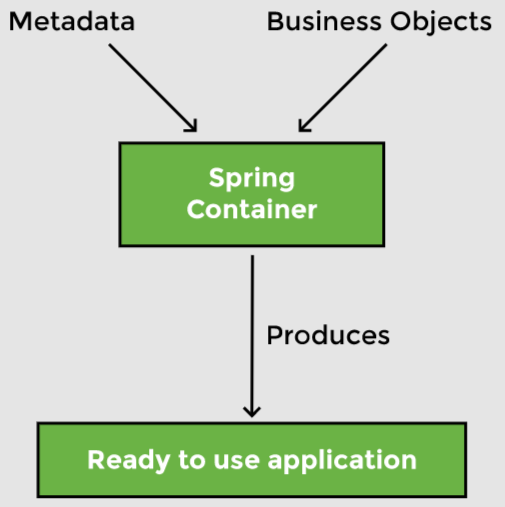
Introduction to Spring

# What is Spring?

**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.**



**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 } |

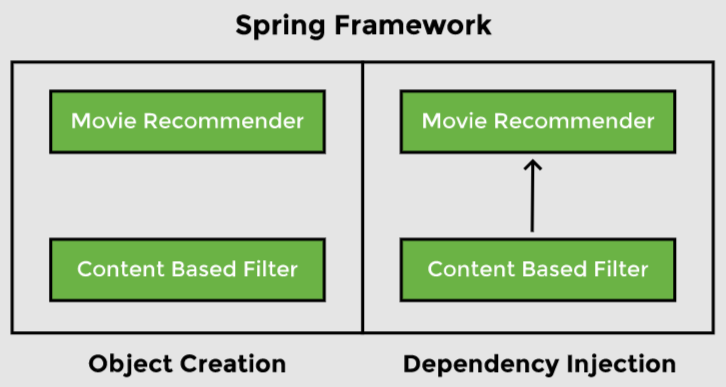
|  |
| --- |
| public class ContentBasedFilter implements Filter {  //implement interface methods } |

|  |
| --- |
| 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. The loose coupling has a number of advantages which we will discuss in later lessons.**

|  |
| --- |
| 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. We will discuss at length how the Spring framework performs this magic in the next section.**



**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 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.**

# Beans

**Beans are the objects of classes that are managed by Spring. Traditionally, objects are 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 a constructor or by using a setter method.**

# Inversion of Control

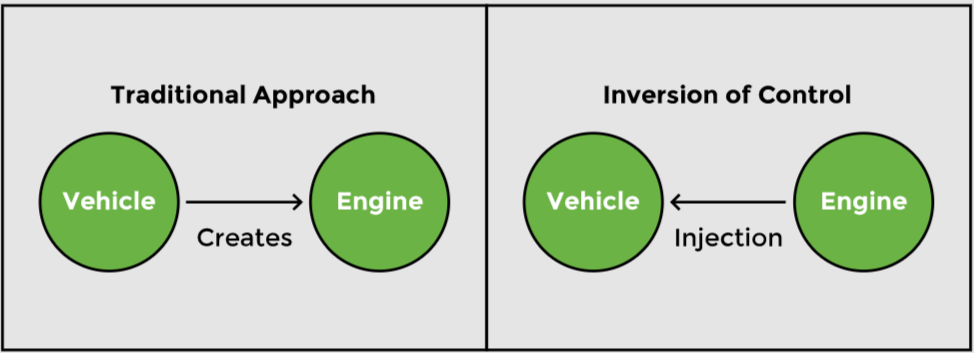
**Traditionally, the class which needed the dependency created an instance of the dependency. The class decided when to create the dependency and how to create it. For example, the Engine class is a dependency of the Vehicle class, which creates its object:**

|  |
| --- |
| 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 into 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 the 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:**

# Bean factory

## Application context

**Both of them are interfaces that have different implementations available. The 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, the bean factory should be used.**

## Bean factory

**The basic version of the Spring IoC container is the bean factory. It is the legacy IoC container and provides basic management for beans and the wiring of dependencies. In Spring, the 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 the application context include Spring AOP features, internationalization, web application context, etc.**

## @Component

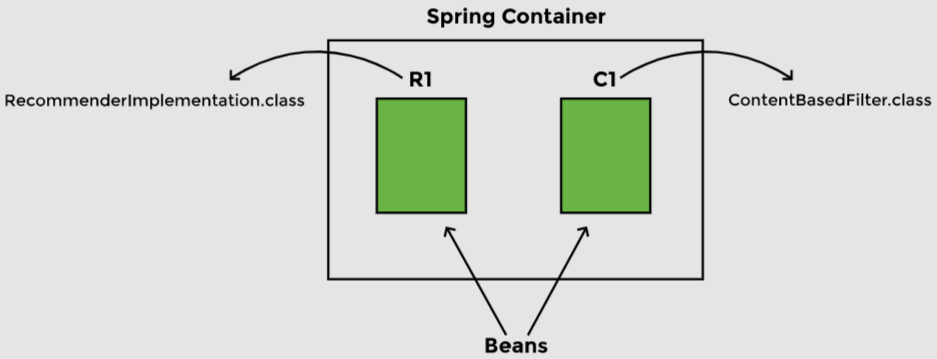
**If we want Spring to create and manage objects, we can do so by adding the @Component annotation at the beginning of the class and importing org. spring framework. stereotype.Component.**

**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**

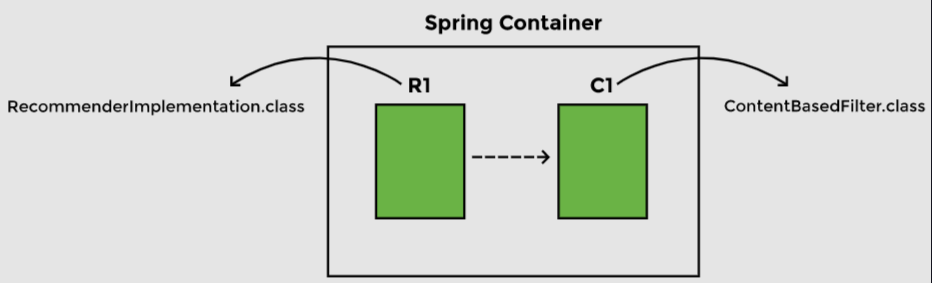
## @Autowired

**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. spring frameworkk.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.**

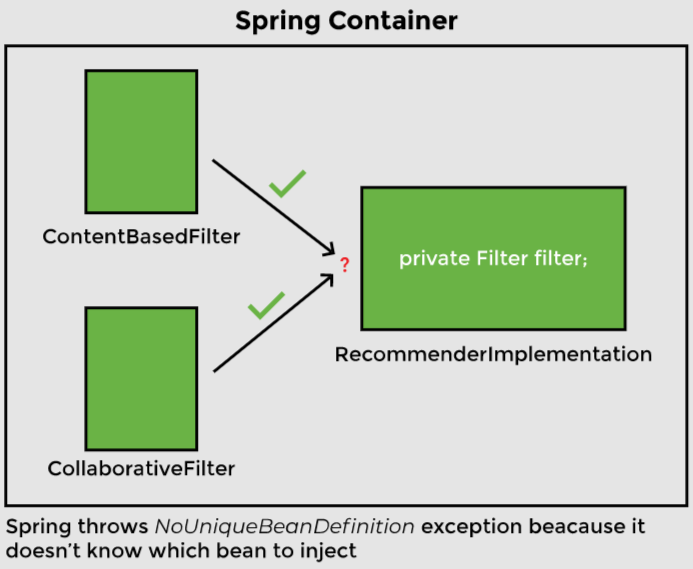


**Identifying bean dependencies**

**Autowiring By Type — @Primary**

## NoUniqueBeanDefinitionException

**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.**

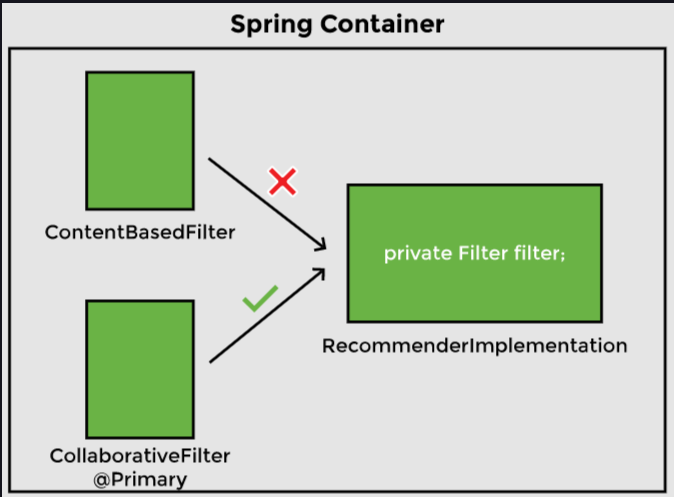


**The NoUniqueBeanDefinitionException occurs. The error message says: Required a single bean but two were found.**

## @Primary annotation

**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. spring framework.context.annotation.Primary. When we run the application now, it uses CollaborativeFilter as expected.**



**Bean with `@Primary` annotation is injected**

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

**If we make both beans primary by adding the @Primary annotation to both implementations of the Filter interface, we will get an error.**

## Autowiring By Name

**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.**

**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).**

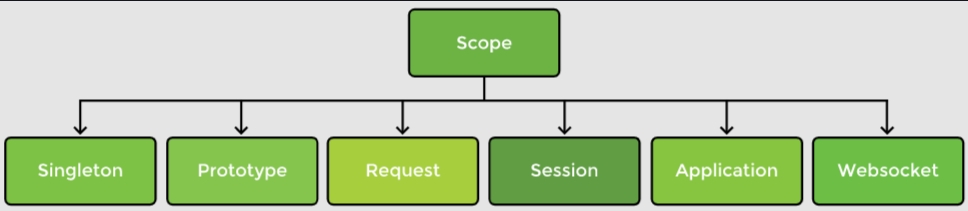
|  |
| --- |
| public class RecommenderImplementation {  @Autowired  private Filter contentBasedFilter;  *//...* } |

|  |
| --- |
| @Component public class ContentBasedFilter implements Filter{  *//...* } |

## Bean Scope

**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.**

**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 a 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.**

## Prototype scope

**The default scope of a bean is a 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.**