## Read and process form data

1. We can read the player name entered in the **search-player-form** and store it in a variable pName by using getParameter method of HttpServletRequest. The following code shows how to read form data:

@RequestMapping("/processPlayerForm")  
public String processForm (HttpServletRequest request, Model model) {  
    String pName = request.getParameter("playerName");  
    return "player-detail";  
}

## Add data to model with addAttribute

1. Now, we will add the player name that the user entered in the form to the model using the addAttribute method. This method takes a name-value pair of an attribute as follows:

@RequestMapping("/processPlayerForm")  
public String processForm (HttpServletRequest request, Model model) {  
    String pName = request.getParameter("playerName");  
    model.addAttribute("name", pName);  
    return "player-detail";  
}

We can use any name for the attribute. Here, we have used name. The value of the attribute is stored in the variable pName. Now the attribute name will be available to the view.

We can pass multiple values from the controller to the JSP. More attributes can be added to the model by calling the addAttribute method with name-value pairs.

## Accessing model data in view

1. Instead of accessing the form data in the **player-detail** page by using ${param.playerName}, we can now access the data placed in the model. In step 3, we set an attribute name in the model. This attribute can be accessed using expression language syntax ${attributeName} as follows:

<body>  
 <br><br>  
 Player name : ${name}  
</body>

The value of the attribute name will be plugged in. Notice that the attribute name used in the controller method and view page is the same.

###### /

search-player-form.jsp

player-detail.jsp

main-menu.jsp

PlayerController.java

**main-menu.jsp**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

<%@ page language="java" contentType="text/html; charset=ISO-8859-1"

    pageEncoding="ISO-8859-1"%>

<!DOCTYPE html>

<html>

<head>

<meta charset="ISO-8859-1">

<title>Main Menu</title>

</head>

<body>

    <h2> Spring MVC - Tennis Player Database</h2>

    <hr><br>

    <a href="showPlayerForm"> Search Player</a>

</body>

</html>





Run

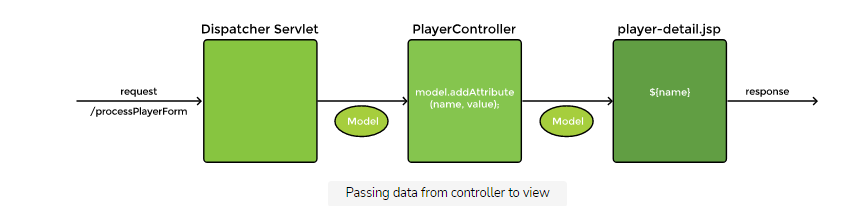
Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/spring/>

The Spring MVC application can be accessed at [**http://localhost:8080/spring/**](http://localhost:8080/spring/)

The code above shows how to pass data using model. The dispatcher servlet maps **/processPlayerForm** request to the processForm method of the PlayerController. It sends the Model object as a parameter. The controller adds data to the model using addAttribute method. The model is accessible to the view **player-detail.jsp** which uses expression laguage syntax ${attributeName} to display the data.



Passing data from controller to view

# Model - Part 2

cont..

**We'll cover the following**

* [Player class](https://www.educative.io/courses/guide-spring-5-spring-boot-2/393zow0EE6p#Player-class)
* [PlayerService class](https://www.educative.io/courses/guide-spring-5-spring-boot-2/393zow0EE6p#PlayerService-class)
* [Search player by name](https://www.educative.io/courses/guide-spring-5-spring-boot-2/393zow0EE6p#Search-player-by-name)
* [Defining beans](https://www.educative.io/courses/guide-spring-5-spring-boot-2/393zow0EE6p#Defining-beans)
* [Dependency injection](https://www.educative.io/courses/guide-spring-5-spring-boot-2/393zow0EE6p#Dependency-injection)
* [Adding search functionality](https://www.educative.io/courses/guide-spring-5-spring-boot-2/393zow0EE6p#Adding-search-functionality)
* [Adding data to model](https://www.educative.io/courses/guide-spring-5-spring-boot-2/393zow0EE6p#Adding-data-to-model)
* [Updating view page](https://www.educative.io/courses/guide-spring-5-spring-boot-2/393zow0EE6p#Updating-view-page)
* [GET vs POST request](https://www.educative.io/courses/guide-spring-5-spring-boot-2/393zow0EE6p#GET-vs-POST-request)

In this lesson, we will add search functionality to our Spring MVC application.

## Player class

1. The **Player** class is reproduced below:

public class Player {  
    private int id;  
    private String name;  
    private String nationality;  
    private Date birthDate;  
    private int titles;  
   
   //constructor  
   //getter and setter methods  
}

## PlayerService class

1. The **PlayerService** class contains a List of **Player** objects and has a method getPlayerByName that takes a String name as input and returns a **Player** object.

The **PlayerService** class is reproduced below:

public class PlayerService {  
    private List<Player> players = Arrays.asList(  
        new Player(1, "Djokovic", "Serbia", Date.valueOf("1987-05-22"), 81),  
        new Player(2, "Monfils", "France", Date.valueOf("1986-07-01"), 10),  
        new Player(3, "Isner", "USA", Date.valueOf("1985-04-26"), 15)  
        );   
   
    public Player getPlayerByName(String name){  
        return players.stream().filter(p -> p.getName().equals(name)).findFirst().get();  
    }  
}

## Search player by name

1. To add search functionality to the controller, we will invoke PlayerService and get the details of a player by passing the player name parameter entered by the user.

Previously, the way we created PlayerService was by creating an object like this:

PlayerService service = new PlayerService();

In the Spring MVC application, we will leverage Spring’s feature of dependency injection.

## Defining beans

1. Instead of us creating the object, we will let Spring create and inject dependencies. **PlayerService** is a dependency of the **PlayerController** class. We will let Spring manage the dependency and inject it in the **PlayerController** class.

For this, first we need to tell Spring that it has to manage the **PlayerService** object. To mark the **PlayerService** class as a service, we will use the @Service annotation. This tells Spring to create and manage **PlayerService** bean.

@Service  
public class PlayerService {  
   //...  
}

Spring can find beans by doing a component scan. For instance, providing a base package for component scan in player-servlet.xml enables Spring to find all beans with the @Component, @Controller, @Servive, and @Repository annotations and manage them.

## Dependency injection

1. To inject the **PlayerService** bean in the **PlayerController** bean, we will use the @Autowired annotation in the **PlayerController** class.

@Autowired  
PlayerService service;

Now, we don’t have to create an instance of the PlayerService class. Spring framework will do the work for us.

## Adding search functionality

1. Now that the service is available for use, all that remains to be done is to call the getPlayerByName method of the service class and get a Player object back.

@RequestMapping(value = "/processPlayerForm")  
public String processForm(HttpServletRequest request, Model model) {  
    String pName = request.getParameter("playerName");  
    Player player = service.getPlayerByName(pName);  
    model.addAttribute("name", pName);  
    return "player-detail";  
}

## Adding data to model

1. The Player object contains several fields. We can add those to the model using the addAttribute method.

@RequestMapping(value = "/processPlayerForm")  
public String processForm(HttpServletRequest request, Model model) {  
    String pName = request.getParameter("playerName");  
    Player player = service.getPlayerByName(pName);  
    model.addAttribute("name", pName);  
    model.addAttribute("country", player.getNationality());  
    model.addAttribute("dob", player.getBirthDate());  
    model.addAttribute("titles", player.getTitles());  
    return "player-detail";  
}

## Updating view page

1. The last task is to update **player-detail.jsp** to access the attributes added to the model in the previous step.

<body>  
<h2>Player info</h2>  
<hr>  
  
<br><br>  
Player name : ${name}  
<br><br>  
DoB: ${dob}  
<br><br>  
Country: ${country}  
<br><br>  
Titles: ${titles}  
</body>

Now, when we search for a player using the **search-player-form** we will be redirected to the **player-detail** page with information about the player. Note, that the application is configured to return results for only three players at the moment as we only entered data about Djokovic, Monfils and Isner in step 2.

###### /

search-player-form.jsp

player-detail.jsp

main-menu.jsp

Player.java

PlayerService.java

PlayerController.java

**PlayerController.java**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

package io.datajek.springmvc;

import javax.servlet.http.HttpServletRequest;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

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

@Controller

public class PlayerController {

    @Autowired

    PlayerService service;

    //method to handle /showPlayerForm

    @RequestMapping("/showPlayerForm")

    public String showForm () {

        return "search-player-form";

    }

    //method to handle /processPlayerForm

    @RequestMapping("/processPlayerForm")

    public String processForm(HttpServletRequest request, Model model) {

        String pName = request.getParameter("playerName");

        Player player = service.getPlayerByName(pName);

        model.addAttribute("name", pName);

        model.addAttribute("country", player.getNationality());

        model.addAttribute("dob", player.getBirthDate());

        model.addAttribute("titles", player.getTitles());

        return "player-detail";





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/spring/>

## GET vs POST request

In the **search-player-form** we have specified the type of request as **GET**. As a result, the information entered by the user is made part of the query string. If we change the type of request to **POST**, the URL string will no longer contain the player name.

<form action = "processPlayerForm" method ="POST">

# @RequestParam Annotation

This lesson shows how to use the @RequestParam annotation for binding form data.

The @RequestParam annotation is used to bind form data to a method parameter.

Passing HttpServletRequest object as a method parameter makes the getParameter method bind the playerName parameter to the variable pName.

@RequestMapping("/processPlayerForm")  
public String processForm (HttpServletRequest request, Model model) {  
    String pName = request.getParameter("playerName");  
    return "player-detail";  
}

Another way of binding form data is by using the request binding annotation @RequestParam.

1. The @RequestParam annotation is used as a method parameter:

public String   
processForm(@RequestParam("playerName") String pName, Model model) {  
  
}

Spring will read the parameter playerName from the request and bind it to the variable pName. Behind the scene, Spring executes:

String pName = request.getParameter("playerName");

The parameter name can be used as the variable name to make the code shorter:

public String processForm(@RequestParam String playerName, Model model) {  
  
}

1. Inside the method, we can use the variable pName. Suppose we want to convert the name to upper case and add it to the model.

public String processForm(@RequestParam("playerName") String pName, Model model) {  
    String theName = pName.toUpperCase();  
    model.addAttribute("name", theName);  
    return "player-detail";  
}

1. In the **player-detail** page, we can access the model attribute name using EL syntax as usual:

<body>  
 <br><br>  
 Player name : ${name}     
</body>

Now the uppercase name will be displayed on the page.

###### /

search-player-form.jsp

player-detail.jsp

main-menu.jsp

PlayerController.java

**PlayerController.java**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

package io.datajek.springmvc;

import javax.servlet.http.HttpServletRequest;

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

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

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

@Controller

public class PlayerController {

    //method to handle /showPlayerForm

    @RequestMapping("/showPlayerForm")

    public String showForm () {

        return "search-player-form";

    }

    //method to handle /processPlayerForm

    @RequestMapping("/processPlayerForm")

    public String processForm(@RequestParam("playerName") String pName, Model model) {

        String theName = pName.toUpperCase();

        model.addAttribute("name", theName);

        return "player-detail";

    }

}





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/spring/>

We can also provide a default value for the attribute incase the user hits **Submit** without entering a player name.

public String processForm(@RequestParam(value="playerName", defaultValue="xyz") String pName, Model model) {  
    String theName = pName.toUpperCase();  
    model.addAttribute("name", theName);  
    return "player-detail";  
}

With his change in the method signature, the default value xyz will be used, if no value is entered in the **search-player-form**.

# Spring MVC Form Tags

In this lesson, we will use Spring MVC form tags which provide automatic data binding.

**We'll cover the following**

* [@RequestMapping at class level](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMvK3AvZPgL#@RequestMapping-at-class-level)
* [Spring MVC form tags](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMvK3AvZPgL#Spring-MVC-form-tags)
* [Textbox form tag](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMvK3AvZPgL#Textbox-form-tag)
  + [Data binding](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMvK3AvZPgL#Data-binding)
* [@ModelAttribute annotation](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMvK3AvZPgL#@ModelAttribute-annotation)

HTML forms are used to get input from the user. Spring MVC provides form tags which support data binding. These tags automatically set and retrieve data of a bean.

Examples of form tags include <form:form>, <form:input>, <form:checkbox>,<form:radiobutton, and <form:select> etc. The taglib reference of the Spring Tag Library is needed on the JSP page to use the above mentioned form tags.

<%@ taglib prefix="form" uri= "http://www.springframework.org/tags/form" %>

To demonstrate the use of Spring MVC form tags, we will create a form **add-player-form**, where the user will enter information about a player. When the form is submitted, the user will be directed to a confirmation page, **player-confirmation**, which will echo the data entered in the \_\_add-player-form.

1. To keep the code for Spring MVC form tags separate, we will not use the already created **Player** class but create a new **Athlete** class in the **io.datajek.springmvc** package.

Initially this class will have just one field, lastName, along with getter and setter methods and a no-argument constructor.

public class Athlete {  
    private String lastName;  
  
    public Athlete() {  
    }     
  
    public String getLastName() {  
        return lastName;  
    }  
  
    public void setLastName(String lastName) {  
        this.lastName = lastName;  
    }  
}

1. Next, we will create a controller class **AthleteController** to handle requests for adding a tennis player.

@Controller  
public class AthleteController {  
   
    //method to handle /showPlayerForm  
   
    //method to handle /processPlayerForm  
}

## @RequestMapping at class level

Since we are using the same request mappings that were used in the **PlayerController**, Spring will throw an ambiguous mapping error. To avoid the error, we can provide a class level mapping as follows:

@Controller  
@RequestMapping("/player")  
public class AthleteController {  
   
 //method to handle /showPlayerForm  
   
 //method to handle /processPlayerForm  
}

Now, the **/showPlayerForm** mapping in **AthleteController** is a sub-mapping. It is relative to the parent mapping **/player**.

Hence, we will get rid of the ambiguous mapping error as requests to **/showPlayerForm** will be mapped to the **PlayerController** while requests to **/player/showPlayerForm** will be mapped to the **AthleteController**.

1. The showForm method will display the **add-player-form**. Since, we will be using Spring MVC Form Tags, we need to pass an empty **Athlete** object to the form so that data binding between the bean properties and form elements can take place.

To pass data between the controller and view, we use the Model object. We will create a new **Athlete** object and then add it to the model as follows:

@RequestMapping("/showPlayerForm")  
public String showForm(Model model) {  
 model.addAttribute("athlete", new Athlete());  
    return "add-player-form";  
}

The addAttribute method takes a name-value pair of an attribute. Here, the name of the attribute is athlete. We will use the same attribute name in the view page. The value of the attribute is an empty **Athlete** object.

## Spring MVC form tags

1. Now, we will create our view page, **add-player-form** and use Spring MVC form tags. Create a new JSP file in the **WEB-INF/views** directory. We will need the reference of the tag library to use the <form:form> tag on the page.

<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>  
  
<!DOCTYPE html>  
<html>  
<head>  
 <title>Add Player</title>  
</head>  
<body>  
 <h2>Player Form</h2>  
 <hr>  
  
 <form:form action = "processPlayerForm" modelAttribute="athlete">  
   
 </form:form>  
</body>  
</html>

Here, we have created a form using the Spring MVC <form:form> tag. The form will send a request to **/processPlayerForm**. We are using the same name for modelAttribute that was set in the previous step. The following image shows how the form is created using HTML form tags and Spring MVC form tags.

HTML vs Spring MVC form tags

## Textbox form tag

1. A text field is created using the <input type="text"> tag in HTML form tags. Spring MVC provides the <form:input> tag.

Text box tag in HTML vs Spring MVC

To create a field for entering the name, we will use the <form:input> tag. The path element in this tag binds the form field to a bean property as follows:

<form:form action = "processPlayerForm" modelAttribute="athlete">  
    <br><br>  
    Name: <form:input path="lastName"/>  
  
    <br><br>  
    <input type="submit" value="Add Player"/>  
  
</form:form>

### Data binding

Behind the scene, when Spring loads the **add-player-form**, it calls getter methods to populate the fields of the **Athlete** object. So, Spring will call athlete.getLastName(). Since we passed an empty **Athlete** object, the lastName field is empty. However, it is easy to pre-populate values in the form too.

Getter method called when loading form

When the **Add Player** button is clicked, Spring calls the setter methods to bind the data entered by the user to the bean properties. So, Spring will call athlete.setLastName() and with the value that the user entered in the text box.

Setter method called when submitting form

## @ModelAttribute annotation

1. We have used the modelAttribute called athlete in the form. This attribute passes data from the view page to the controller when the **Add Player** button is clicked. To read the data in the controller, we will use the @ModelAttribute annotation which binds an attribute to a method parameter.

public static String processForm(@ModelAttribute("athlete") Athlete myAthlete){  
    return "player-confirmation";  
}

When the user clicks the **Add Player** button on the form, behind the scene, the @ModelAttribute binds the data coming from the form to myAthlete variable (which is an object of **Athlete** type). Then, it adds the myAthlete object to the model as an attribute with the name, athlete. This model attribute is available to the view page, **player-confirmation**.

By using the @ModelAttribute annotation, we can simplify the code. This annotation removes the need to pass the Model and Athlete objects as method parameters. There is no need to explicitly add an attribute to the model using addAttribute method.

@ModelAttribute simplifies code

1. The last step is to show the form data on the confirmation page. For this, we will first create a page **player-confirmation** in **WEB-INF/views** folder. Since the form data has been added to the model, we can use expression language syntax **${attributeName}** to access the properties of the model attribute.

<!DOCTYPE html>  
<html>  
<head>  
 <title>Player Confirmation</title>  
</head>  
<body>  
 <h2>Player Confirmation</h2>  
 <hr>  
   
 <br><br>  
 The player has been added.   
   
 <br><br>  
 Name : ${athlete.lastName}  
</body>  
</html>

When Spring encounters ${athlete.lastName}, it will call athlete.getLastName() to display the value stored in the model attribute **athlete**.

###### /

Athlete.java

AthleteController.java

add-player-form.jsp

player-confirmation.jsp

main-menu.jsp

**Athlete.java**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

package io.datajek.springmvc;

public class Athlete {

    private String lastName;

    public Athlete() {

    }

    public String getLastName() {

        return lastName;

    }

    public void setLastName(String lastName) {

        this.lastName = lastName;

    }

}





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/spring/>

# Spring MVC Form Tags - Part 2

In this lesson, we will explore more Spring MVC form tags.

**We'll cover the following**

* [Drop-down list form tag](https://www.educative.io/courses/guide-spring-5-spring-boot-2/m2rDBm0lpjG#Drop-down-list-form-tag)
* [Radio button form tag](https://www.educative.io/courses/guide-spring-5-spring-boot-2/m2rDBm0lpjG#Radio-button-form-tag)
* [Checkbox form tag](https://www.educative.io/courses/guide-spring-5-spring-boot-2/m2rDBm0lpjG#Checkbox-form-tag)
  + [Configuring JSTL](https://www.educative.io/courses/guide-spring-5-spring-boot-2/m2rDBm0lpjG#Configuring-JSTL)
  + [c:forEach tag](https://www.educative.io/courses/guide-spring-5-spring-boot-2/m2rDBm0lpjG#c:forEach-tag)

Picking up from where we left, we will add more items to the **add-player-form**.

## Drop-down list form tag

1. We will add a drop-down list to our form, to let the user select the player’s country. For this, first we will update the **Athlete** class to add a country field along with getter and setter methods for the field.

public class Athlete {  
    private String lastName;  
    private String country;  
  
    //no-arg constructor  
  
    //getter and setter methods  
}

The complete code of the **Athlete** class is given at the end of the lesson.

1. In HTML, a drop-down list is created using a <select> with multiple <option> tags as follows:

<select name="playerCountry">  
    <option>Australia</option>  
    <option>France</option>  
    <option>Serbia</option>  
    <option>Switzerland</option>  
    <option>United States of America</option>  
</select>

Spring MVC’s equivalent tags are <form:select> and <form:option> with path specifying the bean property to which the drop-down list binds to.

We will use Spring MVC form tags to add a drop-down list to the **add-player-form.jsp** as follows:

<form:select path="country">  
    <form:option value="AUT" label="Austria"/>  
    <form:option value="FRA" label="France"/>  
    <form:option value="SRB" label="Serbia"/>  
    <form:option value="SUI" label="Switzerland"/>  
    <form:option value="USA" label="United States of America"/>  
</form:select>

path="country" maps the drop-down list to the country field in the **Athlete** class. The <form:option> tag contains value and label, the former being the actual value that gets passed when the form is submitted while the latter being the item displayed in the drop-down list. Here, we are displaying the name of the country but behind the scene, we are saving the country code.

After adding the drop-down list, the form looks like this:

<form:form action = "processPlayerForm" modelAttribute="athlete">  
    <br><br>  
    Name: <form:input path="lastName"/>  
  
    <br><br>  
    Country: <form:select path="country">  
        <form:option value="AUT" label="Austria"/>  
        <form:option value="FRA" label="France"/>  
       <form:option value="SRB" label="Serbia"/>  
       <form:option value="SUI" label="Switzerland"/>  
       <form:option value="USA" label="United States of America"/>  
    </form:select>  
  
    <br><br>  
    <input type="submit" value="Add Player"/>  
  
</form:form>

When the **Add Player** button is clicked, Spring will call the setter methods to bind the data entered by the user to the bean properties. So, Spring will call athlete.setLastName() and athlete.setCountry() with the value that the user entered in the text box and selected from the drop-down list.

1. In the last lesson, we explained in detail how the form data is passed to the **AthleteController** class and how it can be accessed on the **player-confirmation** page. Now, we will update the confirmation page to show the player’s country:

<body>  
    <h2>Player Confirmation</h2>  
    <hr>  
   
    <br><br>  
    The player has been added.   
   
    <br><br>  
     Name : ${athlete.lastName}  
  
    <br><br>  
    Country : ${athlete.country}  
</body>

When Spring encounters ${athlete.country}, it will call athlete.getCountry() to display the value stored in the model attribute athlete.

On the confirmation page, the ISO country code (being the value of the drop-down list) will be displayed.

## Radio button form tag

1. Radio buttons let the user choose a single option from the available options. We will use radio buttons in our form, to let the user select whether the player is left or right handed. We will also include an option for the rare case of a player being ambidextrous. For this, first we will update the **Athlete** class to add a field called handedness field along with getter and setter methods for the field.

public class Athlete {  
    private String lastName;  
    private String country;  
    private String handedness;  
  
    //no-arg constructor  
  
    //getter and setter methods  
}

1. Spring MVC provides the <form:radiobutton> tag with path that defines bean property mapping and value that contains the actual value that will be stored in the bean property when the form is submitted. A comparison of HTML and Spring MVC radio button tags is shown below:

We will add the radio button option in the form as follows:

Handedness:  
Left-Handed <form:radiobutton path="handedness" value="Left-Handed"/>  
Right-Handed <form:radiobutton path="handedness" value="Right-Handed"/>      
Ambidextrous <form:radiobutton path="handedness" value="Ambidextrous"/>

path maps the radio button to the handedness field of the **Athlete** class. Spring will call the athlete.setHandedness() with the value of the radio button chosen, when the form is submitted.

The complete code of the **add-player-form.jsp** is given at the end of the lesson.

1. Lastly, we will update the confirmation page and add the handedness of the player:

<body>  
    <h2>Player Confirmation</h2>  
    <hr>  
   
    <br><br>  
    The player has been added.   
   
    <br><br>  
    Name : ${athlete.lastName}  
  
    <br><br>  
    Country : ${athlete.country}  
  
    <br><br>  
    Handedness : ${athlete.handedness}  
</body>

${athlete.handedness} results in a call to athlete.getHandedness() to get the value stored in the athlete model attribute.

## Checkbox form tag

1. While radio buttons let the user choose only one option, check boxes allow the selection of multiple options. We will use check boxes to select the Grand Slam titles won by the player. Since, a player can have multiple titles to his name, we will add the grandSlams property as an array of Strings.

public class Athlete {  
    private String lastName;  
    private String country;  
    private String handedness;  
    private String[] grandSlams;  
  
    //no-arg constructor  
  
    //getter and setter methods  
}

1. Spring MVC offers the <form:checkbox> tag similar to the <form:radiobutton> tag used above. We will update the **add-player-form** to add the check box option, for Grand Slam titles won, as follows:

Grand Slam Titles Won:  
Australian Open <form:checkbox path="grandSlams" value="Australian Open"/>  
French Open <form:checkbox path="grandSlams" value="French Open"/>  
Wimbledon <form:checkbox path="grandSlams" value="Wimbledon"/>  
US Open <form:checkbox path="grandSlams" value="US Open"/>

When the form is submitted, Spring will call athlete.getGrandSlams().

A comparison of the HTML and Spring MVC checkbox tag is shown below:

1. On the **player-confirmation** page, we need a mechanism to access the values in the **grandSlams** array. One way is to use Java iterative statements like **for**, **while**, or **do-while**. These can be used in a JSP using scriptlets. An better alternate is to use the JSTL <c:forEach> tag to loop over the values and display the titles won by the player. The <c:forEach> tag is part of the JSTL core tags.

### Configuring JSTL

To be able to use the JSTL core <c:forEach> tag, we first need to include JSTL dependency to our project. This can be done by adding the following dependencies to the **pom.xml** file:

<dependency>  
    <groupId>javax.servlet</groupId>  
    <artifactId>jstl</artifactId>  
    <version>1.2</version>  
    <scope>runtime</scope>  
</dependency>  
   
<dependency>  
    <groupId>taglibs</groupId>  
    <artifactId>standard</artifactId>  
    <version>1.1.2</version>  
    <scope>runtime</scope>  
</dependency>

To be able to use JSTL tags on the JSP page, we need its taglib reference. We will use the following taglib on **player-confirmation.jsp**:

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

### **c:forEach** tag

Now we are all set to use the <c:forEach> tag to loop over the grandSlams array and display it as a list.

The following code first creates an unordered list though the <ul> tag. Then, using the <c:forEach> tag, it loops over athlete.grandSlams array and displays the current item in the variable title as a bulleted list using the <li> tag.

Grand Slam Titles :  
<ul>  
    <c:forEach items="${athlete.grandSlams}" var="title">  
        <li>${title}</li>  
    </c:forEach>  
</ul>

###### /

Athlete.java

AthleteController.java

add-player-form.jsp

player-confirmation.jsp

main-menu.jsp

**add-player-form.jsp**

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

        <br><br>

        Country: <form:select path="country">

            <form:option value="AUT" label="Austria"/>

            <form:option value="FRA" label="France"/>

            <form:option value="SRB" label="Serbia"/>

            <form:option value="SUI" label="Switzerland"/>

            <form:option value="USA" label="United States of America"/>

        </form:select>

        <br><br>

        Handedness: &emsp;

            Left-Handed <form:radiobutton path="handedness" value="Left-Handed"/> &emsp;

            Right-Handed <form:radiobutton path="handedness" value="Right-Handed"/> &emsp;

            Ambidextrous <form:radiobutton path="handedness" value="Ambidextrous"/>

        <br><br>

        Grand Slam Titles Won: &emsp;

            Australian Open <form:checkbox path="grandSlams" value="Australian Open"/> &emsp;

            French Open <form:checkbox path="grandSlams" value="French Open"/> &emsp;

            Wimbledon <form:checkbox path="grandSlams" value="Wimbledon"/> &emsp;

            US Open <form:checkbox path="grandSlams" value="US Open"/>

        <br><br>

        <input type ="submit" value = "Add Player"/>

    </form:form>

</body>

</html>





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/spring/>

**Form Data Validation**

This lesson shows how to use the Hibernate Validator with Spring MVC application.

**We'll cover the following**

* [Hibernate Validator](https://www.educative.io/courses/guide-spring-5-spring-boot-2/39lWwB2OoEM#Hibernate-Validator)
* [Validation annotations](https://www.educative.io/courses/guide-spring-5-spring-boot-2/39lWwB2OoEM#Validation-annotations)
* [Spring validation support](https://www.educative.io/courses/guide-spring-5-spring-boot-2/39lWwB2OoEM#Spring-validation-support)
* [@Vaild annotation](https://www.educative.io/courses/guide-spring-5-spring-boot-2/39lWwB2OoEM#@Vaild-annotation)
* [form:error tag](https://www.educative.io/courses/guide-spring-5-spring-boot-2/39lWwB2OoEM#form:error-tag)
* [CSS error class](https://www.educative.io/courses/guide-spring-5-spring-boot-2/39lWwB2OoEM#CSS-error-class)

Data validation includes checking if the required fields are not left blank, numbers are within a given range, and data is entered in the correct format. The **Standard Bean Validation API** is the preferred approach for validation in Spring applications. We will use the Hibernate implementation of the API known as Hibernate Validator.

**Hibernate Validator**

The current version of Hibernate validator is Hibernate Validator 7.x. Spring 5 is not compatible with this release because of differences in package naming convention. Spring 5 uses Java EE packages starting with **javax.\*** while Hibernate Vaildator 7 has changed package names to **jakarta.\***.

For this reason, we will use an older version of Hibernate Validator that use the **javax.\*** package naming convention. The validator version that is compatible with Tomcat 7 is version 5.1.3. Newer versions of the validator, 6.2.x, can be used with Tomcat 8 or higher.

<dependency>  
    <groupId>org.hibernate</groupId>  
    <artifactId>hibernate-validator</artifactId>  
    <version>5.1.3.Final</version>  
</dependency>

After adding this dependency, we can use the JSR 349 validation annotations.

**Validation annotations**

Different validation annotations can be used on the fields to validate the data. Hibernate implements the following JSR 349 bean validation annotations:

* @NotNull checks if a required field is left blank.
* @Size checks that the input matches a given number of characters or digits.
* @Min and @Max validate that a number is within a given range.
* @Past and @Future apply to date fields to check if the date entered is before or after the current date.
* Regular expressions can be validated using the @Pattern annotation.

Custom validation annotations for business rules of the application can also be created.

In addition to the above mentioned annotations, Hibernate validator also provides additional annotations like @NotEmpty, @NotBlank and @Email etc.

**Spring validation support**

Spring MVC provides support for different bean validation APIs; JSR 303 (EE6), JSR 349 (EE7), and JSR 380 (EE8). The specifications are backward compatible. We have configured our application to automatically detect and enable validation support.

We included the following tag in **player-servlet.xml** while configuring our Spring MVC Application:

<mvc:annotation-driven />

This tag lets Spring MVC detect and enable validation support provided by the bean validation API.

**@Vaild annotation**

We can validate a model object by using the @Valid annotation in the controller class. When the user clicks the **Submit** button, the processForm method is executed. The @Valid annotation makes sure that the data entered by the user passes the validation rules.

Data validation takes place before business logic is executed

Spring will first bind the input from the form with the attributes of the model object, then it will validate the object and check for any constraint violations. The BindingResult object contains the results of the validation check.

@RequestMapping("/processPlayerForm")  
public static String processForm(@Valid @ModelAttribute("athlete") Athlete myAthlete,   
                                 BindingResult result){  
  
}

An important point to note, is the order of parameters. Spring MVC validation will only work as desired if the BindingResult parameter appears immediately after the @ModelAttribute, else, the validation rules will be ignored.

Inside the method, we will check the validation result. If the result object has any errors, it means that one or more constraints have been violated. In that scenario, we want the user to be sent back to the **add-player-form**, to fix the errors.

If the result object does not have any errors, then the user can proceed to the **player-confirmation** page. The hasErrors() method, used in the code below, returns a Boolean value.

public static String processForm(@Valid @ModelAttribute("athlete") Athlete myAthlete,   
                                 BindingResult result) {  
    if (result.hasErrors())   
        return "add-player-form";  
    else  
        return "player-confirmation";  
}

**form:error tag**

In the event of a constraint violation, when the user is redirected to the input form, we need a way to inform him about the error. The <form:error> tag displays the error. The path attribute describes the name of the field for which the error message is displayed. For example, to display errors for the lastName field, we can write:

Name: <form:input path = "lastName"/>  
<form:errors path="lastName" />

Here, we are telling Spring to display an error message, if it is set. When the form is displayed for the first time, we will not see any error message. The error message will only be displayed when the form is shown after validation constraint violation.

**CSS error class**

We can define the CSS style for error messages inline in the **add-player-form** as follows:

<head>  
<title>Add Player</title>  
  
<style>  
.error {  
    color:red;  
    font-style: italic;  
}  
</style>  
  
</head>

The error class defined above, can be referenced in the <form:error> tag:

Name: <form:input path = "lastName"/>  
<form:errors path="lastName" cssClass="error"/>

Now, the error messages will be displayed in red color and in italics.

AthleteController.java

add-player-form.jsp

player-confirmation.jsp

main-menu.jsp

Athlete.java

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

import·javax.validation.Valid;

import·org.springframework.stereotype.Controller;

import·org.springframework.ui.Model;

import·org.springframework.validation.BindingResult;

import·org.springframework.web.bind.annotation.ModelAttribute;

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

@Controller

@RequestMapping("/player")

public·class·AthleteController·{

→

→   //method·to·handle·/showPlayerForm

→   @RequestMapping("/showPlayerForm")

→   public·String·showForm(Model·model)·{

→   →   model.addAttribute("athlete",·new·Athlete());

→   →   return·"add-player-form";

→   }

→   //method·to·handle·/processPlayerForm

→   @RequestMapping("/processPlayerForm")

→   public·static·String·processForm(@Valid·@ModelAttribute("athlete")·Athlete·myAthlete,

→   →   →   →   →   BindingResult·result)·{

→   →   if·(result.hasErrors())·

→   →   →   return·"add-player-form";

→   →   else

→   ····→   return·"player-confirmation";

→   }

→

}





Back

Spring MVC Form Tags - P

# Validating Required Fields

In this lesson, we will learn how to validate required fields.

**We'll cover the following**

* [@NotNull and @Size](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qA0DJ12MNgk#@NotNull-and-@Size)
* [Handling whitespaces in input](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qA0DJ12MNgk#Handling-whitespaces-in-input)
* [@InitBinder annotation](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qA0DJ12MNgk#@InitBinder-annotation)

A required filed is a mandatory field that must be filled in before the form is submitted. In this lesson, we will add a required field to our form. The **add-player-form** has a textbox for name that stores data in the lastName field of the **Athlete** class. We will change this field and split it into first name and last name, with the latter being a required field. The first name can be left blank.

When the user clicks on the **Add player** button, the request goes to the **AthleteController** class which performs validation check and based on the results, either sends the user to the **player-confirmation** page or back to the **add-player-form** for resubmission.

1. First, we will add the field firstName to the **Athlete** class along with its getter and setter methods as follows:

public class Athlete {  
    private String firstName;  
    private String lastName;  
    //...  
}

## @NotNull and @Size

1. To ensure that the lastName field is not left blank, we will add the @NotNull annotation to it as follows:

import javax.validation.constraints.NotNull;  
  
public class Athlete {  
    private String firstName;  
  
    @NotNull(message = "This is a required field.")  
    private String lastName;  
    //...  
}

@NotNull checks for null values but when the field is left blank in the form, an empty string “” binds to the field and it qualifies the validation test and the message is not displayed. To solve this problem, @NotNull annotation is paired with @Size as follows:

import javax.validation.constraints.NotNull;  
import javax.validation.constraints.Size;  
  
public class Athlete {  
    private String firstName;  
  
    @NotNull(message = "This is a required field.")  
    @Size(min=1, message = "This is a required field.")  
    private String lastName;  
    //...  
}

@Size with min=1 ensures that there is at least one character in the field to pass the validation test, thus an empty String “” will fail the validation. Another approach is to use the @NotEmpty annotation from the **org.hibernate.validator** package.

import org.hibernate.validator.constraints.NotEmpty;  
  
public class Athlete {  
    private String firstName;  
  
    @NotEmpty(message = "This is a required field.")  
    private String lastName;  
    //...  
}

1. We will update the **add-player-form** and add the first name field and also put an asterisk (\*) next to the last name to indicate that it is a required field.

<form:form action = "processPlayerForm" modelAttribute="athlete">  
 <br><br>  
 First Name: <form:input path = "firstName"/>  
 <br><br>  
 Last Name \*: <form:input path = "lastName"/>  
 <form:errors path= "lastName" cssClass="error"/>  
 <!-- rest of the code -->  
</form:form>

On the **player-confirmation** page, we will display the first name and last name of the player.

<b>Name</b> : ${athlete.firstName}   
${athlete.lastName}

Now, if the last name field on the form is left blank, the user will be sent back to the form with an appropriate message that indicates why the form submission failed.

## Handling whitespaces in input

1. Right now, if we enter whitespaces in the last name textbox, the validation test will still pass because @NotNull checks if the entered value is not null and @Size checks if the size of the input is greater than zero. We need a way to trim the user input and remove any whitespaces before the validation test is performed.

## @InitBinder annotation

We need to pre-process the form data and trim whitespaces before validation is performed in the controller. We can place the pre-processing code in a method and annotate it with the @InitBinder annotation. The code in the initBinder method will execute first. This method will trim the String values to remove any leading and trailing whitespaces from the input. If the String only has whitespaces, it will be trimmed to null.

The initBinder method takes a parameter WebDataBinder. The return type of the method is void.

@InitBinder  
public void initBinder(WebDataBinder binder) {  
}

We will use the StringTrimmerEditor from the String API which removes whitespaces from String values. We create an object of the StringTrimmerEditor class with true as constructor argument to specify that the String will be trimmed to null if it is composed entirely of whitespaces.

StringTrimmerEditor editor = new StringTrimmerEditor(true);

Next, we will register the custom editor created above with the binder.

binder.registerCustomEditor(String.class, editor);

The complete code of the initBinder method is shown below. We will add it to the controller class to handle whitespaces in input.

@InitBinder  
public void initBinder(WebDataBinder binder) {  
    StringTrimmerEditor editor = new StringTrimmerEditor(true);  
    binder.registerCustomEditor(String.class, editor);  
}

This code will trim all String values coming as part of the request before validation is performed.

Other operations that can be performed in the initBinder method are converting Strings to Date values or customizing the input. In addition to registerCustomEditor method, the WebDataBinder also supports addValidators and addCustomFormatter methods.

###### /

Athlete.java

AthleteController.java

add-player-form.jsp

player-confirmation.jsp

main-menu.jsp

**player-confirmation.jsp**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<%@ page language="java" contentType="text/html; charset=ISO-8859-1"

    pageEncoding="ISO-8859-1"%>

<!DOCTYPE html>

<html>

<head>

<meta charset="ISO-8859-1">

<title>Player Confirmation</title>

</head>

<body>

    <h2>Player Confirmation</h2>

    <hr>

    <br><br>

    The player has been added.

    <br><br>

    <b>Name</b> : ${athlete.firstName} ${athlete.lastName}

    <br><br>

    <b>Country</b> : ${athlete.country}

    <br><br>

    <b>Handedness</b> : ${athlete.handedness}

    <br><br>

    <b>Grand Slam Titles</b> :

    <ul>

        <c:forEach items="${athlete.grandSlams}" var="title">

            <li>${title}</li>

        </c:forEach>





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/spring/>

# Validating Numbers

Learn how to validate number fields in this lesson.

**We'll cover the following**

* [@Min and @Max](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B1MoB9PRgVY#@Min-and-@Max)
* [@NotNull](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B1MoB9PRgVY#@NotNull)

In this lesson, we will show how to validate a number field. Currently, we don’t have a number field in the **Athlete** class. We will add a field rank to store the current ranking the player. The value of rank can be between 1 and 100. Any value outside this range should result in a validation error. We will use @Min and @Max to validate the number range.

## @Min and @Max

1. We will create an Integer field rank and annotate it with @Min and @Max as follows:

import javax.validation.constraints.Max;  
import javax.validation.constraints.Min;  
  
public class Athlete {  
   
    @Min(value=1, message="Value must be greater than or equal to 1.")  
    @Max(value=100, message="Value must be less than or equal to 100.")  
    private Integer rank;  
  
    //constructor  
    // getter and setter methods

We have used the Integer wrapper class instead of the primitive type int as it will help in making the field required and handling type conversion errors.

## @NotNull

1. We can use the @NotNull annotation on the rank field to make it a required field. int is a primitive type and cannot be null. That is why we used the wrapper class Integer, so it can have null values.

@NotNull(message="This is a required field.")  
@Min(value=1, message="Value must be greater than or equal to 1.")  
@Max(value=100, message="Value must be less than or equal to 100.")  
private Integer rank;

If a required field is left blank, an empty String “” binds to it. If we had used int, a conversion from String to int would result in an error.

1. We will update the **add-player-form** to add the new number field (using &ltform:input>) and display errors (using &ltform:errors>) as follows:

Current Rank \*: <form:input path = "rank"/>  
<form:errors path= "rank" cssClass="error"/>

1. The **AthleteController** class contains the validation logic where a validation error takes the user back to the **add-player-form**, while a correct input shows the **player-confirmation** page.

The validation logic in the **AthleteController** works for all fields of the **Athlete** object.

1. Lastly, we will update the **player-confirmation** page to display the rank value.

<b>Current Rank</b> : ${athlete.rank}

The code given below can be executed to verify that an input outside the specified range results in an error message and the confirmation page is displayed only when the input is correct.

###### /

Athlete.java

AthleteController.java

add-player-form.jsp

player-confirmation.jsp

main-menu.jsp

**Athlete.java**

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

    public Athlete() {

    }

    public String getFirstName() {

        return firstName;

    }

    public void setFirstName(String firstName) {

        this.firstName = firstName;

    }

    public String getLastName() {

        return lastName;

    }

    public void setLastName(String lastName) {

        this.lastName = lastName;

    }

    public String getCountry() {

        return country;

    }

    public void setCountry(String country) {

        this.country = country;

    }

    public String getHandedness() {

        return handedness;

    }





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/spring/>

# Validating Dates

In this lesson we will validate dates values in the past and in a specific format.

**We'll cover the following**

* [@Past](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B6k0rORkBw2#@Past)
* [@DateTimeFormat](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B6k0rORkBw2#@DateTimeFormat)

In this lesson, we will show how to validate a date field. Say, we want to store the date when the player won the last title. We will add a field lastWon to the **Athlete** class. This value must be in the past and must follow a specific pattern (dd-mm-yyyy). Any date value violating these constraints should result in a validation error. We will use @Past and @DateTimeFormat to validate the date field.

## @Past

1. We will create a Date field lastWon and generate getter and setter methods. Then, we will use the @Past annotation with an appropriate error message as follows:

import javax.validation.constraints.Past;  
  
public class Athlete {  
   
    @Past(message="Date must be in the past.")  
    private Date lastWon;  
  
    //constructor  
    // getter and setter methods

## @DateTimeFormat

1. Date values can be entered in multiple formats. To ensure that the value is in a specific format dd-MM-yyyy, we will use the @DateTimeFormat annotation. This is not part of the JSR 303 annotations as can be seen from the import:

 import org.springframework.format.annotation.DateTimeFormat;  
  
@DateTimeFormat(pattern = "dd-MM-yyyy")  
@Past(message="Date must be in the past.")  
private Date lastWon;

The pattern attribute specifies the date pattern against which the values will be validated. In case values are entered in a wrong format, a conversion error will occur and Spring shows a default message which is verbose and not user friendly.

The validation logic in the **AthleteController** class works for all fields of the **Athlete** object, hence we do not need to handle validation of the lastWon field separately.

1. Now, We will update the input form to add the date field along with the error message to be displayed if constaints are violated as follows:

Last Won (dd-mm-yyyy):<form:input path="lastWon"/>  
<form:errors path="lastWon" cssClass="error" />

1. In the same manner, we will update the confirmation page to display the date value. We will use the JSTL formatting tag <fmt:formatDate> to display the date. To use this tag, we will include a taglib reference as follows:

<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>

The <fmt:formatDate> tag has type and pattern attributes to format the date using our custom pattern dd-MM-yyyy.

<b>Last Won</b> : <fmt:formatDate value="${athlete.lastWon}" type="date" pattern="dd-MM-YYYY"/>

The code given below can be executed to verify that any value in the future or in an incorrect format results in an error message and the confirmation page is displayed only when the input is correct.

###### /

Athlete.java

AthleteController.java

add-player-form.jsp

player-confirmation.jsp

main-menu.jsp

**Athlete.java**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

package io.datajek.springmvc;

import java.util.Date;

import javax.validation.constraints.Max;

import javax.validation.constraints.Min;

import javax.validation.constraints.NotNull;

import javax.validation.constraints.Past;

import javax.validation.constraints.Size;

import org.hibernate.validator.constraints.NotEmpty;

import org.springframework.format.annotation.DateTimeFormat;

public class Athlete {

    private String firstName;

//  @NotEmpty(message = "This is a required field.")

    @NotNull(message = "This is a required field.")

    @Size(min=1, message = "This is a required field.")

    private String lastName;

    private String country;

    private String handedness;

    private String[] grandSlams;

    @NotNull(message = "This is a required field.")

    @Min(value=1, message="Value must be greater than or equal to 1.")

    @Max(value=100, message="Value must be less than or equal to 100.")

    private Integer rank;

    @DateTimeFormat(pattern = "dd-MM-yyyy")





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/spring/>

# Validating Patterns with Regular Expressions

Learn how to validate fields against a custom format using regular expressions.

**We'll cover the following**

* [@Pattern annotation](https://www.educative.io/courses/guide-spring-5-spring-boot-2/gkoK8DBqGKr#@Pattern-annotation)

Regular expressions define search patterns which can be used for validation purpose. In this lesson, we will show how to validate a currency input to match our custom format.

We will create a field prizeMoney to store the total earnings of the player. The entered value has to follow the $#,###,### format. Values that do not match this format will result in a validation error.

1. First step is to create the field for prize money along with the getter and setter methods.

public class Athlete {  
    //...  
    private String prizeMoney;  
  
    //getter and setter methods  
}

## @Pattern annotation

1. We will add validation rule to the prizeMoney field using a regular expression supplied to the @Pattern annotation along with an appropriate error message.

@Pattern(regexp="^\\$([0-9]{1,3},([0-9]{3},)\*[0-9]{3})$", message="Incorrect format")  
private String prizeMoney;

Here, we are applying a regular expression to validate prizeMoney and display an error message if pattern does not match. The regular expression is broken down in parts and explained below:

* ^ matches the starting position within the string
* \$ is for the $ sign
* [0-9]{1,3} means a digit with variable length from 1 to 3
* [0-9]{3}, group of 3 digits followed by a comma
* \* indicates that the preceding element can occur multiple times
* [0-9]{3} means a group of 3 digits (at the end, hence, without comma)
* $ matches the ending position within the string

1. Next, we will update the input form, **add-player-form**. The prize money field is displayed as a text box with an error tag as follows:

Prize Money (USD):<form:input path="prizeMoney" placeholder="$#,###,###"/>  
<form:errors path="prizeMoney" cssClass="error" />

We have used the placeholder attribute to show the required input format.

Similarly, we will update the confirmation page, **player-confirmation** as follows:

<b>Prize Money</b> : ${athlete.prizeMoney}

The validation logic is valid for all fields of the **Athlete** class.

The code below can be executed to verify that the regular expression works as desired.

###### /

Athlete.java

AthleteController.java

add-player-form.jsp

player-confirmation.jsp

main-menu.jsp

**Athlete.java**

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

import javax.validation.constraints.Past;

import javax.validation.constraints.Pattern;

import javax.validation.constraints.Size;

import org.hibernate.validator.constraints.NotEmpty;

import org.springframework.format.annotation.DateTimeFormat;

public class Athlete {

    private String firstName;

//  @NotEmpty(message = "This is a required field.")

    @NotNull(message = "This is a required field.")

    @Size(min=1, message = "This is a required field.")

    private String lastName;

    private String country;

    private String handedness;

    private String[] grandSlams;

    @NotNull(message = "This is a required field.")

    @Min(value=1, message="Value must be greater than or equal to 1.")

    @Max(value=100, message="Value must be less than or equal to 100.")

    private Integer rank;

    @DateTimeFormat(pattern = "dd-MM-yyyy")

    @Past(message = "Date must be in the past.")

    private Date lastWon;

    @Pattern(regexp="^\\$([0-9]{1,3},([0-9]{3},)\*[0-9]{3})$", message="Incorrect format")

    private String prizeMoney;

    public Athlete() {





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/spring/>

# Custom Error Messages

This lesson shows how to override Spring MVC's default validation messages with our custom error messages.

**We'll cover the following**

* [Type mismatch errors](https://www.educative.io/courses/guide-spring-5-spring-boot-2/q2AmZK2z1XD#Type-mismatch-errors)
* [messages.properties file](https://www.educative.io/courses/guide-spring-5-spring-boot-2/q2AmZK2z1XD#messages.properties-file)
* [Finding the error code](https://www.educative.io/courses/guide-spring-5-spring-boot-2/q2AmZK2z1XD#Finding-the-error-code)
* [Defining custom error message](https://www.educative.io/courses/guide-spring-5-spring-boot-2/q2AmZK2z1XD#Defining-custom-error-message)

We can specify error messages inside the annotation like this:

@NotNull(message="This is a required field.")  
@Past(message="Date must be in the past.")  
@Min(value=1, message="Value must be greater than or equal to 1.")

These error messages override the default error messages for the constraints provided by the Hibernate validator. The default messages are located in the **ValidationMessages.properties** file inside the hibernate-validator jar file.

## Type mismatch errors

This type of error occurs in non String fields when data cannot be converted into the desired datatype. For example if a String in entered in an int field.

In such a case, Spring shows a default message which shows why the exception was thrown. Custom messages can be created in **messages.properties** file.

Another approach is to externalize the error messages by placing them in a properties file. This file is called **messages.properties** and must be placed in the **resources** folder. When there is a constraint violation, Spring will check the **messages.properties** file and look for a key matching the error code. If the key is found it will display the custom error message, else use the default error message for the constraint violation.

## messages.properties file

1. We will first create a new folder called **resources** under the **WEB-INF** directory. The location of the properties file is important.
2. Next, we will create the properties file in the **resources** folder. The name of the file is equally important. It should be called **messages.properties**. This file contains a key value pair of the error type and the corresponding error message.

## Finding the error code

1. To find the error code, we can use the BindingResult object, which contains the result of validation.

public static String processForm(@Valid @ModelAttribute("athlete") Athlete myAthlete,BindingResult result){  
  
    if (result.hasErrors())   
    {  
        System.out.println("Binding result: " + result);  
        return "add-player-form";  
    }  
    else  
       return "player-confirmation";  
}

In case of an error, the output shows the object and field which failed validation. It also shows the error codes from most specific to most generic. We will pick the first (most specific) error code and use it in the **message.properties** file to override the default error message.

## Defining custom error message

1. We will add our custom messages in the properties file by listing the error type, model attribute name, field name, and finally the custom error message as follows:

typeMismatch.athlete.rank=Enter a valid number.

A pictorial explanation of this line of code is shown below:

Error messages can be specific to a field as shown above, or generic. For example:

typeMismatch=Invalid value.  
typeMismatch.java.lang.Integer=Invalid number.  
#typeMismatch.java.util.Date=Enter a valid date.

Generic messages are useful when a custom message for a specific error cannot be found. The type mismatch error for Date field is commented out, so that the message matching the most generic error code in line 1 is displayed when a string input is entered in for lastWon Date field.

1. Now, we need to inform Spring to use the file we just created. We will do this by adding an entry in the config file, **player-servlet.xml**. We will add a bean called messageSource and specify the location of the properties file as follows:

 <bean id= "messageSource"  
    class= "org.springframework.context.support.ReloadableResourceBundleMessageSource" >  
  
 <property name= "basenames">  
     <value>/WEB-INF/resources/messages</value>  
 </property>  
</bean>

Here, we are defining a bean named messageSource. The application context will use this bean to resolve messages based on the properties file created in the previous step. The ReloadableResourceBundleMessageSource in an implementation of the MessageSource interface. The basenames property tells the ReloadableResourceBundleMessageSource class about the location of the properties file. We only specify the name of the file. Spring attaches the **.properties** extension automatically.

Now, if we run the application and enter a text input in the number field, **Current Rank**, or in the Date field **Last Won**, we will get the custom error messages defined in the **messages.properties** file.

###### /

messages.properties

player-servlet.xml

Athlete.java

AthleteController.java

add-player-form.jsp

player-confirmation.jsp

**messages.properties**

1

2

3

4

typeMismatch                   = Invalid value.

typeMismatch.java.lang.Integer = Invalid number.

#typeMismatch.java.util.Date   = Enter a valid date.

typeMismatch.athlete.rank      = Enter a valid number.





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/spring/>

# Custom Validation Annotation

Learn how to create an annotation to validate custom business rules.

**We'll cover the following**

* [Creating custom annotation](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qZWL6KKBpw0#Creating-custom-annotation)
  + [@Retention annotation](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qZWL6KKBpw0#@Retention-annotation)
  + [@Target annotation](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qZWL6KKBpw0#@Target-annotation)
* [Creating a constraint validator](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qZWL6KKBpw0#Creating-a-constraint-validator)
  + [initialize() method](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qZWL6KKBpw0#initialize()-method)
  + [isValid() method](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qZWL6KKBpw0#isValid()-method)
* [@Constraint annotation](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qZWL6KKBpw0#@Constraint-annotation)

In this lesson, we will create an annotation that implements some custom business rules. Spring will validate the user input against our custom validation rules and display our custom error message if the input is not in the desired format.

As an example, we will create a form to enter the record of matches between two players.

The form will prompt the user to enter two players, along with the head-to-head match score. The head-to-head score must follow a specific format. Two numbers must be separated by a hyphen. For example, 28-10 is a valid input. Characters are not allowed for this field. We will attach our custom annotation to validate the head-to-head score against the above-mentioned requirements.

1. First, we will create a new class called **PlayerStats** having three fields as follows:

public class PlayerStats {  
    private String player1;  
    private String player2;  
    private String head2head;  
   
    //no arg constructor   
    //getter and setter methods  
}

1. Next, we will create a controller class **StatsController** to handle requests to show and process the form.

@Controller  
public class StatsController {  
  
    //method to handle /showStatsForm  
    @RequestMapping("/showStatsForm")  
    public String showForm(Model model) {  
        model.addAttribute("playerStats", new PlayerStats());  
        return "head-to-head";  
    }  
  
    //method to handle /processStatsForm      
}

When a request is sent to [**http://localhost:8080/spring/showStatsForm**](http://localhost:8080/spring/showStatsForm), it is mapped to the showForm method of the **StatsController**. The client is directed to the **head-to-head** form. An empty **PlayerStats** object called playerStats is added as a model attribute.

1. Now, we will create the JSP page, **head-to-head**, in the **WEB-INF/views** folder. The layout of the form is shown below:

<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>  
  
<!DOCTYPE html>  
<html>  
<head>  
    <title>Head-to-Head</title>  
</head>  
<body>  
<h2>Head-to-Head Statistics</h2>  
 <hr>  
 <form:form action = "processStatsForm" modelAttribute="playerStats">  
      
     <br><br>  
     Player 1: <form:input path = "player1"/> &emsp; vs. &emsp;  
           
     Player 2: <form:input path = "player2"/>  
       
     <br><br>  
     Head-to-Head:  <form:input path = "head2head" placeholder="##-##"/> &emsp;  
       
     <br><br>  
    <input type ="submit" value = "Submit"/>  
  
 </form:form>  
       
</body>  
</html>

We have a taglib reference at the top which is needed to use Spring MVC form tags. The form has three text fields for taking the player names and head-to-head score as input. The modelAttribute playerStats binds the **PlayerStats** bean to the form. When **Submit** is clicked, a request is sent to processStatsForm.

1. The processForm method has a request mapping of **/processStatsForm** and is executed when the **Submit** button on the form shown above is clicked.

@RequestMapping("/processStatsForm")  
public String processForm(@Valid @ModelAttribute("playerStats") PlayerStats playerStats,  
                          BindingResult result) {  
    if (result.hasErrors())   
       return "head-to-head";  
    else  
       return "stats-confirmation";  
}

The @Valid annotation is used to validate the model object and store the results in the BindingResult object. In case of errors in form, the user is sent back to the **head-to-head** form, otherwise the **stats-confirmation** page is shown which echoes the information entered by the user.

The success page, **stats-confirmation**, is shown below:

<html>  
<head>  
<title>Head To Head Stats</title>  
</head>  
<body>  
<h2>Head-to-Head Statistics</h2>  
<hr>  
  
<br><br>  
<b>${playerStats.player1}</b> &emsp; vs. &emsp; <b>${playerStats.player2}</b>   
  
<br><br>  
Head-to-Head Score : <b>${playerStats.head2head}</b>  
</body>  
</html>

## Creating custom annotation

1. After the above prep work, we will create an annotation to validate the head-to-head score. We will create a new package inside **io.datajek.springmvc**, called **validation**, to keep all validation related files separate. The annotation called @HeadToHead is created as follows:

package io.datajek.springmvc.validation;  
  
public @Interface HeadToHead {  
  
}

We will use additional annotations, @Retention and @Target to further describe the HeadToHead annotation.

import java.lang.annotation.ElementType;  
import java.lang.annotation.Retention;  
import java.lang.annotation.RetentionPolicy;  
import java.lang.annotation.Target;  
  
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.FIELD)  
public @interface HeadToHead {  
   
}

### @Retention annotation

@Retention describes how long an annotation should be retained in the byte code with possible options being SOURCE, RUNTIME, or CLASS. We want our annotation to be retained till runtime, because that is when our annotation will be used.

### @Target annotation

The ElementType property of @Target annotation specifies where the annotation can be applied. We want it to be applied to a field. Other options include METHOD, MODULE, PACKAGE, or CONSTRUCTOR. Multiple options can also be provided as a comma separated list.

Inside the **HeadToHead** class, we need to include three mandatory methods:

* + message() for the default error message
  + groups() to assign the annotation to a group. Related constraints can be grouped together. If the Class<?>[] is left empty, it declares Default group.
  + payload() method associates the constraint with a payload. Payloads specify the metadata information with a constraint e.g., the severity of constraint violation, or error code etc.

In addition to the above methods, we need to implement the value() method, if our annotation has a value parameter (like the @Min(value=1) annotation). This method tells Spring that the annotation has a parameter called **value** and specifies a default value, incase it is not provided with the annotation.

Default values ensure that the annotation is customizable and can be used without any parameters. We can also use it with different parameter values.

public @interface HeadToHead {  
   
    public String message () default "Value must have digits in ##-## format.";  
  
    public Class<?>[] groups() default {};  
  
    public Class<? extends Payload>[] payload() default {};  
  
}

We can apply the annotation to the head2head field of **PlayerStats** class as follows:

import io.datajek.springmvc.validation.HeadToHead;  
  
//...  
@HeadToHead(message= "Invalid format")  
private String head2head;

Here, we are providing an error message with the annotation. If the annotation is used without a message, then the default message specified in the annotation class will be used.

## Creating a constraint validator

1. Now, we will create a class that contains the actual business rules for the annotation. This class is called **Head2HeadConstraintValidator** and it implements the **ConstraintValidator** interface. It takes two parameters, the annotation and the data type that it validates.

public class Head2HeadConstraintValidator implements ConstraintValidator <HeadToHead, String> {  
  
    //custom business logic to validate the annotation  
  
}

The **ConstraintValidator** interface has two methods, initialize() and isValid(), which we will implement.

### initialize() method

The initialize() method is used to get the parameters from the annotation and initialize them. It is used as a post-construct method. If the annotation contains the value() method, it can be used to get the parameters specified in the annotation.

This method is a defined as a default method in the **ConstraintValidator** interface and it is not necessary to provide an implementation. Since, our annotation does not take any value parameter that can be initialized, so there is no need to use this method. The method definition is shown for the sake of completeness:

@Override  
public void initialize(HeadToHead headToHeadScore) {  
      
}

### isValid() method

The isValid() method contains the validation rules. The validator class must provide an implementation of this method. When the form data is submitted, this method gets executed to validate the input. It has two parameters, the data entered in the form and a helper class which can be used to set error messages. The method tests the user input against the validation rules and returns a boolean value.

public boolean isValid(String headToHeadScore, ConstraintValidatorContext context) {  
      
    boolean isValid = headToHeadScore.matches("[0-9]{1,2}-[0-9]{1,2}");  
    return isValid;  
}

According to our validation rules, the entered data must have two numbers separated by a hyphen. Additionally we have imposed a restriction that the numbers may only be 2 digits long.

## @Constraint annotation

1. To connect the annotation class **HeadToHead** with the constraint validator class **Head2HeadConstraintValidator**, we will use the @Constraint annotation. The validatedBy element specifies the validator used to validate fields marked with @HeadToHead annotation.

@Constraint(validatedBy=Head2HeadConstraintValidator.class)  
    public @interface HeadToHead {  
    //...  
}

1. After applying the annotation @HeadToHead, we will add the &ltform:error> tag for the head2head field in **head-to-head.jsp** to display validation errors.

In the code shown below, we have made the head2head field mandatory by using the @NotEmpty annotation. You can add validation rules for the player name fields as well.

###### /

stats-confirmation.jsp

head-to-head.jsp

Head2HeadConstraintValidator.java

HeadToHead.java

StatsController.java

PlayerStats.java

**stats-confirmation.jsp**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

<%@ page language="java" contentType="text/html; charset=ISO-8859-1"

    pageEncoding="ISO-8859-1"%>

<!DOCTYPE html>

<html>

<head>

<meta charset="ISO-8859-1">

<title>Head To Head Stats</title>

</head>

<body>

    <h2>Head-to-Head Statistics</h2>

    <hr>

    <br>

    <br>

    <b>${playerStats.player1}</b> &emsp; vs. &emsp;

    <b>${playerStats.player2}</b>

    <br>

    <br> Head-to-Head Score :

    <b>${playerStats.head2head}</b>

</body>

</html>





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/spring/showStatsForm>

# Dispatcher Servlet Java Configuration

This lesson shows an alternate way of configuring the dispatcher servlet using Java configuration.

**We'll cover the following**

* [Defining the dispatcher servlet](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVl6zK4Yox7#Defining-the-dispatcher-servlet)
* [Dispatcher servlet configuration](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVl6zK4Yox7#Dispatcher-servlet-configuration)

The XML files **web.xml** and **player-servlet.xml** define the Spring MVC application configuration. It is possible to configure our application without using any XML. We will replace the files mentioned above with Java classes to initialize the dispatcher servlet.

1. First, we will create a package for the configuration files called **config** in **io.datajek.springmvc**. This package will contain two files that replace the **web.xml** and **player-servlet.xml**.

## Defining the dispatcher servlet

1. The **web.xml** file (created while configuring Spring MVC Application - Part 1) defined the dispatcher servlet, provided the location of its config file, and set up the URL mapping. To perform the same functionality in Java, Spring provides an abstract class called AbstractAnnotationConfigDispatcherServletInitializer. This class has methods for specifying the root configuration class, dispatcher servlet configuration class and URL mapping.

We will provide an implementation of this class to define the dispatcher servlet and set up the URL mapping by creating a class **MyDispatcherServletInitializer** that extends **AbstractAnnotationConfigDispatcherServletInitializer**. It comes with the following unimplemented methods:

public class MyDispatcherServletInitializer extends   
       AbstractAnnotationConfigDispatcherServletInitializer {  
  
    @Override  
    protected Class<?>[] getRootConfigClasses() {  
        // TODO Auto-generated method stub  
       return null;  
    }  
  
    @Override  
    protected Class<?>[] getServletConfigClasses() {  
       // TODO Auto-generated method stub  
       return null;  
    }  
  
    @Override  
    protected String[] getServletMappings() {  
       // TODO Auto-generated method stub  
       return null;  
    }  
}

1. The getServletConfigClasses method provides the name of the dispatcher servlet config class. In XML configuration, **player-servlet.xml** file had the dispatcher servlet configuration.

We will override this method to point to the config class **PlayerConfig** (which we will create in step 5).

@Override  
protected Class<?>[] getServletConfigClasses() {  
    return new Class[] { PlayerConfig.class };  
}

The above method provides the same functionality as the <servlet> and <init-param> tags in the **web.xml** file.

1. To setup the URL mapping for the dispatcher servlet, we will override the getServletMappings method and provide our custom mapping of “/spingmvc/\*”.

@Override  
protected String[] getServletMappings() {  
    return new String[] { "/springmvc/\*" };  
}

The above method provides the same functionality as the <servlet-mapping> tag in the **web.xml** file.

We have intentionally changed the mapping from “/spring/" to "/springmvc/” so that it can be verified that the application is indeed reading the Java configuration.

## Dispatcher servlet configuration

1. The **player-servlet.xml** file (created while configuring Spring MVC Application - Part 2) added support for component scanning, data validation and formatting, and defined the view resolver.

We will create a class **PlayerConfig** and use annotations to provide the same functionality as the **player-servlet.xml** file as follows:

@Configuration  
@EnableWebMvc  
@ComponentScan(basePackages="io.datajek")  
public class PlayerConfig() {  
  
}

The @Configuration annotation tells Spring that this class contains configuration information. @ComponentScan annotation specifies the base package where Spring will look for components. @EnableWebMvc provides the same functionality as the <mvc: annotation-driven/> tag in XML. It adds support for conversion, formatting and validation. It can also handle processing of @Controller classes and @RequestMapping methods etc.

1. Now we will define a bean for the view resolver. We will use the @Bean annotation and create a method called viewResolver. In the method, we will create an object of InternalResourceViewResolver and set the prefix and suffix as follows:

@Bean  
public ViewResolver viewResolver(){  
    InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();  
    viewResolver.setPrefix("/WEB-INF/views/");  
    viewResolver.setSuffix(".jsp");  
    return viewResolver;  
}

A bean with the name ViewResolver will be created which will map the logical view name to the actual view name.

1. While creating custom error messages, we created the **messages.properties** file and created a bean in **player-servlet.xml** to load that file.

The same can be achieved in java configuration as follows:

@Bean  
public MessageSource messageSource() {  
    ReloadableResourceBundleMessageSource messageSource =   
                                        new ReloadableResourceBundleMessageSource();  
    messageSource.setBasename("/WEB-INF/resources/messages");  
    return messageSource;  
}

Here, we have created a bean of ReloadableResourceBundleMessageSource class and provided the path of the **messages.properties** file using the setBasename method.

1. If there is an error in the **pom.xml** file, stating that the failOnMissingWebXml property is set to true, we can include the maven-war-plugin to get rid of the error.

<plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-war-plugin</artifactId>  
 <version>3.1.0</version>  
</plugin>

We have replaced the **web.xml** and **player-servlet.xml** files with MyDispatcherServletInitializer and PlayerConfig classes. Our application is now configured using pure Java.

The code widget below shows the Java configuration of dispatcher servlet. Note that we have to delete the XML configuration files from the project to run the Java configuration.

###### /

MyDispatcherServletInitializer.java

PlayerConfig.java

main-menu.jsp

**MyDispatcherServletInitializer.java**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

package io.datajek.springmvc.config;

import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

public class MyDispatcherServletInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

    @Override

    protected Class<?>[] getRootConfigClasses() {

        // TODO Auto-generated method stub

        return null;

    }

    @Override

    protected Class<?>[] getServletConfigClasses() {

        return new Class[] { PlayerConfig.class };

    }

    @Override

    protected String[] getServletMappings() {

        return new String[] { "/springmvc/\*" };

    }

}





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/springmvc/>

# Unit Testing in Spring

**JUnit**

In this lesson, we discuss the definition of unit testing and talk about the JUnit5 framework.

**We'll cover the following**

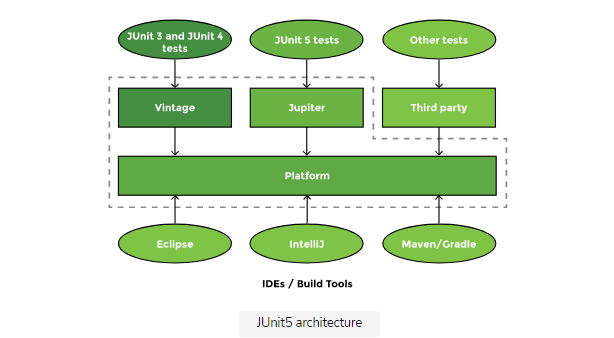
* [JUnit Architecture](https://www.educative.io/courses/guide-spring-5-spring-boot-2/m2j9DkY32MG#JUnit-Architecture)
* [JUnit dependency](https://www.educative.io/courses/guide-spring-5-spring-boot-2/m2j9DkY32MG#JUnit-dependency)

Testing ensures that the application is working as expected. As applications increase in size, it can become difficult to test big chunks of code. Testing is done at different levels and in different ways. Testing an application after deployment is called **screen testing** or **system testing**. Testing a specific method or a class is called **unit testing**. It usually involves writing tests for each method of a class. Unit tests can also be written for a group of methods or to test the whole class.

**JUnit Architecture**

JUnit is a popular unit testing framework. It automates the testing process so that tests can run continually. If there is a change in the code such that the functionality changes, automated testing will catch it. The tests compare the actual output to the expected output to check if the code is working correctly and notify the developer if it is not.

JUnit 5 is the newest release of the framework at the time of writing of this course. It changed the monolithic architecture of JUnit 4 to have separate jar files for the JUnit platform (test engine), Jupiter (JUnit 5 tests), Vintage (JUnit4 tests), and 3rd party extensions while also adding support for new testing patterns. Most IDE’s have built-in support for JUnit 5.



JUnit5 architecture

JUnit tests are automated, which means that the test can be kept running any number of times. The tests can be run every hour or every time there is a change in the code. The developer is notified whenever a test fails. These tests can be run under continuous integration.

**JUnit dependency**

To enable JUnit, we need the JUnit engine and the Jupiter API to create tests. When creating a Spring Boot project, JUnit is already included as a dependency. For other cases, the JUnit dependency can be added as follows:

<dependency>  
    <groupId>org.junit.jupiter</groupId>  
    <artifactId>junit-jupiter-engine</artifactId>  
    <scope>test</scope>  
</dependency>

<scope> specifies where we want the jars to be active. test scope means that this dependency will not be included in the final build.

**Writing a Unit Test**

Learn how to write a unit test with JUnit 5.

**We'll cover the following**

* [Creating a test class](https://www.educative.io/courses/guide-spring-5-spring-boot-2/m2Rm6GJm3Pr#Creating-a-test-class)
* [@Test annotation](https://www.educative.io/courses/guide-spring-5-spring-boot-2/m2Rm6GJm3Pr#@Test-annotation)
* [fail method](https://www.educative.io/courses/guide-spring-5-spring-boot-2/m2Rm6GJm3Pr#fail-method)
* [Steps in writing a unit test](https://www.educative.io/courses/guide-spring-5-spring-boot-2/m2Rm6GJm3Pr#Steps-in-writing-a-unit-test)
* [Assertions](https://www.educative.io/courses/guide-spring-5-spring-boot-2/m2Rm6GJm3Pr#Assertions)
* [Creating multiple test cases](https://www.educative.io/courses/guide-spring-5-spring-boot-2/m2Rm6GJm3Pr#Creating-multiple-test-cases)

To understand how unit testing is done with JUnit, we will create a simple class ArrayMethods. We will write some methods in this class and then test those using JUnit.

1. Our class has methods to find the index of a given number in the array and a method to search if a number is present in the array.

public class ArrayMethods {  
    int findIndex(int[] array, int number) {  
        //...  
        return index;  
    }  
   
    boolean searchForNumber(int[] array, int numberToSearchFor) {    
        boolean found = false;  
        //...  
        return found;  
    }     
}

The findIndex method takes an integer array and a number and returns the index at which the number is found in the array.

The searchArray method accepts an array of integers and a number to search for and returns true if the number is found.

1. JUnit framework allows us to call the findIndex method with an array containing three integers, say 8, 4, and 5, and look for 4 in the array. The index where 4 is found, (1) is returned.

To test the findIndex method, we will create a unit test. The tests should always be separate from the source code. They should be in a different folder so that they are not deployed to production along with the source code.

There are two ways of creating a class for testing purposes. One is to manually create a class in the src/test/java folder and write tests. The other is to use the IDE support. Almost all IDEs have support for JUnit 5. So instead of us creating the test class, the IDE will do that for us and place it in **src/test/java**.

**Creating a test class**

1. For the class that we want to test, a JUnit test can be created in the same way that a class or interface is created. We will call the test **ArrayMethodsTest**. Conventionally, the name of the test is the name of the class being tested with the word *Test* appended to it. This is for convenience, so that we know which class we are testing. It also lets the developer use the package visibility features of Java, where protected methods that are not visible to classes outside the package become visible to the test class.

**@Test annotation**

The @Test annotation is present before the test method. It signifies that the method is a unit test. When JUnit encounters this annotation, it knows it has to run the test. So if we click on the project and choose **Run as Junit test**, JUnit will know which methods it has to run.

**fail method**

By default, the test method contains the fail method. If the test is run, it will fail because of the fail method and the text message “**Not implemented yet**” will be displayed.

@Test  
public void test() {  
    fail("Not implemented yet");  
}

If the test is run after removing the fail method, it will succeed. JUnit works by checking if any of the tests have failed. The test will be a success unless any checks fail. In the JUnit world, absence of failure is success. In IDEs, a colored bar is the visual representation of the status of the execution. A green bar shows up in case a test succeeds, and a red one indicates a failing test.

**Steps in writing a unit test**

1. Every unit test involves a few simple steps. First and foremost, we need to create an object of the class under test and use the object to call the method that we want to test. Then, we set up the inputs which constitute a test scenario, execute the method, and get results. Finally, we compare the results to the expected value.

Let’s say we want to test the findIndex method of the ArrayMethods class. The steps of the test are shown below:

@Test  
public void testfindIndex\_numberInArray() {  
    //1. create object of the class  
    ArrayMethods arrayMethods = new ArrayMethods();  
   
    //2. call method  
    int result = arrayMethods.findIndex(new int[] {8,4,5}, 4);  
  
    //3. compare the actual results to the expected  
}

**Assertions**

1. JUnit provides us with a number of assertions, which are methods that test whether the expectation and reality are the same. One way to confirm if the actual results are the same as expected is to print a message on the console. For huge applications, scanning through logs is a cumbersome process. JUnit’s way of reporting the status of the test is by using one of the assert methods.

To verify if the result of the above test is 1. The JUnit assert method assertEquals can be used. It compares expected and actual values and generates a report in case the two values do not match. The two input arguments to the assertEquals method are the expected result and the actual value returned by the method.

assertEquals(expected, actual);

In our case, the expected value from our input array is 1 and the value actually returned by the search method is stored in the variable result.

assertEquals(1, result);

1. Now let’s assume there is a change in the source code and a different answer is returned (change return index to return ++index). This test will start failing if the actual value does not match the expected value and we will get an assertion error: “Expected <1> but was <2>”. An advantage of continuously running the tests is that we can get notified of any bugs that have been introduced in our code and fix them right away.

**Creating multiple test cases**

1. Following the same simple steps, we can test other test cases to check when a value is not in the array and when an empty input array is sent to the method.

The tests can be improved by making the result variable inline, i.e., by calling the search method inside the assertEquals method as follows:

@Test  
public void testfindIndex\_numberNotInArray() {  
    ArrayMethods arrayMethods = new ArrayMethods();  
    assertEquals(-1, arrayMethods.findIndex(new int[]{8,4,5}, 1));  
}

@Test  
public void testfindIndex\_emptyArray() {  
    ArrayMethods arrayMethods = new ArrayMethods();  
    assertEquals(-1, arrayMethods.findIndex(new int[]{}, 1));  
}

JUnit will run all methods marked with the @Test annotation. There is no particular order in which the tests run. JUnit creates a new test instance every time a test runs. So if we have three tests, there will be three instances of the ArrayMethods class.

UnitTestingApplication.java

ArrayMethods.java

ArrayMethodsTest.java

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

package io.datajek.unittesting;

import static org.junit.jupiter.api.Assertions.\*;

import org.junit.jupiter.api.Test;

class ArrayMethodsTest {

    @Test

    public void testfindIndex\_numberInArray() {

        //1. create object of the class

        ArrayMethods arrayMethods = new ArrayMethods();

        //2. call method

        int result = arrayMethods.findIndex(new int[]{8,4,5}, 4);

        //3. compare the actual results to the expected

        assertEquals(1, result);

    }

    @Test

    public void testfindIndex\_numberNotInArray() {

        ArrayMethods arrayMethods = new ArrayMethods();

        assertEquals(-1, arrayMethods.findIndex(new int[]{8,4,5}, 1));

    }

    @Test

    public void testfindIndex\_emptyArray() {

        ArrayMethods arrayMethods = new ArrayMethods();

        assertEquals(-1, arrayMethods.findIndex(new int[]{}, 1));

    }

}





Run

Save

Reset

Back

# Assert Methods

Learn about a variety of assert methods used to test if the actual output matches the expected output.

**We'll cover the following**

* [assertEquals](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JEpWoBmg9ZJ#assertEquals)
  + [assertArrayEquals](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JEpWoBmg9ZJ#assertArrayEquals)
* [assertTrue](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JEpWoBmg9ZJ#assertTrue)
* [assertNull](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JEpWoBmg9ZJ#assertNull)
* [fail](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JEpWoBmg9ZJ#fail)
* [assertThrows](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JEpWoBmg9ZJ#assertThrows)
* [assertAll](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JEpWoBmg9ZJ#assertAll)

There are a number of assert methods that are available with the import of the Assertions.\* package.

## assertEquals

1. The assertEquals method compares two values — the expected value and the actual value — to determine whether or not both are the same. A test method from the last lesson using assertEquals is reproduced here:

@Test  
public void testfindIndex\_numberNotInArray() {  
    ArrayMethods arrayMethods = new ArrayMethods();  
    assertEquals(-1, arrayMethods.findIndex(new int[]{8,4,5}, 1));  
}

There are many variations of assertEquals with different data types. An optional third String argument can be added that contains a message about what the test is for. This feature is handy for a code base spanning thousands of lines, where the developer might not know what a failing test intended to do. Suppose the following test fails because we want the expected output to be 1, whereas the actual value that was returned is -1. In this case, our message will explain what the method under test is supposed to do.

assertEquals(1, arrayMethods.findIndex(new int[]{8,4,5}, 1),  
                                 "The findIndex method finds the index of a given number");

### assertArrayEquals

1. This method takes two arrays and compares them element by element and finds out if both are the same or not. For example, if there is a sortArray method to sort an array of numbers, the assertArrayEquals method can be used as follows:

assertArrayEquals(new int[] {0,1,3,7}, arrayMethods.sortArray(new int[] {3,1,7,0});

## assertTrue

1. assertTrue takes a condition as argument and checks if it is true. In the same manner, assertFalse checks for a condition to be false. This can easily be achieved through the assertEquals method as well. These are just convenience methods to use when testing for a Boolean condition to be either true or false. The same result can be achieved through the assertEquals method. Both approaches are shown below:

@Test  
void testAssert() {  
    Boolean condition = true;  
    assertEquals(true,true);  
    assertTrue(condition);  
}

## assertNull

1. assertNull and assertNotNull are used to check for null values. The same can be achieved using assertEquals and passing null as the expected value, but as mentioned above, these methods are just for convenience.

@Test  
void testAssert() {  
    String str = null;  
    assertEquals(null, str);  
    assertNull(str);  
}

## fail

1. The fail method fails the test, no matter what. It can be used to mark an unfinished test, so that the developer is notified that it needs to be completed.

@Test  
public void testSortArray() {  
    fail("unimplemented method");  
}

## assertThrows

1. Sometimes we may want to assert exceptions. The assertThrows method is used to check if an exception is thrown. It takes the executable, which causes the exception along with the exception that we are expecting.

For example, suppose we add a method printArray to our class as follows:

void printArray(int[] array, int index) {         
    System.out.println(array[index]);  
}

To test if the ArrayIndexOutOfBound exception is thrown, we can write the following test:

@Test  
public void testfindIndex\_indexOutOfBound() {  
    ArrayMethods arrayMethods = new ArrayMethods();  
    assertThrows(ArrayIndexOutOfBoundsException.class, ()->arrayMethods.printArray(  
                                                               new int[] {1,8,5}, 3));  
}

The test contains the type of the exception that we are expecting (ArrayIndexOutOfBoundsException.class) and a lambda expression that will evaluate to the exception.

## assertAll

1. Notice that we have very simple tests that just assert some basic stuff. We can combine all of them in one method rather than having a separate method for each test case. Consider the following test written for the findIndex method. In the test, we are combining scenarios like finding the index when the number is in the array, when it is not in the array, and when the array is empty. Instead of writing three tests, each testing one scenario, we can use an assertAll to combine all the assertions using lambda expressions as follows:

@Test  
public void testfindIndex() {  
    ArrayMethods arrayMethods = new ArrayMethods();  
    assertAll(  
    () -> assertEquals(1, arrayMethods.findIndex(new int[]{8,4,5}, 4)),  
    () -> assertEquals(-1, arrayMethods.findIndex(new int[]{8,4,5}, 1)),  
    () -> assertEquals(-1, arrayMethods.findIndex(new int[]{}, 1))  
    );  
}

The code for all these tests is shown below. Out of 7, 2 tests will fail, which will cause a build failure. Details of the failing tests can be seen in the output.

UnitTestingApplication.java

ArrayMethods.java

ArrayMethodsTest.java

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

        String str = null;

        assertEquals(null, str);

        assertNull(str);

    }

    //this test will fail because of fail method

    @Test

    public void testSortArray() {

        fail("unimplemented method");

    }

    @Test

    public void testfindIndex\_indexOutOfBound() {

        ArrayMethods arrayMethods = new ArrayMethods();

        assertThrows(ArrayIndexOutOfBoundsException.class, ()->arrayMethods.printArray(new int[] {1,8,5}, 3));

    }

    @Test

    public void testfindIndex() {

        ArrayMethods arrayMethods = new ArrayMethods();

        assertAll(

                () -> assertEquals(1, arrayMethods.findIndex(new int[]{8,4,5}, 4)),

                () -> assertEquals(-1, arrayMethods.findIndex(new int[]{8,4,5}, 1)),

                () -> assertEquals(-1, arrayMethods.findIndex(new int[]{}, 1))

                );

    }

}





Run

Save

Reset

Back

Writing a Unit Test

**JUnit Test Lifecycle Annotations**

In this lesson, we will learn about the annotations used at different stages of the test lifecycle.

**We'll cover the following**

* [@BeforeEach](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7X7YnYk2QQy#@BeforeEach)
* [@AfterEach](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7X7YnYk2QQy#@AfterEach)
* [@BeforeAll](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7X7YnYk2QQy#@BeforeAll)
* [@AfterAll](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7X7YnYk2QQy#@AfterAll)
* [Annotations to scale tests](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7X7YnYk2QQy#Annotations-to-scale-tests)

JUnit provides us with some lifecycle hooks that can be plugged in at various places in the test cycle. For example, some initialization code needs to be run before each test, and some cleanup code needs to be run after each test. If we have multiple tests, there might be some code that we want to initialize before any of the tests run. Similarly, there might be some teardown code that we want to run after each test has run.

JUnit provides annotations that can be used on each of these lifecycle hooks. These annotations are @BeforeAll, @AfterAll, @BeforeEach, and @AfterEach. The following image shows how these annotations are used at different stages in the lifecycle of a test.

Test lifecycle

**@BeforeEach**

Before a test is run, some initialization is needed. In the test cases that we wrote, we created a new instance of the class in every test. A method with the @BeforeEach annotation is executed once before every test method executes. We can initialize the objects or establish a database connection before each test in a method annotated with @BeforeEach. With JUnit, there is no guarantee about the order in which the tests are run. But, it is guaranteed that no matter the order of tests, @BeforeEach is run before the test is run.

@BeforeEach  
void init(){  
    //Initialize the object here  
    System.out.println("Initializing before test");   
    arrayMethods = new ArrayMethods();  
}

We need to have the ArrayMethods object as a member variable to be able to access its methods in various tests. Even if the object instance is shared by all the test cases, its state is being initialized in the init method.

**@AfterEach**

Similar to @BeforeEach, the @AfterEach annotation tells JUnit to run the method after a test has been executed.

@AfterEach  
void afterEachTest(){  
    System.out.println("Clean up after test");  
}

@AfterEach on a method ensures that it is run after the test to perform any clean up stuff.

**@BeforeAll**

A method annotated with @BeforeAll is run once before all test methods of the class are executed. It is a setup method and has to be static. This method is executed even before the instance of the class to be tested is created.

@BeforeAll  
static void beforeAllTests() {  
    System.out.println("Run this code before all tests");  
}

Only a static method can run without being dependent on the class in which it is written.

**@AfterAll**

Just like the @BeforeAll annotation, the @AfterAll annotation on a method ensures that it is run after all the tests have been run. It executes after all test methods of the class are executed. This method is a class level method and is, therefore, static.

@AfterAll  
static void afterAllTests() {  
    System.out.println("Run this code after all tests");  
}

**Annotations to scale tests**

The @Disabled annotation disables the test method. JUnit engine skips any test methods that have the @Disabled annotation on them. This feature is useful for tests that fail because we have yet to implement the logic for the method under test. In such cases, we may want to skip the test for the time being. The developer can write and test the method locally without affecting the overall build output of the application.

@Test  
@Disabled  
public void testSortArray() {  
    fail("unimplemented method");  
}

We have a fail method in the test because we want this test to fail till all the functionality of the method under testing is complete. If the tests are run, the overall build will fail because of the one test that did not succeed. If we use the @Disabled annotation on this test and run tests again, the overall build will succeed because JUnit skips this test.

UnitTestingApplication.java

ArrayMethods.java

ArrayMethodsTest.java

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

        assertEquals(-1, arrayMethods.findIndex(new int[]{}, 1));

    }

    @Test

    @Disabled

    public void testSortArray() {

        fail("unimplemented method");

    }

    @BeforeEach

    void init(){

        //Initialize the object here

        System.out.println("Initializing before test");

        arrayMethods = new ArrayMethods();

    }

    @AfterEach

    void afterEachTest(){

        System.out.println("Clean up after test");

    }

    @BeforeAll

    static void beforeAllTests() {

        System.out.println("Run this code before all tests");

    }

    @AfterAll

    static void afterAllTests() {

        System.out.println("Run this code after all tests");

    }

}





Run

Save

Reset

It can be seen that the beforeAllTests and afterAllTests methods are run only once and the beforeEachTest and afterEachTest methods are run for every test. Also notice that the test for testingUnderDevelopmentMethod was not run because of the @Disabled annotation.

**Stubbing and Mocking**

Learn about stubs and mocks in this lesson and why mocks are preferred over stubs.

**We'll cover the following**

* [The need for stubbing and mocking](https://www.educative.io/courses/guide-spring-5-spring-boot-2/YMrWBpkgO9K#The-need-for-stubbing-and-mocking)
* [Creating a stub](https://www.educative.io/courses/guide-spring-5-spring-boot-2/YMrWBpkgO9K#Creating-a-stub)
* [Mockito dependency](https://www.educative.io/courses/guide-spring-5-spring-boot-2/YMrWBpkgO9K#Mockito-dependency)
* [Creating a mock](https://www.educative.io/courses/guide-spring-5-spring-boot-2/YMrWBpkgO9K#Creating-a-mock)

**The need for stubbing and mocking**

In unit testing, we should be able to test every application layer independently. A test for the service layer should not be dependent on the data layer. We use stubbing and mocking to test a part of our application by mocking an external service or an application layer. There are different mocking frameworks available like Mockito, JMock, and EasyMock

Suppose we have a StudentService class which has a findTotal method to perform the sum of an array. We have another class StudentDao that has a method to return data about student marks. In the findTotal method, we can fetch data using the StudentDao object and then find the total. Now every time the JUnit test runs, the StudentDao class will connect to a database and get marks’ data. This approach has two glaring issues. On one hand, it makes the test execution time longer, and on the other hand, it makes the test susceptible to failure in case the database connection is down. In such a scenario, our test will fail because of database connectivity issues and not because of any mistake in the logic of the code under test. To avoid this, we can either stub or mock the data source and simply test our code without interacting with the database.

In the same manner, if our application connects to a cloud service, we can mock the service instead of actually connecting to it every time.

1. We will create a method findTotal in the StudentService class to find the total of an array. It uses the StudentDao class that returns an array of marks obtained by a student. Our method uses the data returned by the StudentDao class to find the total of the array.

public class StudentService{  
  
    private StudentDao studentDao;  
  
   //other methods  
  
    int findTotal() {     
        //fetch student marks from a database  
        int[] array = studentDao.getMarks();  
   
        int sum = 0;  
      
        for(int value : array) {  
        sum += value;  
        }  
        return sum;  
    }  
}

The StudentDao class has a getMarks method along with other methods. We are not concerned with the actual implementation of getMarks because right now, we are testing the StudentService class.

public class StudentDao {     
    //...  
   
    int[] getMarks() {  
    //fetch and return data  
    return new int[] {};  
    }  
}

1. To test the findTotal method, we will create a new JUnit test called StudentServiceStubTest with a method testfindTotal as follows:

class StudentServiceStubTest {  
    @Test  
    void testfindTotal() {  
        StudentService studentService = new StudentService();  
        int total = studentService.findTotal();  
    }  
}

If the test is run like this, we will get a *NullPointerException* because we do not have any implementation of the getMarks method. Stubs provide a way to create a dummy implementation.

**Creating a stub**

1. To test the findTotal method, we can create a stub class that extends the Dao class. We can write an implementation of the getMarks method and return some dummy data back so that the findTotal method can process that data.

class StudentDaoStub extends StudentDao{  
    @Override  
    public int[] getMarks() {  
        return new int[]{15, 20, 5};  
    }  
}

Since StudentDao is a dependency of the StudentService class, we will pass the stub class to it. For this, we need a constructor in the StudentService class to initialize StudentDao.

public StudentService(StudentDao studentDao) {  
    super();  
    this.studentDao = studentDao;  
}

1. Now we can pass the stub class as a constructor argument when creating the StudentService object inside our test method. Since the result returned by the stub is already known, we can now check if the expected and actual values are the same using the assertEquals method.

@Test  
void testfindTotal() {  
    StudentService studentService = new StudentService(new StudentDaoStub());  
    int total = arrayMethods.findTotal();  
    assertEquals(40, total);  
}

This test will now run successfully as shown below:

UnitTestingApplication.java

StudentServiceStubTest.java

StudentDao.java

StudentService.java

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

package io.datajek.unittesting;

import static org.junit.jupiter.api.Assertions.\*;

import org.junit.jupiter.api.Test;

class StudentServiceStubTest {

    @Test

    void testFindTotal() {

        StudentService studentService = new StudentService(new StudentDaoStub());

        int total = studentService.findTotal();

        assertEquals(40, total);

    }

}

class StudentDaoStub extends StudentDao{

    //@Override

    public int[] getMarks() {

        return new int[]{15, 20, 5};

    }

}





Run

Save

Reset

1. Using stubs, we can imitate the StudentDao class and test our method. In the same way, stubs can also be created for an interface. This approach has some disadvantages. First, to make other test cases, we need to create another stub or another implementation of the StudentDao class. Creating multiple versions of the same class is cumbersome. Second, if the stub is an implementation of an interface and new methods are added to the interface, we will need to update the stub and provide an implementation for the unimplemented methods.

These problems are tackled by mocks.

**Mockito dependency**

1. We can use the mocking support provided by Spring Boot or add the dependency in **pom.xml**. Since we created our project using Spring initializr, JUnit and Mockito are automatically added as dependencies in the Maven dependencies folder and are available under the scope of test.

If this is not the case, the mockito dependency can be added as follows:

<dependency>  
    <groupId>org.mockito</groupId>  
    <artifactId>mockito-core</artifactId>  
    <version>2.10.0</version>  
    <scope>test</scope>  
</dependency>

**Creating a mock**

1. We will create a test for the findTotal method using Mockito. For this, we first create a new JUnit test called StudentServiceMockTest.

The mock method of org.mockito.Mockito creates a mock of a class if you pass it as an argument. Then, you can use the mock to do whatever you want it to. We will mock the StudentDao class as follows:

StudentDao studentDaoMock = mock(StudentDao.class);

The mock method needs a static import of org.mockito.Mockito.mock.

1. Two methods used on the mock are when and thenReturn, which help create a test scenario. When a method is called on studentDaoMock, return an array of numbers as follows:

when(studentDaoMock.getMarks()).thenReturn(new int[] {15, 20, 5});

The when method needs a static import of org.mockito.Mockito.when. Now we can pass the mock as a constructor argument to the StudentService object. The complete test method is shown below:

@Test  
void testfindTotal() {  
    StudentDao studentDaoMock = mock(StudentDao.class);  
    when(studentDaoMock.getMarks()).thenReturn(new int[] {15, 20, 5});  
      
    StudentService studentService = new StudentService(studentDaoMock);  
    int total = studentService.findTotal();  
 assertEquals(40, total);  
}

To create a stub, we need to create a complete class to implement the method and return values for the test case. In case of mocking, no implementation is needed. A mock can easily be repeated for other test case scenarios as well. We can mock the data just by using the when and thenReturn methods. Here is another test when an empty array is returned by the studentDaoMock:

@Test  
void testfindTotal\_EmptyArray() {  
    StudentDao studentDaoMock = mock(StudentDao.class);  
    when(studentDaoMock.getMarks()).thenReturn(new int[] {});  
          
    StudentService studentService = new StudentService(studentDaoMock);  
    int total = studentService.findTotal();  
    assertEquals(0, total);  
}

In this manner, we can mock the class and create tests for other scenarios as well. Here, we have used constructor injection for injecting studentDaoMock in StudentService. Setter injection can also be used. We created a mock for a class but mocks can also be created for interfaces.

UnitTestingApplication.java

StudentServiceMockTest.java

StudentDao.java

StudentService.java

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

import static org.junit.jupiter.api.Assertions.\*;

import org.junit.jupiter.api.Test;

import static org.mockito.Mockito.mock;

import static org.mockito.Mockito.when;

class StudentServiceMockTest {

    @Test

    void testFindTotal() {

        StudentDao studentDaoMock = mock(StudentDao.class);

        when(studentDaoMock.getMarks()).thenReturn(new int[] {15, 20, 5});

        StudentService studentService = new StudentService(studentDaoMock);

        int total = studentService.findTotal();

        assertEquals(40, total);

    }

    @Test

    void testFindTotal\_EmptyArray() {

        StudentDao studentDaoMock = mock(StudentDao.class);

        when(studentDaoMock.getMarks()).thenReturn(new int[] {});

        StudentService studentService = new StudentService(studentDaoMock);

        int total = studentService.findTotal();

        assertEquals(0, total);

    }

}





Run

Save

Reset

Back

JUnit Test Lifecycle Annotations

**Mockito Annotations**

Learn about some of the annotations provided by Mockito to make tests shorter and readable.

**We'll cover the following**

* [@Mock](https://www.educative.io/courses/guide-spring-5-spring-boot-2/xVDwEZjyJqq#@Mock)
* [@InjectMocks](https://www.educative.io/courses/guide-spring-5-spring-boot-2/xVDwEZjyJqq#@InjectMocks)
* [@ExtendWith](https://www.educative.io/courses/guide-spring-5-spring-boot-2/xVDwEZjyJqq#@ExtendWith)

There are some Mockito annotations that help minimize the code for creating and injecting mocks. While writing tests, there can be some repetitive statements. We will use annotations provided by Mockito to get rid of the redundant code and make our tests more readable.

**@Mock**

1. The @Mock annotation is used to create a mock without calling the Mockito.mock method.

Instead of creating a mock inside every test, we can move this statement outside and use the @Mock annotation on it. If we create mocks in the following way, we are using repetitive statements in every test:

StudentDao studentDaoMock = mock(StudentDao.class);

Rather than repeating the above statement in every test, we can create the mocks outside the test method using the @Mock annotation as follows:

@Mock  
StudentDao studentDaoMock;

By using the @Mock annotation, we will create the mock once instead of creating it in every test.

**@InjectMocks**

1. This annotation automatically injects a mock object into the object being tested. We have been passing the mock object to the StudentService object as a constructor argument in the test method as follows:

StudentService studentService = new StudentService(studentDaoMock);

We can do the dependency injection outside the test method and use the @InjectMocks annotation as follows:

@InjectMocks  
StudentService studentService;

1. The tests become shorter when we remove the *create and inject* mock steps from the test method. We can further shorten the method by making the total variable inline. For comparison, a test written with and without annotations is shown below:

//without annotations  
@Test  
void testFindTotal() {  
    StudentDao studentDaoMock = mock(StudentDao.class);  
    when(studentDaoMock.getMarks()).thenReturn(new int[] {15, 20, 5});  
      
    StudentService studentService = new StudentService(studentDaoMock);  
    int total = studentService.findTotal();  
    assertEquals(40, total);  
}

//with annotations  
@Mock  
StudentDao studentDaoMock;  
   
@InjectMocks  
StudentService studentService;  
   
@Test  
void testFindTotal() {  
    when(studentDaoMock.getMarks()).thenReturn(new int[] {15, 20, 5});  
    assertEquals(40, studentService.findTotal());  
}

The test method is short and the test is more readable when Mockito annotations are used.

**@ExtendWith**

1. When the @Mock annotation is used, by default, JUnit does not evaluate it. We need to use the @ExtendWith annotation and pass MockitoExtension.class as an argument to integrate Mockito with the JUnit 5 extension model. MockitoExtension enables the evaluation of the @Mock annotations to initialize mocks, which was previously done using the Mockito.mock method.

Using the Mockito extension, mocks are initialized before each test method and validation is performed after each test method to check if the mock was used in the method.

@ExtendWith(MockitoExtension.class)

These three annotations make the tests very simple and readable.

For the sake of comparison, the tests written using repetitive stubbing and mocking are also reproduced below in addition to the ones written using Mockito annotations.

UnitTestingApplication.java

StudentServiceMockitoAnnotationsTest.java

StudentServiceMockTest.java

StudentDao.java

StudentService.java

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

package io.datajek.unittesting;

import static org.junit.jupiter.api.Assertions.assertEquals;

import static org.mockito.Mockito.when;

import org.junit.jupiter.api.Test;

import org.junit.jupiter.api.extension.ExtendWith;

import org.mockito.InjectMocks;

import org.mockito.Mock;

import org.mockito.junit.jupiter.MockitoExtension;

@ExtendWith(MockitoExtension.class)

class StudentServiceMockitoAnnotationsTest {

    @Mock

    StudentDao studentDaoMock;

    @InjectMocks

    StudentService studentService;

    @Test

    void testFindTotal() {

        when(studentDaoMock.getMarks()).thenReturn(new int[] {15, 20, 5});

        assertEquals(40, studentService.findTotal());

    }

    @Test

    void testFindTotal\_EmptyArray() {

        when(studentDaoMock.getMarks()).thenReturn(new int[] {});

        assertEquals(0, studentService.findTotal());

    }





Run

Save

Reset

Back

Stubbing and Mocking

**Unit Testing in Spring**

In this lesson, we will write two simple unit tests using the JUnit framework to test a method.

**We'll cover the following**

* [Creating a test file](https://www.educative.io/courses/guide-spring-5-spring-boot-2/gk531EX1Q3D#Creating-a-test-file)
* [Initializing the bean](https://www.educative.io/courses/guide-spring-5-spring-boot-2/gk531EX1Q3D#Initializing-the-bean)
* [Using assertArrayEquals to check the output](https://www.educative.io/courses/guide-spring-5-spring-boot-2/gk531EX1Q3D#Using-assertArrayEquals-to-check-the-output)

When writing tests, the foremost thing to remember is that the test code should always be separate from the production code. We will write all our tests in **src/test/java**. This will ensure that the test code is never part of the deployable jar or war and stays in our repository.

We will now write unit tests for the RecommenderImplementation class. The code of the class is reproduced below:

@Component  
public class RecommenderImplementation {  
    @Autowired  
    private Filter filter;  
              
    public RecommenderImplementation(Filter filter) {  
    super();  
    this.filter = filter;  
    }  
  
    //use a filter to find recommendations  
    public String [] recommendMovies (String movie) {  
    String[] results = filter.getRecommendations(movie);  
    return results;  
    }  
}

This class has a dependency on the Filter interface, which has two implementations, ContentBasedFilter and CollaborativeFilter. Autowiring of the Filter is done by type using the @Primary annotation.

Both implementations of the interface are reproduced below. We have used hardcoded return values in both classes.

@Component  
@Primary  
public class CollaborativeFilter implements Filter{  
  
    public String[] getRecommendations(String movie) {  
    //logic of collaborative filter  
    return new String[] {"Finding Nemo", "Ice Age", "Toy Story"};  
    }  
}

@Component  
public class ContentBasedFilter implements Filter{  
  
    public String[] getRecommendations(String movie) {  
    //implement logic of content based filter  
    return new String[] {"Happy Feet", "Ice Age", "Shark Tale"};  
    }  
}

**Creating a test file**

1. We will begin by creating a JUnit test case in **src/test/java**. To create a unit test for the RecommenderImplementation class, we need to match the same package structure in the test file. The test is called RecommenderImplementationUnitTest. Since we want to test the recommendMovies method, we will rename the test method to testRecommendMovies.

class RecommenderImplementationUnitTest {  
    @Test  
    void testRecommendMovies() {  
  
    }  
}

1. Since this test is for the RecommenderImplementation class, we need an object of this class which will be used to call the method being tested (recommendMovies).

The following snippet shows the outline of our unit test:

class RecommenderImplementationUnitTest {  
  
    @Test  
    void testRecommendMovies() {  
        //1. Initialize the object  
        RecommenderImplementation recommenderImpl;  
        //2. Call method on the bean  
        String[] actualResult = recommenderImpl.recommendMovies("Finding Dory");  
        //3. Check if the result is as expected  
    }  
}

**Initializing the bean**

1. To initialize the RecommenderImplementation bean, we will use constructor injection in the test method. Since we are initializing the bean with CollaborativeFilter, we will rename the test for readability.

@Test  
void testRecommendMovies\_withCollaborativeFilter() {  
    RecommenderImplementation recommenderImpl = new RecommenderImplementation(  
                                                       new CollaborativeFilter());  
    //2. Call method on the bean          
    String[] actualResult = recommenderImpl.recommendMovies("Finding Dory");  
    //3. Check if the result is as expected  
}

Having a constructor in the class enables us to initialize the dependency without having to load the Spring Boot application context. When writing code, field injection should be avoided because then the bean can only be initialized by loading the context.

**Using assertArrayEquals to check the output**

1. Inside the test method, we will call the recommendMovies method with an input string. The input doesn’t really matter because we have hardcoded the return values. Since our method returns an array, we will use the assertArrayEquals method to compare the expected values with actual values.

@Test  
void testRecommendMovies\_withCollaborativeFilter() {  
    RecommenderImplementation recommenderImpl = new RecommenderImplementation(  
                                                        new CollaborativeFilter());  
    String[] actualResult = recommenderImpl.recommendMovies("Finding Dory");  
    assertArrayEquals(new String[] {"Finding Nemo", "Ice Age", "Toy Story"}, actualResult);  
}

The test is shown in the code widget below. When run, the test succeeds. To check if it is actually working, you can either change the values returned by the method or change the expected values in the assertArrayEquals method.

1. Since we have two implementations of the Filter interface, we can write another unit test to check the output of the recommendMovies method when ContentBasedFilter is used as follows:

@Test  
void testRecommendMovies\_withContentBasedFilter() {  
    RecommenderImplementation recommenderImpl = new RecommenderImplementation(  
                                                        new ContentBasedFilter());  
    assertArrayEquals(new String[] {"Happy Feet", "Ice Age", "Shark Tale"},   
                                 recommenderImpl.recommendMovies("Finding Dory"));  
}

Here, we have initialized the RecommenderImplementation object with ContentBasedFilter, which leads to a different output when the recommendMovies method is called.

MovieRecommenderSystemApplication.java

RecommenderImplementation.java

RecommenderImplementationUnitTest.java

Filter.java

CollaborativeFilter.java

ContentBasedFilter.java

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

package io.datajek.spring.basics.movierecommendersystem.lesson4;

import static org.junit.jupiter.api.Assertions.\*;

import org.junit.jupiter.api.Test;

import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.test.context.SpringBootTest;

import org.springframework.test.context.ContextConfiguration;

import org.springframework.test.context.junit4.SpringRunner;

class RecommenderImplementationUnitTest {

    @Test

    void testRecommendMovies\_withCollaborativeFilter() {

        RecommenderImplementation recommenderImpl = new RecommenderImplementation(new CollaborativeFilter());

        assertArrayEquals(new String[] {"Finding Nemo", "Ice Age", "Toy Story"}, recommenderImpl.recommendMovies("Finding Dory"));

    }

    @Test

    void testRecommendMovies\_withContentBasedFilter() {

        RecommenderImplementation recommenderImpl = new RecommenderImplementation(new ContentBasedFilter());

        assertArrayEquals(new String[] {"Happy Feet", "Ice Age", "Shark Tale"}, recommenderImpl.recommendMovies("Finding Dory"));

    }

}





Run

Save

Reset

Back

Mockito Annotations

**Unit Testing Using Spring Boot**

In this lesson, we will write a test using the @SpringBootTest annotation and show why it should not be used for unit testing.

**We'll cover the following**

* [spring-boot-starter-test dependency](https://www.educative.io/courses/guide-spring-5-spring-boot-2/xoN67vZQjDJ#spring-boot-starter-test-dependency)
* [Launching application context and autowiring bean](https://www.educative.io/courses/guide-spring-5-spring-boot-2/xoN67vZQjDJ#Launching-application-context-and-autowiring-bean)
* [When to use @SpringBootTest](https://www.educative.io/courses/guide-spring-5-spring-boot-2/xoN67vZQjDJ#When-to-use-@SpringBootTest)

When we create an application using Spring Boot, we get the springboot-starter-test dependency that comes with Mockito and AssertJ as testing libraries. We also automatically get a test file with the @SpringBootTest annotation.

Unit testing should not be done using Spring Boot. The main purpose of unit testing is to test a method or class. However, @SpringBootTest loads the entire context, which makes the test lengthy and defeats the purpose of unit testing. This feature of Spring Boot should be used in integration testing where we test across multiple layers.

The purpose of this lesson is only to show how a unit test can be written using the @SpringBootTest annotation.

**spring-boot-starter-test dependency**

1. The **pom.xml** file of a Spring Boot project has the following dependency on spring-boot-starter-test:

<dependency>  
   <groupId>org.springframework.boot</groupId>  
   <artifactId>spring-boot-starter-test</artifactId>  
   <scope>test</scope>  
</dependency>

We will begin by creating a JUnit test case in **src/test/java**, taking care that we match the package structure of the class being tested with the test file. The test is called **RecommenderImplementationSpringBootTest** as we want to show how to use test features provided by Spring Boot. We will rename the test method testRecommendMovies.

package io.datajek.spring.basics.movierecommendersystem.lesson4;  
  
import org.springframework.boot.test.context.SpringBootTest;  
  
@SpringBootTest  
class RecommenderImplementationSpringBootTest {  
  
    @Test  
    void testRecommendMovies() {  
        //...  
    }  
}

1. Since this test is for the **RecommenderImplementation** class, we will begin by defining an object of this class.

class RecommenderImplementationSpringBootTest {  
  
    private RecommenderImplementation recommenderImpl;  
  
    @Test  
    void testRecommendMovies() {  
        //...  
    }  
}

In this lesson, we will learn how the bean can be obtained from the application context. The following snippet shows the outline of our unit test:

//1. Launch context  
class RecommenderImplementationSpringBootTest {  
  
     //2. Load bean from context  
     private RecommenderImplementation recommenderImpl;  
  
     @Test  
     void testRecommendMovies() {  
         //3. Call method on the bean  
         //4. Check if the result is as expected  
     }  
}

**Launching application context and autowiring bean**

1. @SpringBootTest launches the entire application context, which means that all the beans get loaded. Thus, we can simply use the @Autowired annotation to get the RecommenderImplementation bean.

@Autowired  
private RecommenderImplementation recommenderImpl;

The @Autowired will automatically find and inject the dependency rather than us having to manually do it (like in the previous lesson where we used constructor injection to initialize the bean).

1. Inside the test method, we will call the recommendMovies method with an input string and then use the assertArrayEquals method to compare the expected and actual values.

@Test  
void testRecommendMovies() {  
    assertArrayEquals(new String[] {"Finding Nemo", "Ice Age", "Toy Story"},   
                               recommenderImpl.recommendMovies("Finding Dory"));  
}

This simple test class is shown in the code widget below. You can compare the running time of this test with the running time of the test created when using Spring application. This unit test will take longer because we are launching the application context to run the test.

MovieRecommenderSystemApplication.java

RecommenderImplementationSpringBootTest.java

RecommenderImplementation.java

Filter.java

CollaborativeFilter.java

ContentBasedFilter.java

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

package io.datajek.spring.basics.movierecommendersystem.lesson4;

import static org.junit.jupiter.api.Assertions.assertArrayEquals;

import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest

class RecommenderImplementationSpringBootTest {

    @Autowired

    private RecommenderImplementation recommenderImpl;

    @Test

    void testRecommendMovies() {

        assertArrayEquals(new String[] {"Finding Nemo", "Ice Age", "Toy Story"}, recommenderImpl.recommendMovies("Finding Dory"));

    }

}





Run

Save

Reset

**When to use @SpringBootTest**

The value of @SpringBootTest is in testing the whole application because it loads the complete context like in the production environment. For a unit test, there is no need to load the complete context. In fact, loading the entire context will have an impact on performance since unit tests should only take a few milliseconds to run.

**Unit Testing for Java Context**

Unit tests can be written by launching a part of the context as shown in this lesson.

**We'll cover the following**

* [spring-test and junit dependency](https://www.educative.io/courses/guide-spring-5-spring-boot-2/m7M6vj5nKBO#spring-test-and-junit-dependency)
* [@ContextConfiguration](https://www.educative.io/courses/guide-spring-5-spring-boot-2/m7M6vj5nKBO#@ContextConfiguration)

This lesson shows how to write a unit test using the Spring test features. We will remove the springboot-starter-test dependency from the pom.xml file, write our own dependencies and then use them to test the RecommenderImplementation class.

**spring-test and junit dependency**

1. We will start by replacing the springboot-starter-test dependency — which is automatically included in a Spring Boot project — with the spring-test dependency along with the junit-jupiter-engine dependency.

<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-test</artifactId>  
 <scope>test</scope>  
</dependency>  
   
<dependency>  
 <groupId>org.junit.jupiter</groupId>  
 <artifactId>junit-jupiter-engine</artifactId>  
 <scope>test</scope>  
</dependency>

spring-test is the module in the Spring framework that helps us write unit tests. Since the project was built using Spring Boot, removing the springboot-starter-test dependency leads to compilation errors due to usage of the @SpringBootTest annotation in src/test/java. To remove the errors, you can comment out the @SpringBootTest annotation.

1. We will create a JUnit test case in src/test/java</ called RecommenderImplementationSpringTest. The package structure of the test class must match that of the class being tested. The test class is shown:

package io.datajek.spring.basics.movierecommendersystem.lesson4;  
  
class RecommenderImplementationSpringTest {  
  
    @Test  
    void testRecommendMovies() {  
         //...  
    }  
}

**@ContextConfiguration**

1. @SpringBootTest can be used to launch the context and get the bean in the test class when using Spring Boot. To load the context using Spring, we will use @ContextConfiguration. This loads only part of the configuration and is thus more suitable for unit testing. Since our configuration is in the class MovieRecommenderSystemApplication, we will pass that as an argument to the @ContextConfiguration annotation. This class is where the context is present.

After the context has been defined, we need to run it. This is done by using the @ExtendWith annotation with SpringExtension.class as its argument. SpringExtension provides a bridge between Spring and JUnit.

//load context and run it  
@ExtendWith(SpringExtension.class)  
@ContextConfiguration(classes=MovieRecommenderSystemApplication.class)  
class RecommenderImplementationSpringTest {  
  
}

1. The rest of the test is the same as the test created using Spring Boot. The @Autowired annotation is used to get the RecommenderImplementation bean from the context once it is launched and loaded. Then the assertArrayEquals method is used to test whether the actual output of recommendMovies is the same as the expected output.

//load context  
@ExtendWith(SpringExtension.class)  
@ContextConfiguration(classes=MovieRecommenderSystemApplication.class)  
class RecommenderImplementationSpringTest {  
   
    @Autowired  
    private RecommenderImplementation recommenderImpl;  
   
    @Test  
    public void testRecommendMovies() {  
    assertArrayEquals(new String[] {"Finding Nemo", "Ice Age", "Toy Story"},   
                                 recommenderImpl.recommendMovies("Finding Dory"));  
    }  
}

MovieRecommenderSystemApplication.java

RecommenderImplementationSpringTest.java

RecommenderImplementation.java

Filter.java

CollaborativeFilter.java

ContentBasedFilter.java

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

package io.datajek.spring.basics.movierecommendersystem.lesson4;

import static org.junit.jupiter.api.Assertions.assertArrayEquals;

import org.junit.jupiter.api.Test;

import org.junit.jupiter.api.extension.ExtendWith;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.test.context.ContextConfiguration;

import org.springframework.test.context.junit.jupiter.SpringExtension;

//load context

@ExtendWith(SpringExtension.class)

@ContextConfiguration(classes=MovieRecommenderSystemApplication.class)

class RecommenderImplementationSpringTest {

    @Autowired

    private RecommenderImplementation recommenderImpl;

    @Test

    public void testRecommendMovies() {

        assertArrayEquals(new String[] {"Finding Nemo", "Ice Age", "Toy Story"}, recommenderImpl.recommendMovies("Finding Dory"));

    }

}





Run

Save

Reset

A class which does not provide a constructor or setter injection for its dependencies can only be tested by loading the Spring context. However, if a constructor is present (as in the case of RecommenderImplementation class shown above), the object can be created without loading the context and it can also be mocked.

**Unit Testing for XML Context**

Learn how to write a unit test that uses the XML configuration.

**We'll cover the following**

* [@ContextConfiguration with locations](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qZz2vM3y4v7#@ContextConfiguration-with-locations)
* [Creating a test configuration](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qZz2vM3y4v7#Creating-a-test-configuration)

The @ContextConfiguration annotation is used to load Java as well as XML context. We created an XML application context file for the MovieRecommenderSystemApplication. Following is the XML configuration file:

<beans xmlns="http://www.springframework.org/schema/beans"  
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
       xmlns:context="http://www.springframework.org/schema/context"  
       xsi:schemaLocation="http://www.springframework.org/schema/beans  
           http://www.springframework.org/schema/beans/spring-beans.xsd  
           http://www.springframework.org/schema/context  
           http://www.springframework.org/schema/context/spring-context.xsd">  
  
    <context:component-scan base-package=  
             "io.datajek.spring.basics.movierecommendersystem.lesson14" use-default-filters="false">  
    </context:component-scan>  
    <!-- enable detection of @Autowired annotation -->  
    <context:annotation-config/>  
    
    <bean id="filter"   
        class="io.datajek.spring.basics.movierecommendersystem.lesson14.ContentBasedFilter"/>      
    <bean id="filter2"   
        class="io.datajek.spring.basics.movierecommendersystem.lesson14.CollaborativeFilter"/>  
    <bean id="recommenderImpl" class=  
              "io.datajek.spring.basics.movierecommendersystem.lesson14.RecommenderImplementation"/>  
  
</beans>

In this file, we have mentioned the package for component scanning using the <context:component-scan> tag. We have also defined three beans: filter, filter2, and recommenderImpl.

**@ContextConfiguration with locations**

1. We will create a test called RecommenderImplementationXmlConfigTest in **src/test/java/** for testing the RecommenderImplementation class. When using Java context, the @ContextConfiguration annotation took classes as an argument. To load the XML configuration, we will provide locations as an argument. Since the config file is in the class path, the location can be given as follows:

@ContextConfiguration(locations="/appContext.xml")

The above line will load the file **appContext.xml** that contains the definition of beans and their dependencies.

1. The rest of the test will remain the same as done using the Java context:

@ExtendWith(SpringExtension.class)  
@ContextConfiguration(locations="/appContext.xml")  
class RecommenderImplementationXmlConfigTest {  
   
    @Autowired  
    private RecommenderImplementation recommenderImpl;  
   
    @Test  
    void testRecommendMovies() {  
        assertArrayEquals(new String[] {"Happy Feet", "Ice Age", "Shark Tale"},   
                                recommenderImpl.recommendMovies("Finding Dory"));  
    }  
}

MovieRecommenderSystemApplication.java

RecommenderImplementationXmlConfigTest.java

appContext.xml

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

package io.datajek.spring.basics.movierecommendersystem.lesson14;

import static org.junit.jupiter.api.Assertions.assertArrayEquals;

import org.junit.jupiter.api.Test;

import org.junit.jupiter.api.extension.ExtendWith;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.test.context.ContextConfiguration;

import org.springframework.test.context.junit.jupiter.SpringExtension;

//load context

@ExtendWith(SpringExtension.class)

@ContextConfiguration(locations="/appContext.xml")

class RecommenderImplementationXmlConfigTest {

    @Autowired

    private RecommenderImplementation recommenderImpl;

    @Test

    void testRecommendMovies() {

        assertArrayEquals(new String[] {"Happy Feet", "Ice Age", "Shark Tale"}, recommenderImpl.recommendMovies("Finding Dory"));

    }

}





Run

Save

Reset

**Creating a test configuration**

1. Since we are reading the application context from a file, we can create a separate context for the purpose of testing. All test contexts should be placed in **src/test/resources**. A test context is useful if we want to override something defined in the context, for the purpose of testing.

To create a test context, we first need to create a folder called **resources** under **src/test**. Next, we will create an XML file **testContext.xml** in **src/test/resources**. We can define a test context here or import the context from another file and then over-ride it.

The following code shows how to import the application context from **src/main/resources**:

<beans xmlns="http://www.springframework.org/schema/beans"  
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xmlns:context="http://www.springframework.org/schema/context"  
    xsi:schemaLocation="http://www.springframework.org/schema/beans  
        http://www.springframework.org/schema/beans/spring-beans.xsd  
        http://www.springframework.org/schema/context  
        http://www.springframework.org/schema/context/spring-context.xsd">  
  
    <import resource="classpath:appContext.xml"/>  
    <!-- Now you can change anything in the application context and override it -->  
    <!-- You can change/delete bean definitions -->  
  
</beans>

After importing the context, the beans can be defined for the purpose of testing and any bean from the original context can be over-ridden.

1. To use the test configuration defined above, we will use **testContext.xml** as an argument to the @ContextConfiguration annotation.

@ExtendWith(SpringExtension.class)  
@ContextConfiguration(locations="/testContext.xml")  
class RecommenderImplementationXmlConfigTest {  
    //...  
}

In this way, we can define a separate configuration for the unit test.

**Spring Unit Testing with Mockito**

Let's rewrite the unit test by mocking the dependency using Mockito.

**We'll cover the following**

* [mockito-core dependency](https://www.educative.io/courses/guide-spring-5-spring-boot-2/q2pDLnY32E2#mockito-core-dependency)
* [Mocking an interface](https://www.educative.io/courses/guide-spring-5-spring-boot-2/q2pDLnY32E2#Mocking-an-interface)

The RecommenderImplementation class has a dependency on the Filter interface. Ideally, when writing unit tests, we should only focus on the class under test and mock the dependencies instead of initializing the dependency. We will now write a test for the same class using Mockito.

**mockito-core dependency**

1. To test using the Mockito framework, we need the mockito-core dependency, which can be added to **pom.xml** as follows:

<dependency>  
  <groupId>org.mockito</groupId>  
  <artifactId>mockito-core</artifactId>  
  <scope>test</scope>  
</dependency>

Note that this dependency is automatically available if springboot-starter-test is included in the **pom.xml** file.

1. We are testing the RecommenderImplementation class that depends on the Filter interface and gets data from it by calling the getRecommendations method of the interface. This is not good practice because we are not testing the interface right now. We should not be calling the interface method when testing our class.

public String [] recommendMovies (String movie) {  
    //call interface method  
    String[] results = filter.getRecommendations(movie);  
    //rank the recommendations based   
    return results;  
}

The getRecommendations method has hardcoded return values, but in reality, this is something that changes based on the input string. If the functionality of the method changes or if another 3rd party implementation of the interface is used, our unit test will start failing. We do not want this kind of dependency for our unit test.

The same scenario can be applied when a class in the business layer calls a method of a class in the data layer. For unit tests, we should not go across layers.

**Mocking an interface**

1. Mockito can be used when we want to test the interface with different kinds of data. We will mock the interface using the @Mock annotation as follows:

@Mock  
private Filter mockFilter;

This mockFilter can be injected in the bean using the @InjectMocks annotation.

@InjectMocks  
private RecommenderImplementation recommenderImpl;

We need the @ExtendWith annotation with MockitoExtension.class to enable the use of the annotations mentioned above.

1. Now we are in a position to mock the results of the getRecommendations method using the when and thenReturn methods on the mock object. We will mock the case when no recommendation is found.

when(mockFilter.getRecommendations("Finding Dory")).thenReturn(new String[] {});

Here, we are intercepting the call to getRecommendations and returning a hard coded result without calling the interface method.

1. The test for a case when no recommendation is returned is given below:

@Test  
void testRecommendMovies\_noRecommendationsFound() {  
    when(mockFilter.getRecommendations("Finding Dory")).thenReturn(new String[] {});  
    assertArrayEquals(new String[] {}, recommenderImpl.recommendMovies("Findng Dory"));  
}

Because the mock returned an empty array, the output of the recommendMovies method will also be an empty array. This code is given in the code widget below.

MovieRecommenderSystemApplication.java

RecommenderImplementationMockTest.java

RecommenderImplementation.java

Filter.java

CollaborativeFilter.java

ContentBasedFilter.java

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

package io.datajek.spring.basics.movierecommendersystem.lesson4;

import static org.junit.jupiter.api.Assertions.assertArrayEquals;

import static org.mockito.Mockito.when;

import org.junit.jupiter.api.Test;

import org.junit.jupiter.api.extension.ExtendWith;

import org.mockito.InjectMocks;

import org.mockito.Mock;

import org.mockito.junit.jupiter.MockitoExtension;

@ExtendWith(MockitoExtension.class)

class RecommenderImplementationMockTest {

    @InjectMocks

    private RecommenderImplementation recommenderImpl;

    @Mock

    private Filter mockFilter;

    @Test

    void testRecommendMovies\_noRecommendationsFound() {

        when(mockFilter.getRecommendations("Finding Dory")).thenReturn(new String[] {});

        assertArrayEquals(new String[] {}, recommenderImpl.recommendMovies("Finding Dory"));

    }

}





Run

Save

Reset

Using Mockito, the tests take less time as there is no need to load the application context. Whenever possible, try to avoid using Spring in unit testing as it makes the turnaround time longer.

Spring also provides annotations for testing a particular layer like @WebMvcTest and @DataJpaTest that load only a part of the context and mock the rest of the dependencies.

# Interview Questions

# Spring Framework

**We'll cover the following**

* [Spring framework interview questions](https://www.educative.io/courses/guide-spring-5-spring-boot-2/g7YgjonOGM9#Spring-framework-interview-questions)
  + [What are the major features of Spring 5?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/g7YgjonOGM9#What-are-the-major-features-of-Spring-5?)
  + [Is Spring 5 compatible with older versions of Java?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/g7YgjonOGM9#Is-Spring-5-compatible-with-older-versions-of-Java?)
  + [What are the advantages of using Spring framework?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/g7YgjonOGM9#What-are-the-advantages-of-using-Spring-framework?)
  + [List the modules of Spring framework.](https://www.educative.io/courses/guide-spring-5-spring-boot-2/g7YgjonOGM9#List-the-modules-of-Spring-framework.)
  + [What is tight coupling?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/g7YgjonOGM9#What-is-tight-coupling?)
  + [What is loose coupling?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/g7YgjonOGM9#What-is-loose-coupling?)
  + [What is the purpose of application.properties file?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/g7YgjonOGM9#What-is-the-purpose-of-application.properties-file?)
  + [What is pom.xml file?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/g7YgjonOGM9#What-is-pom.xml-file?)
  + [What design patterns are used in Spring framework?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/g7YgjonOGM9#What-design-patterns-are-used-in-Spring-framework?)
  + [What are some of the best practices for Spring Framework?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/g7YgjonOGM9#What-are-some-of-the-best-practices-for-Spring-Framework?)
  + [Describe some standard Spring events.](https://www.educative.io/courses/guide-spring-5-spring-boot-2/g7YgjonOGM9#Describe-some-standard-Spring-events.)
  + [How does Spring 5 make use of JDK 9 modularity?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/g7YgjonOGM9#How-does-Spring-5-make-use-of-JDK-9-modularity?)

## Spring framework interview questions

### What are the major features of Spring 5?

Spring 5 was released in 2017. Its major features include:

* It requires a minimum of Java 8.
* It enables use of lambda expressions.
* It is based on servlet 4.0 API.
* It includes the reactive programming framework **WebFlux**.
* Spring 5 supports Kotlin for functional programming.
* It enables JUnit 5 support for testing.

### Is Spring 5 compatible with older versions of Java?

No, because Spring 5 requires a minimum of Java 8. It is not compatible with older versions of Java as the code base of Spring has been revamped to take advantage of the features of Java 8.

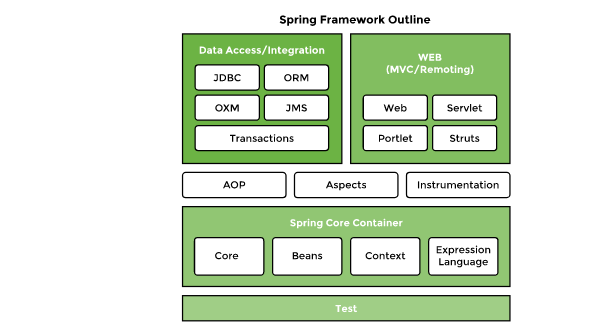
### What are the advantages of using Spring framework?

* Spring framework is open source.
* Spring framework was designed to reduce boilerplate code.
* Dependency injection makes it possible to write loosely coupled code which is easy to modify and maintain.
* Spring architecture is modular which makes it a lightweight framework. The developer has the choice to pick and choose only those modules which are needed for the application.
* Spring also provides excellent integration with a number of frameworks like Hibernate, Struts, EJB etc.
* The autoconfiguration feature reduces the development time where the developer only focuses on the business logic while Spring takes care of the configuration details.
* Spring supports AOP, which separates business logic from system services.
* Spring provides an API for translating technology-specific exceptions into unchecked exceptions.
* Spring supports testable code and test driven development (TDD).

### List the modules of Spring framework.

Spring architecture is modular. The modules can be grouped together based on their functionality.

* The Spring Core Container has Core, Beans, Context and Expression language modules.
* The data access or integration layer has JDBC, ORM, OXM, JSM and Transaction modules.
* The web layer has Web, Servlet, Portlet, Struts etc modules.
* Other modules include Test, AOP, Aspects and Instrumentation.



### What is tight coupling?

When two classes are dependent on each other such that class A needs an object of class B, they are said to be coupled. Tight coupling involves class A creating an instance of class B. In such a scenario, if we want to use another class in place of class B, it would require changing the code in class A.

For example, Vehicle class is dependent on CombustionEngine class and creates its object as follows:

class Vehicle{  
    
    private CombustionEngine engine = new CombustionEngine();  
    //...  
}

If we want to change from CombustionEngine class to ElectricEngine class, it will require modification of code of the Vehicle class.

### What is loose coupling?

Loose coupling removes the dependency of a class on an object. It is achieved by introducing an interface. Now class A is dependent on an interface with say, 2 implementations; class B and class C. The dependency can be plugged in class A by using a setter method or by using a constructor taking an object of the interface type as an argument. In this way switching from class B to class C does not involve changing the code of class A.

For example, Engine class is a dependency of Vehicle class. We can create an Engine interface that has two implementations, CombustionEngine and ElectricEngine.

interface Engine{  
  
    //...  
}

class CombustionEngine implements Engine{    
  
    //...  
}

class ElectricEngine implements Engine{    
  
    //...  
}

There is loose coupling between Vehicle and Engine. In the Vehicle class, either type of engine can be used without any change to the code.

class Vehicle{  
    
    private Engine engine;  
  
    Vehicle(Engine engine){  
        this.engine = engine;  
    }  
  
    //...  
}

### What is the purpose of **application.properties** file?

**application.properties** file is used to configure property values to run the application in different environments. It can be used to configure database details and log generation etc.

**application.properties** file is present in **src/main/resources** directory.

### What is **pom.xml** file?

**POM** stands for Project Object Model. It is an XML file in which the configuration of a Maven project is defined. The <dependency> tag lists all the dependencies with <groupid>, <artifactid>, <version> and <packaging> tags providing necessary details about the dependencies. Maven uses the information in **pom.xml** to build the project.

### What design patterns are used in Spring framework?

Spring framework uses a number of design patterns:

* **Factory Pattern**: BeanFactory and ApplicationContext classes
* **Singleton Pattern**: Singleton-scoped beans
* **Prototype Pattern**: Prototype-scoped beans
* **Proxy Pattern**: Spring Aspect Oriented Programming support
* **Template Method Pattern**: JdbcTemplate, JmsTemplate, JpaTemplate etc.
* **Data Access Object Pattern**: Spring DAO support
* **Model View Controller Pattern**: Spring MVC
* **Front Controller Pattern**: DispatcherServlet in Spring MVC
* **View Helper Pattern**: custom JSP tags separate code from presentation in views.
* **Adapter Pattern**: JMS adapters and JDBC adapters in Spring Integration

### What are some of the best practices for Spring Framework?

* Only use the modules that are needed by the application and remove any extra modules that get added when Spring Tools Suite is used.
* Use annotation according to the application layer. @Service for business logic and @Repository for data layer.
* Create an external property file to configure property values.
* Use Spring Initializer for starting new Spring Boot projects to get a tested and approved set of dependencies and to save time.
* Use Controller classes to delegate business logic to the service layer.
* Use a logging framework instead of doing it manually using System.out.println().

### Describe some standard Spring events.

Events provide a way of communication between loosely coupled components. **ApplicationContext** in Spring publishes certain events during the lifecycle of a Spring application.

Standard Spring events include:

* **ContextStartedEvent**: triggered when the context is started.
* **ContextRefreshedEvent**: triggered when the context is either initialized or refreshed using the refresh() method.
* **ContextStoppedEvent**: triggered when the context is stopped using the stop() method.
* **ContextClosedEvent**: triggered when the close() method is called on the ApplicationContext.
* **RequestHandledEvent**: triggered to let all beans know that an HTTP request has been handled.

### How does Spring 5 make use of JDK 9 modularity?

Java 9 introduces modules which are a reusable collection of related packages, resources (XML files) and a module descriptor that specifies name, dependencies, and packages that are available for use.

A Spring 5 project can be built around this concept. Consider the following example in which we have a ModuleExample class in the **io.datajek.modularitydemo** package.

package io.datajek.jdkmodularity;  
public class ModuleExample{  
   public String displayMessage(){  
       return "Spring 5 with JDK 9 Modularity";  
   }  
}

To make this package available to another package, we can make it part of a module. This is done using the module keyword. The export keyword is used to specify the package whose public methods are available to other modules.

module io.datajek.modularityexample {  
   export io.datajek.jdkmodularity;  
}

This module can be used in any other java project by using the require keyword. This defines a dependency relationship where **io.datajek.modularityexample.demo** depends on **io.datajek.modularityexample**.

module io.datajek.modularityexample.demo {  
   requires io.datajek.modularityexample;  
}

The ModuleExample class and its public methods can now be accessed in the new module as follows:

public class ModuleExampleExplained {  
   public static void main(String[] args){  
       ModuleExample moduleExample = new ModuleExample();  
       moduleExample.displayMessage();  
   }  
}

# IoC and Dependency Injection

**We'll cover the following**

* [Inversion of Control (IoC)](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7Arrv4YDz7r#Inversion-of-Control-(IoC))
  + [What is Inversion of Control (IoC)?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7Arrv4YDz7r#What-is-Inversion-of-Control-(IoC)?)
  + [What are the advantages of Inversion of Control?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7Arrv4YDz7r#What-are-the-advantages-of-Inversion-of-Control?)
  + [What is the responsibility of an IoC container?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7Arrv4YDz7r#What-is-the-responsibility-of-an-IoC-container?)
  + [Describe the two types of IoC container.](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7Arrv4YDz7r#Describe-the-two-types-of-IoC-container.)
  + [Give an example of the BeanFactory implementation.](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7Arrv4YDz7r#Give-an-example-of-the-BeanFactory-implementation.)
  + [What is ApplicationContext?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7Arrv4YDz7r#What-is-ApplicationContext?)
  + [Give examples of the ApplicationContext implementations.](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7Arrv4YDz7r#Give-examples-of-the-ApplicationContext-implementations.)
  + [What is the difference between BeanFactory and ApplicationContext?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7Arrv4YDz7r#What-is-the-difference-between-BeanFactory-and-ApplicationContext?)
  + [How is ApplicationContext configured in Spring?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7Arrv4YDz7r#How-is-ApplicationContext-configured-in-Spring?)
  + [What is WebApplicationContext?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7Arrv4YDz7r#What-is-WebApplicationContext?)
  + [What happens if the context is not closed?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7Arrv4YDz7r#What-happens-if-the-context-is-not-closed?)
* [Dependency injection](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7Arrv4YDz7r#Dependency-injection)
  + [What is dependency injection?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7Arrv4YDz7r#What-is-dependency-injection?)
  + [How is dependency injection related to inversion of control?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7Arrv4YDz7r#How-is-dependency-injection-related-to-inversion-of-control?)
  + [What are the types of dependency injection?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7Arrv4YDz7r#What-are-the-types-of-dependency-injection?)
  + [What is constructor injection?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7Arrv4YDz7r#What-is-constructor-injection?)
  + [How does setter injection work?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7Arrv4YDz7r#How-does-setter-injection-work?)
  + [Explain setter injection for objects and literal values.](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7Arrv4YDz7r#Explain-setter-injection-for-objects-and-literal-values.)
  + [Explain injection of Java Collection types.](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7Arrv4YDz7r#Explain-injection-of-Java-Collection-types.)
  + [What is the difference between constructor and setter injection?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7Arrv4YDz7r#What-is-the-difference-between-constructor-and-setter-injection?)
  + [Which dependency injection approach is better?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7Arrv4YDz7r#Which-dependency-injection-approach-is-better?)
  + [What is method injection?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7Arrv4YDz7r#What-is-method-injection?)
  + [What is a circular dependency and how should it be resolved?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7Arrv4YDz7r#What-is-a-circular-dependency-and-how-should-it-be-resolved?)

## Inversion of Control (IoC)

### What is Inversion of Control (IoC)?

Suppose, class A is dependent on class B. In case of tight coupling, class A is responsible for creating an object of class B. In case of loose coupling, the framework takes the responsibility of creating and populating the dependency. The control of creating dependency shifts from the class to the framework. This is called Inversion of Control (IoC).

### What are the advantages of Inversion of Control?

* IoC minimizes the amount of code in the application since the framework is now responsible for creating and wiring the dependencies.
* Inversion of Control leads to loose coupling where it becomes easy to change implementations of an interface.
* It leads to better testability as mock objects can be created for the application.
* IoC supports eager instantiation as well as lazy loading.

### What is the responsibility of an IoC container?

An IoC container performs the following tasks:

* It instantiates the application class.
* It identifies the beans along with their dependencies and wires the dependencies.
* It manages the lifecycle of the beans from the time they are created till the time they are destroyed.

The IoC container uses the configuration metadata, in the form of an XML file or Java annotations, which contains instructions about the objects and their dependencies.

### Describe the two types of IoC container.

The two types of IoC container are **BeanFactory** and **ApplicationContext**. These are interfaces, with various implementations, that act as the IoC container.

**BeanFactory** provides basic functionality of an IoC container while **ApplicationContext** adds extra functionality like AOP, message resource handling for internationalization and **WebApplicationContext** for web applications.

Spring recommends using **ApplicationContext** unless the resources are limited, like for example on a mobile device or for applet based applications.

### Give an example of the **BeanFactory** implementation.

The most commonly used implementation of **BeanFactory** is the XmlBeanFactory class. This container reads the metadata from an XML config file to create a fully configured application.

### What is **ApplicationContext**?

**ApplicationContext** is a type of IoC container. It extends the **BeanFactory** interface.

Similar to the **BeanFactory**, the **ApplicationContext** can load bean definitions, wire beans together and return beans upon request. Additional features of **ApplicationContext** that are not part of the **BeanFactory** are support for AOP and internationalization, publishing events, and application layer specific contexts like **WebApplicationContext**.

### Give examples of the **ApplicationContext** implementations.

Commonly used implementations of **ApplicationContext** are:

* **ClassPathApplicationXmlContext**: reads the configuration from an XML file for standalone java applications.
* AnnotationConfigApplicationContext: uses annotation based configuration for standalone java applications.
* **AnnotationConfigWebApplicationContext** and **WebXmlApplicationContext**: for web applications.

### What is the difference between **BeanFactory** and **ApplicationContext**?

The default implementation of **BeanFactory** uses lazy initialization. It instantiates beans when the getBean() method is called. **ApplicationContext** extends the **BeanFactory** interface but the default implementation uses eager initialization. The beans are instantiated when the application starts. However, this behavior can be overridden.

**BeanFactory** does not support annotation based dependency injection. This feature was included in **ApplicationContext**.

### How is ApplicationContext configured in Spring?

There are multiple ways to configure application context:

* Application context can be configured using XML. The context can be created using **ClassPathXmlApplicationContext**, **FileSystemXmlApplicationContext**, or **GenericXmlApplicationContext** class which looks for XML file defining the configuration.
* Annotations can also be used to automatically register classes in the application context. The @Component annotation (along with @Controller, @Service and @Repository annotations) are used on classes and the **AnnotationConfigApplicationContext** class is used to create the application context.
* Java configuration using the @Configuration annotation on classes and @Bean annotation on methods is another way to configure the application context. The **AnnotationConfigApplicationContext** class is used to create the context by scanning the annotations.

### What is WebApplicationContext?

**WebApplicationContext** interface extends **ApplicationContext** interface. It provides configuration for web applications.

**WebApplicationContext** defines request, session and global application scopes in addition to the singleton and prototype scopes in **ApplicationContext**.

It has the ability to resolve themes and decide which servlet it is associated with.

### What happens if the context is not closed?

Not closing the context leads to resource leak. The close() method destroys all beans, releases the locks and closes the bean factory. Similarly, using a try-with-resources block also ensures that each resource is closed when the block exits.

## Dependency injection

### What is dependency injection?

Dependency injection is a concept which states that the developer should not create objects manually in the code but specify how the objects should be created. The IoC container reads this information and instantiates the object with the required dependencies.

Dependency injection is the process of finding a bean to be autowired. If class A has a dependency on class B, then the process of identifying the dependency, creating an instance of class B and autowiring the object of class B in class A is called dependency injection.

### How is dependency injection related to inversion of control?

Inversion of control (IoC) is a general concept which can be expressed in different ways. Dependency injection is an example of IoC.

IoC concept is that the control of creating and wiring the dependencies shifts from the developer to the framework. Dependency injection is the process by which the framework identifies the dependencies of an object, finds a match, and wires the dependency in the object.

### What are the types of dependency injection?

A dependency can be injected in several ways:

* Field injection
* Setter injection
* Constructor injection

### What is constructor injection?

In constructor injection, the IoC container creates the object by calling a constructor with a number of arguments where each argument represents a dependency on another class.

The following code example uses a constructor to inject the **Engine** dependency in **Vehicle** class.

@Component  
Class Vehicle {  
  
    private Engine engine;  
  
    Vehicle(Engine engine) {  
        this.engine = engine;  
    }  
    //...  
}

If the class contains more than one constructor, then the @Autowired annotation must be used on a constructor to tell the spring container that this constructor is to be used for dependency injection. For a class with one constructor, like the one shown above, the @Autowired annotation is optional.

### How does setter injection work?

Setter injection works by calling setter methods to set the property values. Spring container creates the object by calling the no-argument constructor and then calls setter methods to populate the dependencies.

The Engine dependency is injected using the setEngine() method. When Spring finds the @Autowired annotation, it will call the setter method for dependency injection.

@Component  
Class Vehicle {  
  
  private Engine engine;  
  
    @Autowired  
    void setEngine (Engine engine) {  
        this.engine = engine;  
    }  
  
    Engine getEngine ( ) {  
        return engine;  
    }  
    //...  
}

### Explain setter injection for objects and literal values.

In the XML file, the property tag is used to specify the properties of an object. The properties can be literal values or objects of other classes. In case of literal values, we use the value tag to set the value of the property and in case of objects we use the ref tag to provide the reference of the object that needs to be injected.

The following example shows the **Vehicle** class with two properties, a literal numWheels an object engine. The XML file defines the beans for both the **Engine** and **Vehicle** classes. In the **Vehicle** bean, the value tag sets the literal value while the ref tag is used to provide the reference of the **Engine** object.

<bean   
  id="engineType"   
  class="io.datajek.spring.di.Engine" />  
       
<bean   
  id="vehicle"   
  class="io.datajek.spring.di.Vehicle" >  
    <property name="numWheels" value="4" />  
    <property name="engine" ref="engineType" />  
</bean>

### Explain injection of Java Collection types.

To inject **java.util.Collection** types such as List, Set, Map and Properties, Spring provides the following tags: <list>, <set>, <map>, and <prop>.

The <list> tag allows duplicate values while the <set> tag does not allow duplicates. The <map> tag injects name-value pairs where name and value can be of any type while <prop> tag is used to inject name-value pairs of String type.

### What is the difference between constructor and setter injection?

* Constructor injection is not partial while setter injection offers partial dependency injection. If an object has 5 fields, it is not possible to pass just 1 in a 5 argument constructor.
* Constructor injection does not override setter, whereas setter injection overrides constructor if both are defined. The IoC container by default chooses setter injection.
* Constructor injection works well if the number of properties is large, whereas setter injection would make the code longer in such a scenario.
* Setter injection is flexible because it is possible to change the value of the property without creating a new bean instance. In case of constructor injection, a new bean is needed if a property is modified.

### Which dependency injection approach is better?

All dependency injection approaches have the same outcome. This is a very debatable question with some people favoring one style while others touting for another.

The documentation for older versions of Spring suggested that constructor injection be used for all mandatory dependencies while setter injection for optional dependencies. However the @Required annotation on a setter method can be used to make it a mandatory dependency.

### What is method injection?

Any method can be used for setting the dependency if the @Autowired annotation is used on it. This is referred to as method injection. The method can have any name. As long as it has the @Autowired annotation, Spring will find a matching dependency to inject.

### What is a circular dependency and how should it be resolved?

When beanA has a dependency on beanB and benB has a dependency on beanA, it results in a circular dependency. In this case both beans try to inject each other via constructor and Spring throws **BeanCurrentlyInCreationException**.

More than two beans can also result in a circular dependency as follows:

beanA => beanB => beanC => beanD => beanA

Spring creates beans in order in which they are needed. If a bean has a dependency, then the dependency is created first and then injected to complete the creation of the bean. In case of circular dependency, spring cannot decide which bean to create first.

Circular dependency issue arises when using constructor injection because the beans are created when the context is loaded. If using setter or field injection, the beans are created but their dependencies are injected only when they are needed. Thus the circular dependency issue can be avoided.

When using constructor injection, @Lazy annotation can be used. This tells spring that when initializing the bean, inject a proxy. The bean is fully created only when it is needed.

@Component  
public class ClassA {  
  
    private ClassB classB;  
  
    @Autowired  
    public ClassA(@Lazy ClassB classB) {  
        this.classB = classB;  
    }  
}

# Beans

**We'll cover the following**

* [Spring beans](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RLkpO0Jn9kE#Spring-beans)
  + [What are beans in Spring?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RLkpO0Jn9kE#What-are-beans-in-Spring?)
  + [What is the lifecycle of a Spring Bean?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RLkpO0Jn9kE#What-is-the-lifecycle-of-a-Spring-Bean?)
  + [What are custom bean lifecycle methods?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RLkpO0Jn9kE#What-are-custom-bean-lifecycle-methods?)
  + [What are some features of custom init and destroy methods?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RLkpO0Jn9kE#What-are-some-features-of-custom-init-and-destroy-methods?)
  + [What information does the bean definition contain?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RLkpO0Jn9kE#What-information-does-the-bean-definition-contain?)
  + [How can you provide a bean id when using annotations?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RLkpO0Jn9kE#How-can-you-provide-a-bean-id-when-using-annotations?)
  + [Are Spring beans same as JavaBeans?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RLkpO0Jn9kE#Are-Spring-beans-same-as-JavaBeans?)
  + [How are beans created?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RLkpO0Jn9kE#How-are-beans-created?)
  + [What does the @Bean annotation do?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RLkpO0Jn9kE#What-does-the-@Bean-annotation-do?)
  + [Both @Bean and @Component annotations create beans. What is the difference between the two?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RLkpO0Jn9kE#Both-@Bean-and-@Component-annotations-create-beans.-What-is-the-difference-between-the-two?)
  + [How can dependencies be injected using the @Bean annotation?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RLkpO0Jn9kE#How-can-dependencies-be-injected-using-the-@Bean-annotation?)
  + [What are the different scopes of a bean?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RLkpO0Jn9kE#What-are-the-different-scopes-of-a-bean?)
  + [What is the default bean scope in Spring?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RLkpO0Jn9kE#What-is-the-default-bean-scope-in-Spring?)
  + [What is the default scope in the web context?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RLkpO0Jn9kE#What-is-the-default-scope-in-the-web-context?)
  + [When are singleton and prototype scopes used?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RLkpO0Jn9kE#When-are-singleton-and-prototype-scopes-used?)
  + [How is bean scope defined?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RLkpO0Jn9kE#How-is-bean-scope-defined?)
  + [Is Singleton scope in Spring same as the Singleton design pattern?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RLkpO0Jn9kE#Is-Singleton-scope-in-Spring-same-as-the-Singleton-design-pattern?)
  + [Are Singleton beans thread safe?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RLkpO0Jn9kE#Are-Singleton-beans-thread-safe?)
  + [Explain prototype bean scope.](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RLkpO0Jn9kE#Explain-prototype-bean-scope.)
  + [Does Spring manage the complete lifecycle of beans?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RLkpO0Jn9kE#Does-Spring-manage-the-complete-lifecycle-of-beans?)
  + [What is an inner bean?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RLkpO0Jn9kE#What-is-an-inner-bean?)

## Spring beans

### What are beans in Spring?

Simply put, a Spring bean is a Java object. When Java objects are created by the Spring container, they are referred to as Spring Beans.

Beans are managed by the Spring container using the configuration metadata in the form of XML or Java annotations. The container instantiates, assembles and manages the lifecycle of a bean. For example, the @Component annotation on a class tells Spring framework that it has to manage the lifecycle of the objects of that class.

### What is the lifecycle of a Spring Bean?

Spring container instantiates a bean and initializes it. It also injects the required dependencies. When the context is destroyed, all the initialized beans are also destroyed.

Spring provides post initialization and pre destruction methods for custom tasks. These methods can be invoked using XML config file or Java annotations.

### What are custom bean lifecycle methods?

During the lifecycle of the bean, Spring allows the developer to add custom code during bean initialization and bean destruction. This can include code for custom business logic at initialization or destruction as well as setting up and cleaning up resources like a database connection or a file etc.

The XML tags init-method and destroy-method are used to define the custom methods inside the <bean> tag. The @PostContruct and @PreDestroy annotations also accomplish the same task.

### What are some features of custom init and destroy methods?

* Custom methods for initialization and destruction can have any name.
* They can have any access modifier; private, public, or protected.
* They can have any return type, but since the return value cannot be captured, void is mostly used as the return type.
* They cannot have any input arguments.

### What information does the bean definition contain?

Bean definition contains information for the container in the form of configuration metadata. Bean definition contains the following information:

* How the bean is created.
* Lifecycle details of the bean.
* Dependencies of the bean.

### How can you provide a bean id when using annotations?

We can specify the bean id in the component scan annotation.

@Component("bean1")  
public class MyBean {  
    ...  
}

When Spring registers **MyBean** class as a bean, it will store **bean1** as the bean id. The following code retrieves the bean:

MyBean theBean = context.getBean("bean1", MyBean.class);

If explicit bean id is not provided Spring generates a default bean id. The default bean id is the class name with the first letter lower-case.

@Component  
public class MyBean {  
    ...  
}

Now, when Spring registers **MyBean** class as a bean, it will store **myBean** as the bean id. To retrieve the bean from the container, we need to use the default bean id:

MyBean theBean = context.getBean("myBean", MyBean.class);

### Are Spring beans same as JavaBeans?

JavaBeans are Java classes that follow certain coding conventions. They have a public no-arg constructor, have private properties and allow access to properties using getter and setter methods and implement the java.io.Serializable interface.

Spring beans are objects whose lifecycle is managed by the Spring container. They do not follow the rigorous requirements of JavaBeans.

Spring beans are often the same as JavaBeans but they don’t have to be. Spring beans can have constructor with arguments and may not implement the java.io.Serializable interface.

### How are beans created?

Spring framework creates beans in an order. When Spring framework encounters a bean definition in XML file or through an annotation, it checks if the class is dependent on any other class. Suppose class A has a dependency on class B. In this case, Spring will create the object of class B. Once the dependency has been created, the bean for class A can be created by injecting the bean of class B in class A.

### What does the @Bean annotation do?

The @Bean annotation is used on a method to indicate that the method returns a bean to be managed by Spring. The name of the method indicates the bean id. This annotation is used in classes marked with @Configuration annotation.

The @Bean annotation provides the same functionality as the <bean> tag in XML configuration.

The following configuration class creates the a bean of **Vehicle** type using the @Bean annotation. The bean id is **vehicle**.

@Configuration  
class VehicleConfig {  
  
    @Bean  
    public Vehicle vehicle() {  
        return new Vehicle();  
    }  
  
}

To retrieve the bean, the bean id is used in the getBean method as follows:

Vehicle myVehicle = context.getBean("vehicle", Vehicle.class);

The @Bean annotation can also specify the initMethod and destroyMethod for the bean.

### Both @Bean and @Component annotations create beans. What is the difference between the two?

Some differences between the two annotations are:

* @Component enables Spring to auto-detect and auto-configure beans while @Bean is used to explicitly declare a bean rather than letting Spring auto-detect it.
* @Component is a class level annotation while @Bean is a method level annotation.
* Since @Component annotation is used on a class, it keeps the bean definition and class declarations together while @Bean decouples them.

### How can dependencies be injected using the @Bean annotation?

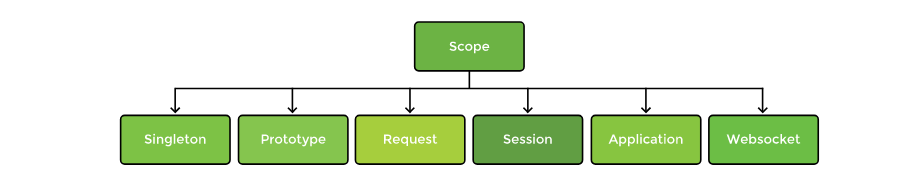
Consider a case where there are two classes, **Vehicle** and **Engine**. **Engine** is a dependency of **Vehicle** class. The @Bean annotation can be used to define the beans and inject the dependencies as follows:

@Configuration  
public class VehicleConfig{  
    //create Engine bean  
    @Bean  
    public Engine myEngine(){  
        return new Engine();  
    }  
    //create Vehicle bean and inject Engine bean in it  
    @Bean  
    public Vehicle myVehicle(){  
        return new Vehicle(myEngine());  
    }  
}

Here, the dependency is injected by calling myEngine() when creating the vehicle bean.

### What are the different scopes of a bean?

In Spring 5, six scopes have been defined. They are:

* Singleton. Only one instance of the bean per IoC container is created.
* Prototype. A new instance of the bean is created every time.
* Session. One bean is created for every Http session.
* Request. One bean is created per Http request.
* Application. One bean is created per ServletContext.
* Websocket. One bean is created per WebSocket.
* 

The singleton and prototype scopes can be used in any application while the last four scopes are only available for a web application.

### What is the default bean scope in Spring?

A Spring bean is initialized as a singleton bean by default.

### What is the default scope in the web context?

The default bean scope in Spring is Singleton and this applies to web application context as well.

### When are singleton and prototype scopes used?

Singleton scope is used for beans which are stateless whereas prototype scope is for beans in which we need to maintain a state.

The IoC container handles both types differently. It manages the entire lifecycle of singleton beans but does not do so in the case of prototype beans. The developer is responsible for destroying the prototype beans.

### How is bean scope defined?

The @Scope annotation is used to set the scope of a bean. In an XML file, the scope attribute can be used as shown below:

@Component  
@Scope("prototype")  
public class ShoppingCart {  
  
}

<bean id="cart" class="io.datajek.spring.beanscope.ShoppingCart" scope="prototype" />

### Is Singleton scope in Spring same as the Singleton design pattern?

According to the Singleton design pattern, there can be one bean per Java class. In Spring, singleton scope means one bean per bean id per Spring container (ApplicationContext).

The following XML code creates two beans of the same class:

<bean id="vehicle1" class="io.datajek.spring.Vehicle" scope="singleton">  
    <property name="numWheels" value="4"/>      
</bean>      
<bean id="vehicle2" class="io.datajek.spring.Vehicle" scope="singleton">  
    <property name="numWheels" value="6"/>      
</bean>

Both beans have a different bean id so two Singletons will be created by the Spring container.

### Are Singleton beans thread safe?

No, Singleton beans are not thread safe. Thread safety depends upon the implementation of the bean. Singleton is a design pattern that focuses on how beans are created, not how they are executed.

A singleton bean means there is one instance of the bean in the application context. All threads access the same class variables which can lead to inconsistency. Prototype scope on the other hand is thread safe but it is at the expense of performance because now there are multiple objects instead of one.

### Explain prototype bean scope.

Prototype scope means that every time the developer asks for an instance of the bean, the Spring container will create a new instance and return it. This is different from the singleton scope, where only one instance of the bean is created and the Spring container returns the reference of the same instance whenever it receives a request for the bean.

### Does Spring manage the complete lifecycle of beans?

Spring manages the lifecycle of beans from construction to destruction. However, this statement is not true in the case of prototype beans. The Spring container instantiates, configures and assembles a prototype object. But it keeps no track of the object after it is handed over to the client.

For a prototype bean, the initialization method is called but the destruction method is not called and the client is responsible for destroying the bean and releasing the resources that it acquired.

### What is an inner bean?

A bean which exists within the scope of another bean is called inner bean.

When a bean is used as a property of another bean, it becomes an inner bean. In this case the <bean> tag is used inside the <property> or <constructor-arg> tag.

The container ignores the id or name tags for an inner bean. Inner beans are always anonymous and have prototype scope.

For example, we can declare a bean of **Engine** type as an inner bean inside the **Vehicle** bean. The following code shows how to set the properties of the outer and inner bean:

<bean id="vehicle" class="io.datajek.spring.beanscope.Vehicle">  
    <property name="numWheels" value ="4" >  
    <property name="engine">  
        <bean class="io.datajek.spring.beanscope.Engine">  
            <property name="type" value="Combustion   Engine"></property>  
            <property name="rpm" value="4000"></property>  
        </bean>  
    </property>  
</bean>

The inner bean is not accessible outside the scope of the outer bean.

Back

# Spring Annotations

**We'll cover the following**

* [Spring annotations interview questions](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B1WwWk0pw5N#Spring-annotations-interview-questions)
  + [What is the purpose of @Component annotation?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B1WwWk0pw5N#What-is-the-purpose-of-@Component-annotation?)
  + [What is the difference between @Component, @Service, @Repository, and @Controller?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B1WwWk0pw5N#What-is-the-difference-between-@Component,-@Service,-@Repository,-and-@Controller?)
  + [Why is @Primary annotation used?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B1WwWk0pw5N#Why-is-@Primary-annotation-used?)
  + [Why is @Qualifier annotation used?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B1WwWk0pw5N#Why-is-@Qualifier-annotation-used?)
  + [Which annotations takes precedence: @Primary or @Qualifier?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B1WwWk0pw5N#Which-annotations-takes-precedence:-@Primary-or-@Qualifier?)
  + [Why is the @Required annotation used?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B1WwWk0pw5N#Why-is-the-@Required-annotation-used?)
  + [What is the purpose of @Autowired annotation?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B1WwWk0pw5N#What-is-the-purpose-of-@Autowired-annotation?)
  + [Both @Bean and @Component annotations create beans. What is the difference between the two?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B1WwWk0pw5N#Both-@Bean-and-@Component-annotations-create-beans.-What-is-the-difference-between-the-two?)
  + [What is the difference between @Inject and @Autowired in Spring? Which one to use under which condition?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B1WwWk0pw5N#What-is-the-difference-between-@Inject-and-@Autowired-in-Spring?-Which-one-to-use-under-which-condition?)

## Spring annotations interview questions

### What is the purpose of @Component annotation?

@Component annotation is used to define a bean. This annotation is placed on a Java class. The xml file contains the <context: component-scan> tag to enable component scanning. Spring container scans the package (and its subpackages) specified in the component-scan tag. It will register all classes marked with the @Component annotation as beans.

Using the @Component annotation can save a lot of time spent in writing lengthy bean definition code in XML.

### What is the difference between @Component, @Service, @Repository, and @Controller?

A typical application is divided into layers. The @Component annotation is generic and denotes any Spring managed bean. It can be used in any layer of the application whereas the other three are specific to layers. @Controller is used in the web layer, @Service is used on classes in the business layer and @Repository is used in the data layer and provides the additional functionality of making unchecked exceptions translated as Spring DataAccessException.

### Why is @Primary annotation used?

When two beans of the same type qualify to be autowired, then @Primary annotation is used to break the tie. Suppose there are two implementations of the Engine interface, CombustionEngine and ElectricEngine. The class Vehicle has a dependency on Engine interface. If both implementations have the @Component annotation then a compiler error will occur stating that more than one bean of the same type was found. In this case, we can guide Spring to use either CombustionEngine or ElectricEngine as the primary choice by using the @Primary annotation.

### Why is @Qualifier annotation used?

If more than one bean of the same type exists, we can choose the bean to be autowired using the @Qualifier annotation. The bean having this annotation qualifies to be autowired.

### Which annotations takes precedence: @Primary or @Qualifier?

When both the @Primary and @Qualifer annotations are present, then @Qualifier takes precedence. @Primary defines a default value and bean marked with this annotation will be used unless otherwise indicated. @Qualifier annotation is specific and is used when a particular bean is needed.

Suppose there are two beans of the same type and one is used in 90% of the cases, then it makes sense to make it the default choice by using @Primary annotation. The @Autowired that needs the other bean can use the @Qualifier annotation while all other @Autowired will automatically choose the bean marked with @Primary.

### Why is the @Required annotation used?

@Required is a method level annotation. It is used on setter methods and makes setter injection of the property mandatory. The BeanInitializationException is thrown if the property value is not initialized. If a setter method has @Autowired annotation on it, then @Required is not needed.

This annotation has been deprecated because constructor injection is used for setting all mandatory dependencies.

### What is the purpose of @Autowired annotation?

@Autowired annotation specifies where and how autowiring is done. This annotation can be used on setter methods, with a constructor argument, on a property as well as on methods with multiple arguments. By default autowiring is done by type.

### Both @Bean and @Component annotations create beans. What is the difference between the two?

Some differences between the two annotations are:

* @Component enables Spring to auto-detect and auto-configure beans while @Bean is used to explicitly declare a bean rather than letting Spring auto-detect it.
* @Component is a class level annotation while @Bean is a method level annotation.
* Since @Component annotation is used on a class, it keeps the bean definition and class declarations together while @Bean decouples them.

### What is the difference between @Inject and @Autowired in Spring? Which one to use under which condition?

Both these annotations perform the same function and are used for dependency injection by type. The order of dependency injection of both the annotations is as follows:

1. By type
2. Using @Qualifer annotation
3. By name.

The only difference between both annotations is that @Inject is a CDI annotation which makes it framework independent and @Autowired is a Spring framework annotation. Thus using @Inject may be helpful if the application is moved to another framework.

# Spring Configuration and Component Scan

**We'll cover the following**

* [Spring configuration](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVA0Dx2RAj2#Spring-configuration)
  + [What is a configuration file?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVA0Dx2RAj2#What-is-a-configuration-file?)
  + [How is configuration metadata provided to Spring container/ how are beans configured in Spring container?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVA0Dx2RAj2#How-is-configuration-metadata-provided-to-Spring-container/-how-are-beans-configured-in-Spring-container?)
  + [What is the difference between XML and Annotation configuration?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVA0Dx2RAj2#What-is-the-difference-between-XML-and-Annotation-configuration?)
  + [How is annotation based configuration enabled in Spring?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVA0Dx2RAj2#How-is-annotation-based-configuration-enabled-in-Spring?)
  + [Can there be multiple configuration files in a project?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVA0Dx2RAj2#Can-there-be-multiple-configuration-files-in-a-project?)
* [Component scan](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVA0Dx2RAj2#Component-scan)
  + [What is component scan?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVA0Dx2RAj2#What-is-component-scan?)
  + [How is component scan done in Spring Boot?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVA0Dx2RAj2#How-is-component-scan-done-in-Spring-Boot?)
  + [What is the difference between context: annotation-config and context:component-scan tags?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVA0Dx2RAj2#What-is-the-difference-between-context:-annotation-config-and-context:component-scan-tags?)

## Spring configuration

### What is a configuration file?

A configuration file is an XML file that describes the different classes in the application and how they are linked to one another. XML configuration files can get lengthy and difficult to manage in large projects. Annotation based approach is easier and cleaner.

### How is configuration metadata provided to Spring container/ how are beans configured in Spring container?

Configuration metadata can be provide in the following ways:

* **XML based configuration**: In XML config files, beans are defined using the <bean> tag containing a number of different configuration options. For example, the following XML configuration defines two beans and specifies constructor injection to inject **myDependency** bean in **myBean**.

<bean name="myDependency"  
     
class="io.datajek.spring.MyDependency">  
</bean>  
  
<bean name="myBean"   
   class="io.datajek.spring.MyBean">  
 <constructor-arg ref="myDependency"/>  
</bean>

* **Annotation based configuration**: In this approach, annotations are used. The @Component as well as some other annotations can be used to define beans. This approach makes the XML file smaller. The <context:component-scan/> tag defines the package where Spring searches for classes with the @Component annotation and automatically registers beans.<

<context:component-scan base-package "io.datajek.spring"/>

**MyBean** and **MyDependency** classes will be registered as beans if they are annotated with @Component annotation.

* **Java based configuration**: This approach eliminates XML altogether. The @Configuration annotation on a class defines it as a configuration class. Beans can be manually defined using the @Bean annotation or the @ComponentScan annotation can be used to automatically detect beans defined using annotations. For example:

@Configuration  
@ComponentScan("io.datajek.spring")  
public class MyConfig {  
   
}

All classes marked with @Component will be registered as beans after the component scan.

### What is the difference between XML and Annotation configuration?

Traditionally, XML was used to set up the configuration. Annotations make the same task easier and understandable. The result of both approaches is the same but less amount of code is written when using annotations.

### How is annotation based configuration enabled in Spring?

By default annotation based configuration is disabled. To enable annotation based configuration in XML file, the <context:annotation-config/> tag is used.

### Can there be multiple configuration files in a project?

Large projects can have multiple configuration files. They can be loaded using the @Import annotation and mentioning the list of files after the @Configuration annotation.

@Configuration  
@Import({VehicleConfig.class, EngineConfig.class})  
public class ApplicationConfig {

Multiple config files can also be imported in a single XML file using the <import> tag.

<import resource="vehicle.xml"/>  
<import resource="engine.xml"/>

## Component scan

### What is component scan?

Component scan is the process in which Spring searches for beans. Spring needs to know the location (package) to search for beans. Component scan can be defined in two ways:

* Using the @Component annotation and its sub-classes in which case Spring searches all the packages and sub-packages containing these annotation.
* Using XML configuration with <context:component-scan> tag specifying the base package.

### How is component scan done in Spring Boot?

In Spring Boot, the @SpringBootApplication annotation triggers a component scan on the package and its sub-packages where it is used.

### What is the difference between context: annotation-config and context:component-scan tags?

<context: annotation-config> enables the detection of dependency injection annotations like @Autowired, @Qualifier, @PostConstruct, and @PreDestroy.

<context: component-scan> specifies the base package for component scan. Component scan is used for finding beans. This tag can recognize the dependency injection annotations mentioned above as well as bean annotations like @Component, @Repository, and @Service etc.

# Autowiring and Misc Questions

**We'll cover the following**

* [Auto-wiring](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B8WmzmgWoRJ#Auto-wiring)
  + [What is autowiring in Spring?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B8WmzmgWoRJ#What-is-autowiring-in-Spring?)
  + [What are the different modes of autowiring in Spring?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B8WmzmgWoRJ#What-are-the-different-modes-of-autowiring-in-Spring?)
  + [How does autowiring internally work?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B8WmzmgWoRJ#How-does-autowiring-internally-work?)
  + [What is autowiring by constructor?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B8WmzmgWoRJ#What-is-autowiring-by-constructor?)
  + [What are the limitations of autowiring?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B8WmzmgWoRJ#What-are-the-limitations-of-autowiring?)
  + [Is it possible to exclude a bean from being autowired?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B8WmzmgWoRJ#Is-it-possible-to-exclude-a-bean-from-being-autowired?)
  + [What does the @Autowired annotation do?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B8WmzmgWoRJ#What-does-the-@Autowired-annotation-do?)
* [Miscellaneous questions about Spring](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B8WmzmgWoRJ#Miscellaneous-questions-about-Spring)
  + [What happens if we specify an interface instead of a class in getBean() method?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B8WmzmgWoRJ#What-happens-if-we-specify-an-interface-instead-of-a-class-in-getBean()-method?)
  + [Why do we need a no-arg constructor?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B8WmzmgWoRJ#Why-do-we-need-a-no-arg-constructor?)
  + [What is Spring Security?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/B8WmzmgWoRJ#What-is-Spring-Security?)

## Auto-wiring

### What is autowiring in Spring?

Connecting beans together in the Spring container is called autowiring. It is the process by which collaborating beans are tied together without the developer having to write explicit object instantiation code.

It reduces the code as well as the development time because it removes the need to write dependency injection code.

### What are the different modes of autowiring in Spring?

When using XML configuration the autowiring mode can be specified using the autowire attribute in the <bean> tag. The modes of bean autowiring are:

1. no: the default autowiring mode is no autowiring in which case the developer has to provide explicit bean reference using the ref attribute.
2. byName: the bean is injected by matching the property, that needs to be autowired, with a bean that has the same name. The property name must match a bean name for this type of autowiring to work.
3. byType: the bean is injected by matching the property, that needs to be autowired, with a bean of the same type. If no matches are found, the property is not set. If more than one matches are found an error occurs.
4. constructor: the dependency is injected by calling the constructor with a bean whose type matches with the constructor argument. If no matches are found, an error occurs.

### How does autowiring internally work?

Spring calls the setter method for the property when autowiring mode is byName and byType.

When autowiring mode is constructor and the class defines multiple constructors, then Spring will call the constructor with the most number of parameters.

### What is autowiring by constructor?

Autowiring by constructor is similar to autowiring by type. It is applied to a constructor argument. Consider the example of a **Vehicle** class which has a dependency on the **Engine** class. The following code injects the dependency explicitly without using autowiring:

<bean id="vehicle" class="io.datajek.spring.Vehicle">  
    <constructor-arg>  
        <ref bean="engine" />  
    </constructor-arg>  
</bean>  
          
<bean id="engine" class="io.datajek.spring.Engine" />

Autowiring by constructor can be used as follows:

<bean id="vehicle" class="io.datajek.spring.Vehicle" autowire="constructor" />  
         
<bean id="engine" class="io.datajek.spring.Engine" />

In this case, the constructor of the **Vehicle** class is called to inject the **Engine** object.

### What are the limitations of autowiring?

* Autowiring cannot be used for primitive data types like String and int. It can only be used for object references.
* Autowiring is not explicit, so it may lead to unexpected behavior. It is better to do explicit wiring of bean dependencies.
* It is possible to override autowiring if the dependencies are specified using <constructor-arg> and <property> tags.
* Autowiring is lookup based and can be expensive in large applications.

### Is it possible to exclude a bean from being autowired?

When using XML configuration, the autowire-candidate attribute can be used in the <bean> tag to exclude that bean from becoming a candidate for autowiring. The Spring container makes the bean definition unavailable when autowiring takes place.

However, if autowiring mode is set to byName then the above solution will not work.

### What does the @Autowired annotation do?

The @Autowired annotation can be used to guide Spring framework how autowiring is to be done. This annotation can be used on fields, setter methods or with constructor arguments.

By default, it uses dependency injection by type. This annotation is often used with @Qualifier to remove the ambiguity when more than one beans of the matching type are found.

## Miscellaneous questions about Spring

### What happens if we specify an interface instead of a class in getBean() method?

We can pass an interface to a method and behind the scene, Spring casts the object. It behaves in the same way as getBean(String). Type safety is ensured when casting and BeanNotOfRequiredTypeException is thrown if bean of the same type is not found.

ClassCastException cannot be thrown on casting the result correctly as can happen with getBean(String) method.

### Why do we need a no-arg constructor?

When we do not define a constructor for our class, the compiler defines a default one for us. But when we declare a parameterized constructor, the compiler does not create the default one. If we are creating an object without passing any arguments, it will need the default no-arg constructor. If it does not exist in the code, we need to explicitly define a no-arg constructor.

### What is Spring Security?

It is a Spring module that provides authentication and authorization functionality to Spring MVC applications. It also provides the PasswordEncoder interface to secure user passwords. Spring Security takes care of common security vulnerabilities.

Spring security intercepts a user request and checks if the user is authorized to access the protected resources. It reads the application’s security configuration and also looks at the user’s passwords and roles to see if the user is authenticated and authorized to access the web resource.

# Spring Boot

**We'll cover the following**

* [Spring Boot interview questions](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMDOXB09EEK#Spring-Boot-interview-questions)
  + [What is Spring Boot?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMDOXB09EEK#What-is-Spring-Boot?)
  + [Why do we need Spring Boot?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMDOXB09EEK#Why-do-we-need-Spring-Boot?)
  + [How can a Spring Boot application be created?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMDOXB09EEK#How-can-a-Spring-Boot-application-be-created?)
  + [What is Spring Initializr?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMDOXB09EEK#What-is-Spring-Initializr?)
  + [What are the components of Spring Boot?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMDOXB09EEK#What-are-the-components-of-Spring-Boot?)
  + [What is the purpose of @SpringBootApplication annotation?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMDOXB09EEK#What-is-the-purpose-of-@SpringBootApplication-annotation?)
  + [What is the purpose of @RestController annotation in Spring Boot?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMDOXB09EEK#What-is-the-purpose-of-@RestController-annotation-in-Spring-Boot?)
  + [What is the purpose of @ConditionalOnMissingBean annotation?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMDOXB09EEK#What-is-the-purpose-of-@ConditionalOnMissingBean-annotation?)
  + [What is an embedded server in Spring Boot?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMDOXB09EEK#What-is-an-embedded-server-in-Spring-Boot?)
  + [Can we change the default embedded server to Jetty?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMDOXB09EEK#Can-we-change-the-default-embedded-server-to-Jetty?)
  + [How are Spring Boot web applications deployed?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMDOXB09EEK#How-are-Spring-Boot-web-applications-deployed?)
  + [What is Spring Boot Dependency Management?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMDOXB09EEK#What-is-Spring-Boot-Dependency-Management?)
  + [What happens when a web application is run using Spring Boot?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMDOXB09EEK#What-happens-when-a-web-application-is-run-using-Spring-Boot?)
  + [How can the default port of the embedded tomcat server be changed?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMDOXB09EEK#How-can-the-default-port-of-the-embedded-tomcat-server-be-changed?)
  + [How can the default web server in a Spring Boot application be disabled?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMDOXB09EEK#How-can-the-default-web-server-in-a-Spring-Boot-application-be-disabled?)
  + [What are Spring Boot starter projects?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMDOXB09EEK#What-are-Spring-Boot-starter-projects?)
  + [What is spring-boot-starter-parent?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMDOXB09EEK#What-is-spring-boot-starter-parent?)
  + [Can Spring Boot be used for applications not built using the Spring framework?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMDOXB09EEK#Can-Spring-Boot-be-used-for-applications-not-built-using-the-Spring-framework?)
  + [How can we connect Spring Boot with databases?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMDOXB09EEK#How-can-we-connect-Spring-Boot-with-databases?)
  + [What does the exclude attribute of the @SpringBootApplication do?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMDOXB09EEK#What-does-the-exclude-attribute-of-the-@SpringBootApplication-do?)
  + [How can external configuration be done in Spring Boot?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMDOXB09EEK#How-can-external-configuration-be-done-in-Spring-Boot?)
  + [What is Spring Actuator?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMDOXB09EEK#What-is-Spring-Actuator?)
  + [How can the Actuator be accessed in Spring Boot?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMDOXB09EEK#How-can-the-Actuator-be-accessed-in-Spring-Boot?)
  + [How can you create custom endpoints in Spring Boot Actuator?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMDOXB09EEK#How-can-you-create-custom-endpoints-in-Spring-Boot-Actuator?)
  + [Explain Spring Boot DevTools.](https://www.educative.io/courses/guide-spring-5-spring-boot-2/RMDOXB09EEK#Explain-Spring-Boot-DevTools.)

## Spring Boot interview questions

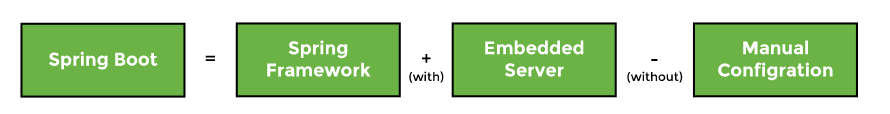
### What is Spring Boot?

Spring Boot is a project that contains a pre configured set of dependencies for different projects which aims at reducing the time for application configuration. It allows a developer to set up a running Spring application with a small amount of code.

### Why do we need Spring Boot?

Spring Boot minimizes boilerplate code. It has a predefined set of dependencies and their versions which work together. Spring Boot autoconfigures the dependencies making the developers task easier. It helps kick start the development process.

Since it comes with an embedded server, there is no need to deploy WAR files.



### How can a Spring Boot application be created?

There are various ways on which a Spring Boot application can be created.

* Using Spring Initializr
* Spring Boot CLI
* Spring Tools Suite (STS)

### What is Spring Initializr?

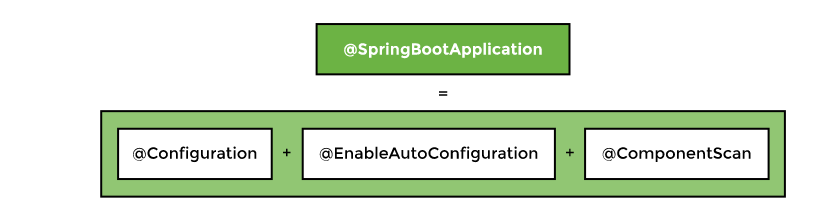
Spring Initializr is a web tool on the official Spring site. It asks for the project details and then configures all dependencies based on the project requirements to create a working project that is available for downloaded.

### What are the components of Spring Boot?

The core components of Spring Boot are:

* **Spring Boot Starters**: a set of pre configured dependencies that help kick start a project.
* **Spring Boot AutoConfiguration**: provide configuration for functionality common to many Spring applications.
* **Spring Boot CLI**: allows running and testing a Spring Boot application from the command prompt.
* **Spring Boot Actuator**: provides metrics about application’s performance.

### What is the purpose of @SpringBootApplication annotation?



The @SprignBootApplication is used to launch the application context. It is a conglomeration of three annotations.

This annotation is used on the main class of a Spring Boot application.

### What is the purpose of @RestController annotation in Spring Boot?

The @RestController annotation combines the @Controller and @ResponseBody annotation. This annotation eliminates the need to annotate methods that map requests with the @ResponseBody annotation thereby making the code cleaner. The @ResponseBody annotation serializes the return object into HttpResponse.

### What is the purpose of @ConditionalOnMissingBean annotation?

This annotation is used with the @Bean annotation and tells the auto configuration class to initialize a bean only when it cannot find it in the application context. If the bean of that type already exists, then it will not be created.

@ConditionalOnMissingBean  
@Bean  
public MyBean myBean() {  
    return new MyBean();  
}

**myBean** will be initialized only when no other bean of type **MyBean** exists in the context. If it does, then **myBean** will not be registered as a bean.

The main purpose of this annotation is to provide a fallback bean, in case no bean of that type is present.

### What is an embedded server in Spring Boot?

Embedded server means that the server is included in the application JAR and does not need to be deployed separately. Tomcat is the embedded server in SpringBoot applications. Embedded server makes the deployment of web applications easy and independent.

### Can we change the default embedded server to Jetty?

Tomcat is the default web server that comes with spring-boot-starter-web. To change the web server to Jetty, we will exclude tomcat dependency and add jetty dependency.

<dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-web</artifactId>  
    <exclusions>  
        <exclusion>  
            <groupId>org.springframework.boot</groupId>  
            <artifactId>spring-boot-starter-tomcat</artifactId>  
        </exclusion>  
    </exclusions>  
</dependency>  
  
<dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-jetty</artifactId>  
</dependency>

### How are Spring Boot web applications deployed?

Spring Boot web applications can be deployed as Jar or War files. Traditionally web applications were deployed as War files so as to save memory on the server. However now large size Jar files can also be deployed since hardware is cheap.

Spring provides a plugin to package a web application. This can be included in the pom file as follows:

<plugin>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-maven-plugin</artifactId>  
</plugin>

The <packaging> attribute can be used to change the packaging to War. The default value is Jar.

We will get a deployable jar or war file after the Maven Package phase is executed.

### What is Spring Boot Dependency Management?

Spring Boot has a list of dependencies that it supports. These include all Spring modules that can be used with Spring Boot as well as some third party libraries. The developer need not mention the version numbers of the dependencies as they are automatically configured by Spring Boot.

### What happens when a web application is run using Spring Boot?

The Tomcat server is automatically launched when a Spring Boot web application is run as a Java Application.

### How can the default port of the embedded tomcat server be changed?

The server.port property in the application.properties file can be used to assign a different port to Tomcat server.

### How can the default web server in a Spring Boot application be disabled?

The spring.main.web-application-type=none property in the application.properties file can be used to disable the default web server and change the web application type.

### What are Spring Boot starter projects?

Spring Boot offers starter projects to kick start an application. Starter projects include all the relevant dependencies needed for the project. They aim at saving the time spent finding dependencies and their version which work together. Examples of starter projects include: web starter containing Spring MVC, Tomcat, Jackson, etc., test starter containing JUnit, Mockito, Hamcrest, etc., Data JPA Starter containing in-memory databases, AOP starter, and actuator starter etc.

Only one dependency is needed in the pom file for the starter, for example, spring-boot-starter-web that contains all the required dependencies for a web application.

### What is spring-boot-starter-parent?

By using spring-boot-starter-parent in our project, we can reuse the default settings of Spring Boot. spring-boot-starter-parent defines the default java version, default encoding as well as plugin configuration for different plugins like maven-failsafe-plugin, maven-jar-plugin etc. It also specifies the working combination of dependencies and their version numbers. Everything in the parent plugin is inherited by the child pom. The starter parent dependency can be included as follows:

<parent>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-parent</artifactId>  
    <version>2.4.0.RELEASE</version>  
</parent>

The latest version can be found from the Maven Central Repository.

### Can Spring Boot be used for applications not built using the Spring framework?

No, Spring Boot is a module of Spring and can only be used for applications using the Spring framework.

### How can we connect Spring Boot with databases?

The spring-boot-starter-data-jpa is used to connect a Spring Boot application to relational databases using JPA. When using JDBC, Spring Boot provides classes like JdbcTemplate, NamedParameterJdbcTemplate and DataSource that can be used to efficiently connect to, and perform database operations.

### What does the exclude attribute of the @SpringBootApplication do?

The exclude attribute, when used with the @SpringBootApplication annotation is used to exclude a package from component scanning. This attribute is used without the basePackages filter.

### How can external configuration be done in Spring Boot?

To enable using different configuration in different environments, the following options can be used in Spring Boot:

* **Command line properties**: the application can be run using command line arguments which are converted to properties.
* **Application properties**:an application properties file can be placed in the current directory of the application, classpath or config directory. Spring Boot reads this file to load external configuration.
* **Profile specific properties**: The application-{profile}.properties file resides in the same location as the above mentioned properties file. Spring Boot reads the file corresponding to the active profile using the {profile} placeholder.

### What is Spring Actuator?

Spring Actuator provides data about the application that can be used for performance analysis as well as debugging. The data provided by Actuator includes, among many other things, the health of the application, beans created, controller mappings, and memory used etc. The Actuator endpoints are secured using Spring Security.

### How can the Actuator be accessed in Spring Boot?

To access Spring Actuator, we need to add the spring-boot-starter-actuator dependency in **pom.xml**. The Maven dependency is shown as follows:

<dependency>  
<groupId>org.springframework.boot</groupId>  
<artifactId>spring-boot-starter-actuator</artifactId>  
</dependency>

### How can you create custom endpoints in Spring Boot Actuator?

The @Endpoint annotation is used to create custom endpoints. To create a new endpoint, we can create a class and annotate it with the @Controller annotation along with the @Endpoint annotation.

### Explain Spring Boot DevTools.

Spring Boot DevTools increase the productivity by decreasing the time needed for any change to be updated on the server. When a change is made to the code, it needs to be redeployed on the server and the server needs to be restarted. This wastes a lot of time which could otherwise be spent on development. Spring Boot DevTools reloads the change without the need to restart the server.

# Spring Data JDBC and JPA

**We'll cover the following**

* [Spring Data JDBC](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVMxn3E158D#Spring-Data-JDBC)
  + [How does Spring provide DAO support?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVMxn3E158D#How-does-Spring-provide-DAO-support?)
  + [Describe Spring DAO support classes.](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVMxn3E158D#Describe-Spring-DAO-support-classes.)
  + [What annotation is used to mark DAO in Spring?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVMxn3E158D#What-annotation-is-used-to-mark-DAO-in-Spring?)
  + [What is Spring Data JDBC?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVMxn3E158D#What-is-Spring-Data-JDBC?)
  + [What tasks are performed by Spring JDBC?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVMxn3E158D#What-tasks-are-performed-by-Spring-JDBC?)
  + [What is the difference between JDBC and Spring JDBC?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVMxn3E158D#What-is-the-difference-between-JDBC-and-Spring-JDBC?)
  + [Name the classes in Spring JDBC API?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVMxn3E158D#Name-the-classes-in-Spring-JDBC-API?)
  + [What are the advantages of JdbcTemplate in Spring?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVMxn3E158D#What-are-the-advantages-of-JdbcTemplate-in-Spring?)
  + [Why is NamedParameterJdbcTemplate class used?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVMxn3E158D#Why-is-NamedParameterJdbcTemplate-class-used?)
  + [How are records from a database fetched when using JdbcTemplate?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVMxn3E158D#How-are-records-from-a-database-fetched-when-using-JdbcTemplate?)
  + [What is the difference between a RowMapper and ResultSetExtractor?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVMxn3E158D#What-is-the-difference-between-a-RowMapper-and-ResultSetExtractor?)
  + [Explain Spring’s exception handling support with DataAccessException.](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVMxn3E158D#Explain-Spring%E2%80%99s-exception-handling-support-with-DataAccessException.)
  + [What is SQLExceptionTranslator?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVMxn3E158D#What-is-SQLExceptionTranslator?)
  + [What are JdbcTemplate callback interfaces?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVMxn3E158D#What-are-JdbcTemplate-callback-interfaces?)
* [Spring Data JPA](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVMxn3E158D#Spring-Data-JPA)
  + [What is Spring Data?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVMxn3E158D#What-is-Spring-Data?)
  + [What is JPA?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVMxn3E158D#What-is-JPA?)
  + [What is the difference between JPA and Spring Data JPA?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVMxn3E158D#What-is-the-difference-between-JPA-and-Spring-Data-JPA?)
  + [What is the difference between CrudRepository and JpaRepository?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVMxn3E158D#What-is-the-difference-between-CrudRepository-and-JpaRepository?)
  + [Which No SQL databases does Spring support?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVMxn3E158D#Which-No-SQL-databases-does-Spring-support?)
  + [Which ORMs are supported by Spring?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVMxn3E158D#Which-ORMs-are-supported-by-Spring?)
  + [What is Hibernate?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVMxn3E158D#What-is-Hibernate?)
  + [What are the different types of transaction management supported by Spring?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVMxn3E158D#What-are-the-different-types-of-transaction-management-supported-by-Spring?)
  + [Which transaction management type is preferred in Spring?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVMxn3E158D#Which-transaction-management-type-is-preferred-in-Spring?)
  + [What are some benefits of using Spring Transactions?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVMxn3E158D#What-are-some-benefits-of-using-Spring-Transactions?)
  + [What happens when the @Transactional annotation is used on a method?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVMxn3E158D#What-happens-when-the--@Transactional-annotation-is-used-on-a-method?)
  + [What is the difference between JdbcTemplate and JPA?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/qVMxn3E158D#What-is-the-difference-between-JdbcTemplate-and-JPA?)

## Spring Data JDBC

### How does Spring provide DAO support?

Spring DAO support makes it easy to work with different data access technologies in a consistent way which makes it easy to switch between technologies like JDBC, Hibernate, or JDO. Main features of Spring DAO support are:

* **Exception handling**. Technology specific exceptions like **SQLExceptions** are translated to Spring’s **DataAccessExceptions**. Checked exceptions in Hibernate, JPA, and JDO can be converted to runtime exceptions.
* **Abstract DAO support classes**. Spring provides abstract classes for different data access technologies which provide methods for data source and other configuration settings. These include the JdbcDaoSupport, HibernateDaoSupport, JdoDaoSupport, and JpaDaoSupport classes.

### Describe Spring DAO support classes.

Spring provides technology specific abstract classes for DAO support. These classes have methods for providing data source and other configuration settings specific to the data access technology. Abstract DAO support classes are:

* JdbcDaoSupport class requires DataSource and returns JdbcTemplate instance.
* HibernateDaoSupport class requires SessionFactory and returns HibernateTemplate instance.
* JdoDaoSupport class requires PersistenceManagerFactory and returns JdoTemplate instance.
* JpaDaoSupport class requires EntityManagerFactory and returns JpaTemplate instance.

### What annotation is used to mark DAO in Spring?

The @Repository annotation is used for DAO classes. This annotation provides the Spring exception translation facility which converts a technology specific expectation to a **DataAccessException**.

### What is Spring Data JDBC?

Spring Data JDBC framework is part of Spring Data. It provides support for JDBC repositories. Spring Data JDBC is a simple but limited ORM. It does not implement features like caching, lazy loading, and write-behind. Unlike Spring Data JPA, it does not provide schema generation and the developer must explicitly create the schema.

Spring JDBC uses methods like update(String sql), execute (String sql), and query(String sql, ResultSetExtractor rse) to interact with the database.

### What tasks are performed by Spring JDBC?

Spring JDBC is designed to reduce boilerplate code. It performs the following tasks:

* Opens the connection
* Prepares and executes statement
* Iterates over the resultset
* Handles any exceptions that arise
* Manages the transaction
* Closes the connection

### What is the difference between JDBC and Spring JDBC?

Spring JDBC removes the need of manual exception handling and transaction management.

Checked exceptions that need to be explicitly written in JDBC are converted to runtime exceptions in Spring JDBC.

Spring JDBC automatically cleans up resources by closing the database connection.

### Name the classes in Spring JDBC API?

* JdbcTemplate class is the central class that handles JDBC workflow managing creation of SQL statement and its execution.
* NamedParameterJdbcTemplate class provides support for named parameters to replace the ? placeholders in a query.

### What are the advantages of JdbcTemplate in Spring?

The JdbcTemplate class uses the JDBC API to connect to a database and execute SQL queries. JdbcTemplate class removes the need to write boilerplate JDBC connection management code. It also removes the need for manual exception handling.`

### Why is NamedParameterJdbcTemplate class used?

The NamedParameterJdbcTemplate class is used to pass a value to a named parameter. It is better than a prepared statement with question marks. When we use names instead of ? it is better to remember the data in the column.

### How are records from a database fetched when using JdbcTemplate?

The query method of the JdbcTemplate class is used to fetch records from a database. The results are mapped using either the ResultSetExtractor, or the RowMapper interface.

### What is the difference between a RowMapper and ResultSetExtractor?

The RowMapper is executed for every row. It keeps looping over the result set and makes it available one row at a time. On the other hand, the ResultSetExtractor fetches all the records at once.

RowMapper is used when each row maps to a single object, e.g., return a list of players along with their country.

If we need to map multiple rows returned by a query to a single object, ResultSetExtractor is the appropriate choice, e.g., return a list of countries along with the number of players from that country. ResultSetExtractor allows manipulation of data as all rows of the result set are available at once.

### Explain Spring’s exception handling support with DataAccessException.

DataAccessException is a kind of runtime exception. It wraps technology specific exceptions to let the developer handle errors without knowing the details of the data access API which means that it is possible to catch JDBC exceptions in a generic way without knowing that JDBC is being used in the DAO layer. This allows one to change between data access technologies (e.g., JDBC, JPA, Hibernate) without changing a lot of code.

The developer need not write catch blocks for runtime exceptions.

### What is SQLExceptionTranslator?

The SQLExpectionTranslator interface translates a SQLException to Spring’s DataAccessException. The returned exception contains the original SQLException as the root cause.

### What are JdbcTemplate callback interfaces?

JdbcTemplate processes a query with the questionmark (?) placeholder. Spring offers a number of callbacks to execute the query. These are as follows:

* PreparedStatement: operates on a prepared statement and can execute multiple operations on a single prepared statement.
* ResultSetExtractor: processes the entire result set at once.
* RowCallbackHandler: can perform processing on each row like converting rows to XML, streaming them to a file and filtering rows.
* RowMapper: maps a single row of the result set to an object

## Spring Data JPA

### What is Spring Data?

Spring Data is a Spring project for providing a consistent way of accessing different data stores. It provides support for both relational and no-sql databases. Spring Data contains many modules, with the core modules being spring-data-commons. Other modules include spring-data-jdbc, spring-data-redis, and spring-data-mongo.

The aim of Spring Data is to bring consistency in the way data is accessed in SQL and NoSQL databases.

### What is JPA?

Java Persistence API (JPA) is an interface that defines a mapping between relations and objects. It defines how a Java object is mapped to a database table. JPA provides a mechanism to query the database and provide the results back as a collection of Java objects or provide Java objects to be stored in the database. The underlying ORM implementation converts these API calls to SQL queries.

When using JPA, we can easily change the underlying data access technology e.g., using Oracle in production and H2 during testing.

### What is the difference between JPA and Spring Data JPA?

JPA is a specification for accessing a data store for storage and retrieval.

Spring Data JPA is a part of Spring Data project. It is not a JPA implementation. Spring Data JPA uses Hibernate as the default JPA implementation.

Spring Data JPA contains JPA repositories which define query methods. This eliminates the need to write common queries. The name of the method declared in the repository interface is automatically converted into low level SQL queries by Spring.

### What is the difference between CrudRepository and JpaRepository?

CrudRepository provides CRUD operations.

JpaRepository extends the PagingAndSortingRepository which extends the CrudRepository.

JpaRepository provides methods for pagination and sorting of records in addition to CRUD operations.

### Which No SQL databases does Spring support?

Spring supports a number of No SQL databases including MongoDB, Cassandra, Redis and HBase.

### Which ORMs are supported by Spring?

Hibernate, JDO, Oracle TopLink and OJB are supported by Spring.

### What is Hibernate?

Hibernate is by far the most popular JPA implementation. It existed even before JPA came into existence and has a lot more to it than being a JPA implementation. Hibernate makes creating queries under the hood and interacting with databases easy.

### What are the different types of transaction management supported by Spring?

Spring provides support for both programmatic and declarative transaction management. Programmatic transaction management offers more flexibility but it is difficult to maintain as it requires controlling transactions through code. The transaction management code is tightly bound with the business logic as the developer is responsible to manually commit or rollback a transaction.

In declarative transaction management, the business logic and transaction management code is separate. Transactions are implemented as a cross-cutting concern or by using the @Transactional annotation.

### Which transaction management type is preferred in Spring?

Declarative transaction management which separates transaction management from the business logic is preferred because it can be implemented as a cross-cutting concern.

### What are some benefits of using Spring Transactions?

* Spring transaction management allows the developer to handle transactions across different data access technologies in a consistent way.
* Declarative transaction management is implemented using the Spring aspect-oriented programming and does not impact application code.
* Programmatic transaction management API offered by Spring is simpler as compared to Java Transaction API.

### What happens when the @Transactional annotation is used on a method?

The @Transactional annotation is used in declarative transaction management and is used to handle transactions. It is based on the AOP concept.

Under the hood, the @Transactional annotation behaves in the same manner as the transaction advice. Spring creates a proxy around the bean to inject transaction management behavior before, after or around the method calls on the bean.

The @Transactional annotation supports both local and global transactions. It can only be applied to public methods.

### What is the difference between JdbcTemplate and JPA?

In JdbcTemplate, the database schema is not accessed via a domain model. JdbcTemplate allows more flexibility as it works on a lower level, but it also increases boilerplate code.

JPA requires that the database schema is mapped to a domain model. It makes interacting with databases simpler.

# Database Relationships in Spring

**We'll cover the following**

* [Interview questions about implementing database relationships in spring.](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JEo1ZoAPnjg#Interview-questions-about-implementing-database-relationships-in-spring.)
  + [What are the database relationship annotations in JPA?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JEo1ZoAPnjg#What-are-the-database-relationship-annotations-in-JPA?)
  + [Name some annotations which can be used to customize database relationships.](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JEo1ZoAPnjg#Name-some-annotations-which-can-be-used-to-customize-database-relationships.)
  + [What is the mappedBy attribute?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JEo1ZoAPnjg#What-is-the-mappedBy-attribute?)
  + [What is lazy loading?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JEo1ZoAPnjg#What-is-lazy-loading?)
  + [What happens if the owning side of the relationship is saved before the inverse side?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JEo1ZoAPnjg#What-happens-if-the-owning-side-of-the-relationship-is-saved-before-the-inverse-side?)
  + [Why should CascadeType.REMOVE be used after due consideration?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JEo1ZoAPnjg#Why-should-CascadeType.REMOVE-be-used-after-due-consideration?)
  + [What is the difference between CascadeType.REMOVE and orphanRemoval attribute?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JEo1ZoAPnjg#What-is-the-difference-between-CascadeType.REMOVE-and-orphanRemoval-attribute?)

## Interview questions about implementing database relationships in spring.

### What are the database relationship annotations in JPA?

Four annotations are used to define the underlying database structure in JPA. These are @OneToOne, @OneToMany, @ManyToOne, and @ManyToMany. Each relationship must be marked with one of the annotations.

### Name some annotations which can be used to customize database relationships.

JPA provides annotations to customize the database columns, tables and foreign key values. @JoinColumn, @JoinColumns, @JoinTable, @PrimaryKeyJoinColumn and @PrimaryKeyJoinColumns annotations can be used to change the default values.

It should be noted that these annotations can be used on the owning side of the relationship only. Using them on the inverse side will result in an error.

### What is the mappedBy attribute?

The mappedBy attribute is used on the inverse side of a relationship. This attribute specifies the name of the property in the owner side of the relationship.

If the mappedBy attribute is missing, JPA treats the bidirectional relationship as two unidirectional relationships.

### What is lazy loading?

Lazy loading means that data is only loaded when it is needed. When we fetch the Tournament entity, the associated Registrations are not loaded. If the EntityManager is closed or id before the inverse siddef the entity is detached, then we will not be able to get the registrations.

### What happens if the owning side of the relationship is saved before the inverse side?

The owning side (parent) keeps a reference (foreign key) of the inverse side (child). When the owning entity is persisted before the inverse entity, Hibernate can automatically reorder the SQL statements to avoid relationship dependency problems. Some JPA providers do not resolve relationship dependencies and throw an exception or a foreign key violation constraint. Additional configuration is needed in such a case.

### Why should CascadeType.REMOVE be used after due consideration?

In one-to-many or many-to-many relationships, CascadeType.REMOVE should be used with care. Reason being, it removes more records than necessary. For example, deleting a Professor entity deletes all the Courses taught by the professor.

Same goes for CascadeType.ALL. In to-many relationships all the cascade types should be explicitly mentioned instead of using CascadeType.ALL.

### What is the difference between CascadeType.REMOVE and orphanRemoval attribute?

Both CascadeType.REMOVE and orphanRemoval attribute are used to remove records from database. When the association between the parent and child entities is removed, the child entity becomes an orphan. Using CascadeType.REMOVE does not remove the orphaned entity. orphanRemoval attribute has a more aggressive approach and removes the orphaned record as soon as the association between the parent and child entity is removed.

# Spring AOP

**We'll cover the following**

* [Spring AOP interview questions](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7n99Mz0LxPr#Spring-AOP-interview-questions)
  + [What is AOP and how does it relate to OOP?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7n99Mz0LxPr#What-is-AOP-and-how-does-it-relate-to-OOP?)
  + [What is a cross cutting concern in AOP?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7n99Mz0LxPr#What-is-a-cross-cutting-concern-in-AOP?)
  + [Is there a difference between the terms concern and cross cutting concern in Spring AOP?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7n99Mz0LxPr#Is-there-a-difference-between-the-terms-concern-and-cross-cutting-concern-in-Spring-AOP?)
  + [What problems does AOP solve?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7n99Mz0LxPr#What-problems-does-AOP-solve?)
  + [Name some implementations of AOP?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7n99Mz0LxPr#Name-some-implementations-of-AOP?)
  + [What is the difference between Spring AOP and AspectJ AOP?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7n99Mz0LxPr#What-is-the-difference-between-Spring-AOP-and-AspectJ-AOP?)
  + [How does Spring implement a cross cutting concern?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7n99Mz0LxPr#How-does-Spring-implement-a-cross-cutting-concern?)
  + [What is a proxy in Spring AOP?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7n99Mz0LxPr#What-is-a-proxy-in-Spring-AOP?)
  + [What is a target object?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7n99Mz0LxPr#What-is-a-target-object?)
  + [What advantage does Spring AOP provide?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7n99Mz0LxPr#What-advantage-does-Spring-AOP-provide?)
  + [What is a JoinPoint?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7n99Mz0LxPr#What-is-a-JoinPoint?)
  + [What is advice?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7n99Mz0LxPr#What-is-advice?)
  + [List the different types of advice in AOP?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7n99Mz0LxPr#List-the-different-types-of-advice-in-AOP?)
  + [Which advice type is appropriate for a try catch block?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7n99Mz0LxPr#Which-advice-type-is-appropriate-for-a-try-catch-block?)
  + [What is the difference between Joinpoint and ProceedingJoinPoint?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7n99Mz0LxPr#What-is-the-difference-between-Joinpoint-and-ProceedingJoinPoint?)
  + [What is pointcut?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7n99Mz0LxPr#What-is-pointcut?)
  + [What is a named pointcut?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7n99Mz0LxPr#What-is-a-named-pointcut?)
  + [What is an Aspect?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7n99Mz0LxPr#What-is-an-Aspect?)
  + [What is weaving?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7n99Mz0LxPr#What-is-weaving?)
  + [When does the Spring framework perform weaving?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7n99Mz0LxPr#When-does-the-Spring-framework-perform-weaving?)
  + [Which AspectJ Pointcut Designators are supported by Spring AOP?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7n99Mz0LxPr#Which-AspectJ-Pointcut-Designators-are-supported-by-Spring-AOP?)
  + [Are there any limitations of Spring AOP?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/7n99Mz0LxPr#Are-there-any-limitations-of-Spring-AOP?)

## Spring AOP interview questions

### What is AOP and how does it relate to OOP?

AOP stands for Aspect Oriented Programming. It divides the application logic into parts called concerns. AOP is similar to OOP in terms of modularity. It differs from OOP in the way modularity is defined. In OOP modularity is achieved by using classes whereas in AOP modularity is achieved using aspects that are applied to different class methods. Since aspects or concerns apply to the whole application, their processing should be centralized.

### What is a cross cutting concern in AOP?

A cross cutting concern is something that affects the whole application. Instead of handling it separately in every layer, it is handled in a centralized manner. Examples of cross cutting concerns are security, logging, transaction management, and error handling etc.

### Is there a difference between the terms concern and cross cutting concern in Spring AOP?

**Concern** can be defined as the functionality of a particular module in the application. For example in an e-commerce application, concerns may be inventory management and user management etc.

A **cross cutting concern** is a concern that is applicable across multiple application layers. For example logging and security functionality is needed by every module of an application.

### What problems does AOP solve?

AOP modularizes the code in terms of cross cutting concerns. The absence of AOP leads to two main problems.

* **Code tangling**: the code for the cross cutting concern gets mixed with business logic.
* **Code scattering**: Identical code is present in all modules.

### Name some implementations of AOP?

Popular implementations of AOP include Apache AspectJ, JBoss by Red Hat and Spring AOP.

### What is the difference between Spring AOP and AspectJ AOP?

Spring AOP does weaving at runtime using proxy while AspectJ does compile time weaving using AspectJ Java tools.

Only method level pointcuts are supported by Spring AOP while AspectJ also supports field level pointcut.

### How does Spring implement a cross cutting concern?

Spring AOP separates the business logic and cross cutting concern. The developer focuses on the program logic. Spring provides many aspects for cross cutting concerns which can be woven into the application.

### What is a proxy in Spring AOP?

In simple words, a proxy is an object that looks like another object but has some added functionality. In Spring AOP, proxy is the object created after applying advice to a target object.

Proxy = Advice + Target Object

Target object is also called a proxy object.

### What is a target object?

Target object is an object to which a cross cutting concern has been added. It is also called advised object or proxy object.

### What advantage does Spring AOP provide?

Spring AOP allows us to add or change a concern without having to change the application code. Since we separate the concerns from the application logic, concerns can be dynamically plugged in before, after or around the application logic. The code also becomes easy to maintain. Another advantage is that the developer can concentrate on business logic rather than the cross cutting concerns.

Spring AOP configures aspects as normal beans. Also, no special compilation unit is needed when using Spring AOP.

### What is a JoinPoint?

JoinPoint is a point in the program such as method execution where an aspect can be plugged in. There are different types of JoinPoints like field access, error handling and method execution.

Spring AOP only provides support for method execution join points. Any method inside the class can be called a join point if any cross cutting concern is applied to it and an aspect’s code is inserted into the normal flow of the application.

### What is advice?

The logic of the aspect is called advice. It is the action that is taken when an aspect is executed. In terms of programming, advice is the execution of the method where a joinpoint matches a pointcut.

### List the different types of advice in AOP?

There are 5 types of advice:

* Before
* After
* After Returning
* After Throwing
* Around

### Which advice type is appropriate for a try catch block?

In order to try and catch exceptions, the @AfterThrowing advice type is used. The method annotated with this annotation is run after the method exits by throwing an exception.

### What is the difference between Joinpoint and ProceedingJoinPoint?

Proceedingjoinpoint extends the Joinpoint interface.

Joinpoint is used with the @Before, @After, @AfterReturning and @AfterThrowing advice types.

Proceedingjoinpoint is used with @Around advice. The @Around advice type is different from the rest because it can control when/if a method is executed. It also has a return value.

### What is pointcut?

Pointcut is the expression that is matched to a JoinPoint to determine whether the advice should be executed or not. Spring framework uses the AspectJ pointcut expression language. These contain matching method or class name patterns.

### What is a named pointcut?

When we need a pointcut at multiple places in the application, rather than using the lengthy pointcut expression, we can give it a name. This is done by creating a pointcut configuration class where we associate every pointcut with a method. Now the method name can be used in place of the long pointcut expression.

### What is an Aspect?

An aspect is a class denoted by the @Aspect annotation. It contains advice and joinpoints. Aspect defines a concern that cuts across multiple application layers.

### What is weaving?

Weaving is a process in which the aspects are plugged in at different points in the program execution. In Spring AOP, weaving is done at runtime. AspectJ provides both compile-time and load-time weaving.

### When does the Spring framework perform weaving?

Spring framework performs weaving at runtime. The process of weaving aspects into the application classes takes place when the classes are being loaded in JVM.

### Which AspectJ Pointcut Designators are supported by Spring AOP?

Spring provides support for some of the AspectJ Pointcut Designators (PCD) that can be used in pointcut expressions. For example:

* **execution**: matches method execution joinpoints
* **within**: matches to joinpoints of certain types
* **this**: matches to joinpoints where the target object is of a given type
* **args**: matches to joinpoints where the arguments of the given type
* **@annotation**: matches to joinpoints where the method has a given annotation

### Are there any limitations of Spring AOP?

* Spring AOP only supports method level joinpoints.
* Advice is only applicable on public methods. Methods with private or protected visibility cannot be advised.
* When weaving with proxies, advice is not executed on local method calls.
* Aspects can only be applied to Spring beans.

# Spring MVC

**We'll cover the following**

* [Spring MVC interview questions](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JY3oNnPK7yo#Spring-MVC-interview-questions)
  + [What is Spring MVC?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JY3oNnPK7yo#What-is-Spring-MVC?)
  + [What are the advantages of using Spring MVC?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JY3oNnPK7yo#What-are-the-advantages-of-using-Spring-MVC?)
  + [What is the flow of request in the MVC architecture?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JY3oNnPK7yo#What-is-the-flow-of-request-in-the-MVC-architecture?)
  + [What are the steps required to create a Spring MVC application?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JY3oNnPK7yo#What-are-the-steps-required-to-create-a-Spring-MVC-application?)
  + [Is spring-mvc jar included in spring-core?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JY3oNnPK7yo#Is-spring-mvc-jar-included-in-spring-core?)
  + [What is the Front Controller Pattern?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JY3oNnPK7yo#What-is-the-Front-Controller-Pattern?)
  + [Describe the function of the DispatcherServlet.](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JY3oNnPK7yo#Describe-the-function-of-the-DispatcherServlet.)
  + [How is the Dispatcher servlet configured?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JY3oNnPK7yo#How-is-the-Dispatcher-servlet-configured?)
  + [How does the dispatcher servlet map a request to a controller method?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JY3oNnPK7yo#How-does-the-dispatcher-servlet-map-a-request-to-a-controller-method?)
  + [What is a controller?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JY3oNnPK7yo#What-is-a-controller?)
  + [What is the function of @Controller annotation?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JY3oNnPK7yo#What-is-the-function-of-@Controller-annotation?)
  + [What is the difference between @Controller and @RestController annotations?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JY3oNnPK7yo#What-is-the-difference-between-@Controller-and-@RestController-annotations?)
  + [What is the function of @RequestMapping annotation?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JY3oNnPK7yo#What-is-the-function-of-@RequestMapping-annotation?)
  + [When using @RequestMapping annotation, can multiple paths map to the same controller method?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JY3oNnPK7yo#When-using-@RequestMapping-annotation,-can-multiple-paths-map-to-the-same-controller-method?)
  + [When using @RequestMapping annotation, can the same HTTP request be mapped to multiple controller methods?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JY3oNnPK7yo#When-using-@RequestMapping-annotation,-can-the-same-HTTP-request-be-mapped-to-multiple-controller-methods?)
  + [When using @RequestMapping annotation, can multiple HTTP request methods map to the same controller method?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JY3oNnPK7yo#When-using-@RequestMapping-annotation,-can-multiple-HTTP-request-methods-map-to-the-same-controller-method?)
  + [What is an ambiguous mapping error?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JY3oNnPK7yo#What-is-an-ambiguous-mapping-error?)
  + [What is the function of @ResponseBody annotation?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JY3oNnPK7yo#What-is-the-function-of-@ResponseBody-annotation?)
  + [What are the shortcut annotations for HTTP request methods?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JY3oNnPK7yo#What-are-the-shortcut-annotations-for-HTTP-request-methods?)
  + [What is the isELIgnored attribute and what is its default value?](https://www.educative.io/courses/guide-spring-5-spring-boot-2/JY3oNnPK7yo#What-is-the-isELIgnored-attribute-and-what-is-its-default-value?)

## Spring MVC interview questions

### What is Spring MVC?

Spring MVC is the module of Spring that implements the front controller and MVC design pattern. It provides a decoupled approach to developing web applications.

In Spring MVC, the three components of the MVC architecture, the front controller, view resolver and model are not dependent on each other. All incoming requests are handled by the front controller which is called the DispatcherServlet.

### What are the advantages of using Spring MVC?

* Spring MVC is based on interfaces which are independent of each other allowing separation of concerns.
* Because there is no explicit dependency between the interfaces, Spring MVC applications are easily testable.
* The view technology is customizable and switching between JSP, Velocity or Thymeleaf etc. can be done easily.
* Spring MVC supports RESTful web services.

### What is the flow of request in the MVC architecture?

The dispatcher servlet is the front controller which receives all requests from the client. It has a mapping of all controllers and maps the incoming request to the appropriate controller.

The controller executes the request and returns a model and view name to the dispatcher servlet. The model contains the results.

The dispatcher servlet uses the view resolver to resolve the name of the view and populates it with results from the model and displays it to the client as a web page.

### What are the steps required to create a Spring MVC application?

To create a simple MVC application, we perform the following steps:

* Include the spring-context and spring-webmvc dependencies in the **pom.xml** file.
* Define the dispatcher servlet in **web.xml** to handle all requests.
* Configure the controller classes using XML or annotations.
* Create URL mappings in controller classes to handle incoming requests.
* Configure a view resolver.

### Is spring-mvc jar included in spring-core?

spring-mvc jar is not a part of spring-core. It needs to be added separately to the classpath. It has a dependency on spring-core jar which is downloaded automatically if a build tool like Maven is used.

### What is the Front Controller Pattern?

Front Controller Pattern is a design pattern in which request handling is centralized. All client requests go to the front controller which contains a mapping of all controllers/servlets. The front controller maps the client request to the relevant controller/servlet, receives the results back, and renders the view to the client.

In Spring **DispatcherServlet** is the front controller. It receives all the requests from the client and forwards them to the appropriate controller.

### Describe the function of the DispatcherServlet.

The DispatcherServlet is at the heart of Spring MVC. It is the front controller which handles all the HTTP requests and responses.

The dispatcher servlet uses a configuration file containing information of all controllers and the URL mappings to forward an incoming request to the correct controller. It gets a model object and view back from the controller. The dispatcher servlet then forwards the view name to the view resolver and renders the response back to the client.

### How is the Dispatcher servlet configured?

Dispatcher servlet can be configured using XML or programmatically using the ServletContainerInitializer interface.

* **XML configuration**: The dispatcher servlet is configured in the **web.xml** file like any other servlet. The <servlet-name> tag specifies the name, while <servlet-class> tag specifies the class as **org.springframework.web.servlet.DispatcherServlet**.

To direct all requests to go through the dispatcher servlet, the <servlet-mapping> tag is used with the “**/\***” URL mapping. The <load-on-startup> tag with value 1 is used to preload the dispatcher servlet, otherwise it is loaded when a request comes.

* **Java configuration**: Spring provides its own implementation of the ServletContainerInitializer interface. We can use the class AbstractAnnotationConfigDispatcherServletInitializer and provide implementation of the getServletConfigClasses and getServletMappings methods to configure the dispatcher servlet.

### How does the dispatcher servlet map a request to a controller method?

The dispatcher servlet uses handler mappings and annotations like @Controller and @RequestMapping to map a request to a controller method.

The HandlerMapping interface provides a mapping between requests and handler objects. Spring provides implementations of this interface where incoming request URLs can be mapped to controller classes.

### What is a controller?

A controller is a class that handles user requests. It is annotated with the @Controller annotation.

A controller class contains the business logic of the application. Based on the client request, a controller populates the model which is shown to the user as a view.

### What is the function of @Controller annotation?

The @Controller annotation is used at class level. It marks the class as a controller class and tells the Spring framework that this class will handle user requests.

### What is the difference between @Controller and @RestController annotations?

Web applications and REST APIs differ in their return types with the former returning a view comprising of HTML, CSS and JavaScript while the later returning JSON or XML data.

The @Controller annotation populates the model map and returns a view name that is sent to the view resolver. The @RestController annotation returns an object which is written to HTTP response as JSON or XML.

The @RestController annotation is a combination of @Controller and @ResponseBody annotations. To return JSON or XML in a web application, we need to use the @ResponseBody annotation explicitly with the @Controller annotation.

### What is the function of @RequestMapping annotation?

When the @RequestMapping annotation is used at class level, it acts as a parent mapping which maps the URL of an incoming request.

When it is used on method level, it maps the URL as well as the HTTP request method.

### When using @RequestMapping annotation, can multiple paths map to the same controller method?

Usually, the @RequestMapping annotation is used to map a single path to a controller method but this is not a rule. Multiple mappings can be specified in the value attribute as can be seen from the following example:

@RequestMapping(value = { "/hello", "/hi" })  
@ResponseBody  
public String multipleMapping() {  
    return "Hello World!";  
}

Both **/hello** and **/hi** are mapped to the multipleMapping method which returns a String response.

### When using @RequestMapping annotation, can the same HTTP request be mapped to multiple controller methods?

We can have multiple methods in the controller with the same request mapping as long as the request method is different. In the following example, **/userForm** is mapped to both the showForm and processForm methods. One method is for GET request while the other is for POST request.

@RequestMapping(value = "/userForm")  
public String showForm() {  
    return "user-form";  
}  
  
@RequestMapping(value = "/userForm", method = RequestMethod.POST)  
public String processForm() {  
    return "user-confirmation";  
}

When @RequestMapping annotation is used without the method attribute, it defaults to HTTP GET request.

### When using @RequestMapping annotation, can multiple HTTP request methods map to the same controller method?

Yes, it is possible to map different HTTP request methods to the same controller method. The method attribute accepts multiple values. In the following example, both PUT and POST requests to **/userForm** are mapped to the addUser method.

@RequestMapping(value = "/userForm",   
                method = {RequestMethod.PUT, RequestMethod.POST})  
public String addUser() {  
    return "user-confirmation";  
}

### What is an ambiguous mapping error?

When Spring finds more than one controller methods having the same URL mapping, HTTP request method, parameters, headers and media type, it throws the ambiguous mapping error because it cannot choose which controller method to map the incoming request to. Changing anything from the above mentioned list to differentiate the controller methods will resolve this error.

### What is the function of @ResponseBody annotation?

The value returned by a controller method is resolved to a view name. However, if we want to write directly to the response body, we use the @ResponseBody annotation to tell Spring that instead of resolving the return value as a view name, it should be displayed to the client as a String.

The following method resolves “hello-world” as a view name and displays the page by this name to the user.

@RequestMapping(value = "/hello")  
public String welcome() {  
    return "hello-world";  
}

When the @ResponseBody annotation is used on the same method, “hello-world” is displayed as the response.

@RequestMapping(value = "/hello")  
@ResponseBody  
public String welcome() {  
    return "hello-world";  
}

### What are the shortcut annotations for HTTP request methods?

The @RequestMapping annotation can be used with the method attribute to specify the HTTP request method. Spring also provides shortcut annotations for five HTTP request types which are as follows:

* @GetMapping
* @PostMapping
* @PutMapping
* @PatchMapping
* @DeleteMapping

The purpose of these annotations is to make the code short and readable. Behind the scene, all these annotations are resolved to the @RequestMapping annotation with the respective HTTP method attribute. The example below shows the same request mapping in two different ways.

@RequestMapping(value = "/userForm", method = RequestMethod.POST)  
//OR  
@PostMapping((value = "/userForm")

### What is the isELIgnored attribute and what is its default value?

The isELIgnored attribute is used to disable the evaluation of scriptlets. The default value of this attribute is true for JSP 1.2. For JSP 2.0 and above, the default value is false.