# Course Overview

## Course Overview

Hi everyone. My name is Bryan Hansen. Welcome to my course, Spring Fundamentals. I'm the director of software development at Software Technology Group. The Spring Framework has taken the Java community by storm, making web and enterprise development much simpler than it previously was. In this course, we're going to cover the fundamentals of Spring development by going through the various configuration methods that are available inside Spring. Some of the major topics that we will cover include Java configuration, autowiring, and advanced bean configuration. By the end of this course, you'll know the configuration methods available in Spring and how they'll benefit your code. Before beginning the course, you should be familiar with Java, but no previous Spring experience is required. From here, you should feel comfortable diving into Spring with courses on Spring MVC, Spring Data JPA, and Spring Security. I hope you'll join me on this journey to learn the fundamentals of Spring development with the Spring Fundamentals course, at Pluralsight.

# What Is Spring?

## Version Check

## Introduction

What is all the hype around the Spring Framework? Do you already know that you want to learn Spring, but aren't sure where to start? Hello. I'm Bryan Hansen, and welcome to my course on Spring Fundamentals on Pluralsight. In this course, we're going to be learning about developing with Spring, and we'll use the various configuration methods to illustrate some of the common problems we often face in development. We will develop part of an application without Spring and then show how to configure that part of the application using Java configuration, XML configuration, and annotations. We will also discuss the lifecycle of Spring and how to use beans in a more appropriate fashion.

## What Is Spring?

What is Spring? Well, the Spring Framework started out as just an inversion of control container. This technique is often referred to as dependency injection. It was conceived to reduce or replace some of the complex configuration of the earlier Java Enterprise Edition. Spring was later built around using Java without EJBs, so its original concept was how to work better with EJBs, and then they realized they just didn't need those for a lot of situations. So it kind of transitioned into, okay, how can I do the same development without using Enterprise JavaBeans? This course is designed to show you how to use Spring, and since Spring is a dependency injection container, we will also discuss why. Dependency injection is very simply removing hard-coded wiring in your app and using a framework to inject dependency resources where they are needed.

## Update

So, again, what is Spring? Well, it is a framework built around reducing the complexities around enterprise Java development and later also providing enterprise development without EJBs. Spring can essentially be used with or without Enterprise JavaBeans and primarily now is used without them. This is an important point because Spring enabled us to do enterprise development without using an application server. A lot of people don't realize that Tomcat really isn't an application server, and it's just a web server. This is one of the reasons that Tomcat has taken over for Java developments as a standard container. It's very easy to use, it's lightweight, and until Spring, you either weren't using enterprise features or you had to use a more complex, harder-to-use application server. I should also mention that it is completely POJO based. Any code you write in Spring can be written without using Spring at all. POJO, as you may or may not remember, stands for a plain old Java object. The Spring Framework really isn't doing any magical thing behind the scene. It's just helping us write better, cleaner code and doing things POJO based and interface driven. So we've already talked about it being lightweight, but we should also point out that Spring was built out of the frustrations of JEE, so it's very unobtrusive. It shouldn't be getting in your way. If it is, you're using it wrong. Spring also uses AOP and proxies to apply things like transactions to your code to get those cross cutting concerns out of your application, so your code should actually be smaller and more lightweight from using Spring. One of the most appealing parts of Spring to me is that it's built around best practices. We end up having design patterns in our code without us even realizing that we're using them, things like singletons, factories, abstract factories. All of those best practices are built into our code. Some people may argue that I should know those to get into it. That's true. Having that knowledge of patterns will only help increase your understanding of what Spring is doing, but it can help make new developers or less experienced developers come up to speed and use these patterns more effectively without having that knowledge firsthand. There's patterns like a singleton in there, a factory, an abstract factory, the template method. Spring actually uses template method a lot, and although it's really not a pattern, it's a design methodology or a best practice, there's annotation-based configuration. If all of this doesn't make much sense to you, don't be concerned because you can still learn Spring without understanding many of those details.

## The Problem

What problem is Spring trying to solve? Whenever I'm going to introduce another framework or tool into my code base, I want to look and see what it's actually buying me. These are the big questions I ask, what is this trying to solve for me? Well, Spring brings to the table quite a few things. It increases our testability. It increases our maintainability or the ability to maintain our code. It also helps us with scalability. I call these the ables, the testable, maintainable, scalable-type properties of our code. It decouples things and makes it so that we can add caching layers, things like that in there. There's a little bit beyond the scope of what we're going to cover in this class, but these are all focused at Spring and the principles behind Spring help us deliver on. It also helps us reduce code complexity. And lastly, though, and this is my biggest point that it tries to solve for us is that it puts the focus on our business. The business doesn't care what framework I'm using. They care about what we're getting coded done, and Spring helps us get complex code done faster, makes it more maintainable, testable, and helps remove that complexity, and just makes everything a little bit easier for us as a developer.

## Business Focus

As I mentioned in the previous slides, that Spring helps us focus on the business need. If you've ever done much database development in Java, you've used JDBC. JDBC, at first glance, doesn't look that bad, but then you really start picking it apart, and it's really not very pretty. Look at all this extra stuff we have in our code just to do a simple SELECT statement. We have connections, and prepared statements, and result sets, and then we have this try catch block. We have a DriverManager. We have all this stuff in there. And look at that great big finally block or the if rs = null, if statement = null, and if connection = null. Then we had this empty curly brace catching exceptions. There's a big bunch of just code in here when really all our business cares about is this line here where we say select whatever our fields are from the CAR table where this ID equals whatever, and then the only other line that our business really cares about is where we grab that actual string and store it in an object and return it. Now this may look like a great selling point for Hibernate instead of Spring, but actually I'm going to show you a few things that Spring does to make this easier. So, all these places where we're assigning things, and handling, and closing, and all that type of stuff can easily be contained by Spring for us. We should handle this by using Spring to inject these resources into our code.

## The Solution

The solution we're looking to get from Spring or from using the Spring Framework is that we can remove configuration code or lookup code, and developers can focus on the business needs. Like I said, that's a big one for me, and it's the business that doesn't care that I have a try catch block or how I handle this exception. They care that when we ask for a car, we get that car back. Our code can also focus on testing. So in that previous slide, it really wasn't that testable as far as things were hard coded in there. I have a DriverManager that I'm getting a connection. Where's that connection coming from? That type of stuff. Spring also helps us with doing annotation or XML-based development, as well as the Java configuration that we're going to mainly focus on. If we want to annotate our code rather than having so much configuration code in there, Spring can help us with all of that as well. To achieve all of this, Spring encourages development through interfaces. We will dive into this concept deeper in the implementation of our code, but interface-driven development really changes the approach we take with our code. So it makes things easier to test, makes things more focused on what the business needs, and removes that configuration lookup code through doing annotations or Java-based configuration or even the XML-based approach as well.

## Business Focus Revisited

Let's look at that business focus slide again, the JDBC code that we had looked at earlier, and show what it could look like by using the Spring Framework. This was our code before with all of the getConnection, DriverManager, prepareStatement stuff, that huge finally block that essentially does nothing but close connections for us. And then let's see what that code could look like while utilizing Spring. Now I will say that this one particular library in Spring, not necessarily Spring core, that this is the Spring JDBC template code, but it's the same concept that we're trying to drive in Spring. Notice that nowhere in here has it mentioned a connection result set, a prepared statement, a DriverManager. It doesn't talk about closing any of that stuff off. In fact, there's not even a try catch finally statement in here at all. We have our find by and our return statement, and this is using the EntityManager stuff in there, but this example of a template method pattern is just being done for us. This is one of those cases where Spring is using a template method pattern, and we don't really have to know how or what it's doing behind the scenes. Now this may scare some of you that there is this huge black box with Spring, and I hope to alleviate those concerns for you throughout this course, but look at how much smaller and more condensed this code is, and it's really just doing what we want it to do. There's no reason we should have to write what's on the left for every query we need to make inside of our application.

## How It Works

How does this all work? Great, we've seen that it can reduce business focus for us, but how does it do this? Well, as I mentioned earlier, everything in Spring is a simple POJO, a plain old Java object. Spring itself can be thought of as a glorified HashMap. It's actually called the application context. The application context is the configured Spring container with all of the dependencies wired up in it. Now I don't want to simplify it too much. It's doing more than just creating a HashMap and shoving objects in there, but I also don't want people to be afraid of it because there's too much going on behind the scenes. There's too much magic happening with Spring because there's really not. Spring can also be used as a registry and that's how I'm going to show you to start with because we're just going to use a little main method and an application to run it. But to really get all the benefit of Spring, we'll go through its wiring constructs and use it to autowire our application.

## Demo: What We Are Building

Since this is an update for this course, we're trying to standardize the application that we used to build across multiple courses. And we've opted to do a conference registration app, and we're going to specifically focus on the model and wiring of this application. We're going to be looking at a conference application and the configuration that we do throughout that. We'll focus primarily on the model of that wiring so the session objects for the conference session and the speakers for those sessions, and we may add attendees in there as well to demonstrate all of the characteristics that we want to for wiring up this application.

## Summary

Now that we've spoken a little bit about what Spring is, the problems that we're trying to solve with Spring, and what the solution is, we can now appreciate what that problem is and how Spring can be a solution for it. The business focus is what you're going to see me drive towards in all of our examples that we do and how this enables us to get what the business wants done faster. Let's get our project set up and dive into the architecture of our app now.

# Architecture and Project Setup

## Sample App Intro

Let's dive into a little bit of the architecture of Spring and the application that we're going to build throughout this course. We'll walk through the basic project set up and a historical approach to what Spring solves for us. We're also going to build a small sample application in this module that will show you the concepts that Spring will help us with, and we're going to use that demo in all of the subsequent demos in the rest of this course.

## Architecture

I'm not going to take you through a huge architecture discussion and show you a bunch of class diagrams or anything like that, but I want to take a minute to talk about why Spring was developed the way that it is. Spring was developed to make existing tasks easier. Before Spring came around, we used some design patterns from the JEE blueprints to help establish better code and repeatable processes. These blueprints helped to establish a consistent way of doing things, but often still made code brittle and untestable. If you've ever had to, say, recompile code because you were moving to a different environment or change things like URLs or connection strings because of your environment, then you realize these are the things that we are trying to fix. This problem is referred to as write once, run anywhere and is often abbreviated with WORA or pronounced WORA. These are the problems that we are faced with. It can be a lot simpler than this too. You just may want the actual implementation not hard coded inside your application. This is what Spring's architecture is going to help us with.

## Prerequisites

There are a few prerequisites you'll need to do the examples in this course. I'm using Java 11, and I'm assuming that you already know how to install Java, so we won't be covering that in this course. I'm also going to use Maven. I do have a module at the end where I'm going to show you how I would use Spring Boot to do this, but people have requested to have just Spring stand alone by itself. Spring no longer offers up a way to just download the _____ jars, and you're forced to use Maven or Gradle to download those dependencies. There is a Pluralsight course on Maven. If you have more questions about that, you can research that and follow it. The next thing you'll need is an IDE. This is an updated version of the previous course, and I have chosen to use IntelliJ. We got feedback that a majority of our users wanted to see and use the examples within IntelliJ, so we have changed the course to use it. And, finally, Tomcat. You won't need this for any of the examples that we're going to do walking through this course, but the sample app that I use at the end, and if you follow along with that, uses Spring Boot. I will deploy that app to Tomcat as well. So if you have it, great, if you don't, don't sweat it. Mainly, Java 11 and the latest version of IntelliJ will work.

## Sample App Setup

To begin setting up our application to show you the capabilities of Spring, I'm going to start off with opening up my IDE. I'm using IntelliJ for this course, as I had mentioned in the prerequisite section. You can see the version I'm using here, but I don't think that would really matter to you. Any recent version should work for what we're going to do inside of this course. I'm going to click Create New Project, and inside of here, I'm going to select the project SDK as Java 11. I am not going to choose create from archetype because that'll walk us through a bunch of dialogs that we don't care about for this course, and I'm going to select Maven over here on the left. Then I'm going to click Next. For our GroupId, I'm going to put in com.pluralsight, and for the ArtifactId, I just want to enter conference, and click Next. And then for our workspace I like to just have something simple in here as workspace and the project name of conference. So I have dev/workspace/conference. I'm going to click Finish. It will ask me if I want to create that directory. Yes, I do, and then it will launch my application. So as you can see inside of here now, we have a basic POM file, and it's created our folder structure for us containing the source, main, java directory, and resources directory, as well as the source, test, java directory. We're going to break down each of these in subsequent examples, so let's stop here as we have our base project set up for us now.

## Demo: Sample App pom.xml

Before we go any further inside of our application, there is actually one thing we need to fix inside of our pom.xml, and this has been the case for every IDE I have ever used. We need to add a specific build version for our app. Now you can go look in the Maven course and see why and what. Suffice it to say that we want to go below the last line of version, but before we have closed off our project. So you can see in my example here, after line 9, I've added some whitespace, so on line 11, I'm going to add this build section, and I'm just going to paste it in and walk you through it. You can add it on your own. So I have added in here a build section, and inside of that, there is a plugins section, plural, and then an individual configured plugin. And in here, we have a groupId of org.apache .maven .plugins and an artifactId of maven-compiler-plugin. The version is 3.8 .1, and the configuration is where we specify that we want Java 11 and a target of Java 11, so the source Java 11 and the target Java 11. If you don't do this the first time you go to run your application, it will come back and tell you that it doesn't support Java 5. Now one thing that I think IntelliJ doesn't do a very good job of is warning you of that. Some of the other IDs will tell you that this is compiling it down to a Java 5 version, and that's because IntelliJ will compile code as you go using their internal IDE compiler and not the Maven compiler. So they're going through and checking code as you go, and it's compiling it to Java 11, but when you go to run the application, it will run it with what the default is in Maven, which, by default, is Java 5. So this is how you override to tell it to use Java 11. Now that we have our Maven

## Demo: Sample App Add Model

project structure in place, we can go ahead and get started adding Java code to our application. To do this, it's quite easy. All we want to do is navigate down to our source, main, java folder that Maven created for us when we created our project, and right-click and select New, Java Class. And at first use, you may not be aware that you can add a package in here at the time of creating your class. You do that by fully qualifying the name of the package structure you want. So I'm going to type in here com.pluralsight .model .Speaker, and Speaker is a capital S signifying the class name. When I hit Enter, it'll go ahead and create this object structure for me, and if you look at your project navigator on the left, you can see that it created that folder structure modeling the package structure and the object class name correctly, and it's also indicated up here on line 1 of our code with package com.pluralsight .model. Now we can enter in a couple of member variables in here. I'm just going to do a private String firstName and a private String lastName. And once I have those in here, I'm going to give myself a little extra space and right-click and say Generate, Getter and Setter, select both of them, and hit OK. Once I save this, it will compile it into our application, and that's really it. That's all you need to do to add the model object to your code. Quite simple. Again, navigate to source, main, java, right-click, fully qualify the package name and the class name. It'll create our speaker object for us, and then we'll enter our member variables and generate our getters and setters. Now that this is in place, let's go ahead and add our repository object to our class structure as well. The next piece of the

## Demo: Sample App Add Repository

application that we're going to add, and it's arguably the most complex piece of the small app that we're going to do, is the repository tier. I'm going to exit out of fullscreen mode here, and you want to navigate down to your source, main, java directory. Right-click and say New, Java Class. And just like before, we're going to enter in our package structure here of com.pluralsight, but this time we're going to do a repository.HibernateSpeakerRepositoryImpl, and click Enter, and that should've created another package. As you can see over here on the left, I now have the model and repository package, and then the class would've been named HibernateSpeakerRepositoyImpl, so make sure yours looks like mine does here. And you may argue that I don't like having the implementation in the class name, and I actually agree with you. The implementation in the class name can sometimes be thought of as a code smell, but the beauty of Spring is is that we code to an interface, so when this is all done, we're going to have a speaker repository interface, and that's all our app cares about. We can swap out these back implementations based upon what stage of development we're in. In fact, this is a good practice to do if I have a database team that I'm waiting for tables or views or those types of things for. I can create a speaker repository stub and extract my interface out against that, and my contract will stay the same. So let's go ahead and add a method inside of here. We're going to do a public List of Speakers, and we want to name this method findAll. And inside of here, we just want to create a basic list and return some stuff from that, so we'll say that we have our Speaker, and this is going to just be speakers = new ArrayList. And inside of here, we want to create an instance of a speaker, so we'll say Speaker speaker = new Speaker. And I'm going to just use my name. Go ahead and customize it to yours if you want, say speaker.setFirstName, and I'll put Bryan inside of here, and speaker.setLastName. And here we have our basic speaker object, and we want to add that to our speakers list, so we'll say speakers.add (speaker), and then, lastly, return our speakers, and save that. There are a couple ways we can extract an interface or create an interface for this. One is I could navigate over to my source, main, java directory again and go into my com.pluralsight repository package and create a new interface, but the easier approach is to just right-click inside of my class and say Refactor and Extract Interface. It's going to bring up a dialog, and it's going to ask me what I want for the class to extract from and the interface name. I'm going to change this to SpeakerRepository. And the package is correct and the target is correct, and I want to choose the findAll():List Speaker members to make an interface force, so it's going to extract those method names out into the interface for me. I'm going to click Refactor, and if this is the first time that you've done this, it will show you that it's made changes, and you say, Yes, I want to proceed. And we haven't used this anywhere else, so we're good to proceed with that. And we can open this up and see our SpeakerRepository with our findAll method inside of here. So it's just gone ahead and changed our implements SpeakerRepository signature on our class name and created the interface for it. Now when we're utilizing this inside of our code, we can just go ahead and reference it by SpeakerRepository rather than that implementation with hibernate in the name.

## Demo: Sample App Add Service

The next piece we're going to add to our application is the service tier. Now we're not talking about web services, but more so services as in the business logic tier. So similar to the repository, we're going to go ahead and click on source, main, java. Right-click and say New, Class and enter in our package of com.pluralsight. And this time, we're going to add service on the end of it. And for a class name, we're going to say SpeakerServiceImpl, and click Enter. Verify that you have the service package on the left or at the top of your code and that the name is correct. And then inside of here, we're going to do a couple of things. We're going to add a finder method as well in here and say public List Speakers, and it's going to be a very similar signature to what we did for the finder in our repository, but this would enable us to do other things such as swap out our repository tier and that type of stuff, so you may be looking at this going, this seems like a lot of overkill for a basic example, which it is. And we want to do our findAll here, and we're going to utilize the repository that we just created, but we haven't imported that yet, so for now, we're going to hard code this, and this is part of the problem that we're going to solve with Spring. So we're going to say private, and we're going to do SpeakerRepository repository = new HibernateSpeakerRepositoryImpl. Now this is what your code would like without Spring in it. We haven't added Spring to our application yet, so this is pretty accurate, and we would say return repository.findAll. Now just like we had with the repository, we don't have an interface for our service yet either. So let's do the same thing and say right-click, Refactor, Extract, Interface, and we want to call this the SpeakerService, and click the Member findAll, and Refactor, and it will do the same thing. And you can uncheck these if you don't like the dialogs asking you to make sure that you're not changing this elsewhere. We say OK, and we should see our interface here of SpeakerService. Now we can move on to creating a simple, main method that will test out all the pieces of the application that we have just written to be used together.

## Demo: Sample App Run Application

We now have all the pieces of our application built to where we can wire it up, so let's go ahead and right-click on source, main, java, and say New, Java Class. And I am going to use the default package for this. I just want to enter in here Application and click Enter. And we're going to create a public static void main in this file and pass in our String args, our basic Java signature for a main method. To demonstrate this, we're going to wire up our

SpeakerService. Just call it an instance of service = new SpeakerServiceImpl, and we will just do a System.out .println, a basic happy path testing if you've never heard that term before. We're going to just test that our application works exactly how we think it should, and we'll say service.findAll .get (0) .getFirstName now. In my instance, I used my name, so this should, when we run it, print out Bryan. I'm going to go ahead and right-click on main and say Run Application.main. You'll see it compile here in our build window, and you'll see that it printed out Bryan down below here. Now if you get an error here saying that it was compiled with Java 11, but ran with Java 5, and that's not supported, it's because you need to go back into the POM and add that build section that we talked about earlier. I've seen a lot of people miss that step, so make sure you've added that in there. Let's walk through everything that we did here really quick and what our application did. So we create an instance of SpeakerService, it goes to our SpeakerServiceImpl, so our interface is going to our implementation. This is going to load our repository tier, or, in this instance, our fake Hibernate instance, and go through our SpeakerRepository to that RepositoryImpl where it would simulate us calling a database and returning a list of speakers. In this instance, we had Bryan and added that to our list and returned it out. Let's go see how we would now configure all of that using Spring.

## Configuration

Spring is all about removing configuration code from your application. Why is configuration code such a bad thing in your app? Well, it makes things brittle, and by brittle, I mean hard to move to different environments. You maybe haven't experienced this yet in your career, but have you ever had to recompile your code because you were deploying it to a different server environment? What about testing? One of the things I hear as I'm coaching people through agile development is that, well, we don't do unit tests because it's really hard to test our code. We have really complex code. And I'll tell you right now that it's not the complexity of the code that makes it hard to test, it's the way the code was written. It's just a perfect example if you opt to not test code because it was too hard. It's not the testing that's hard, it's the configuration of that that's hard. Maybe it has a reference to JNDI or a database or some other things like that. More often than not, there is configuration code or code that doesn't have to deal with the normal flow of business logic that muddies up what you're trying to accomplish in your application. We've already developed part of the application without Spring, and now I'll show you how to configure that using the Java configuration approach, then we'll move onto annotations, and, lastly, we'll briefly show the XML configuration method. Let's examine some of those pain points a little closer, though, before we dive in.

## Demo: Pain Points Walkthrough

So to specifically call out some of the pain points we're going to address, notice here in this code on line 7, I have the keyword new and I'm referencing the SpeakerServiceImpl. So already, my application knows and has a hard-coded reference to this SpeakerServiceImpl object. These are the types of things that we're going to use Spring to eliminate from our code. I could create some instance of a factory or an abstract factory to go ahead and load this code in there, but Spring already does all this for us. Likewise, if you go into our SpeakerServiceImpl, you'll notice here on line 11, again, we have the keyword new referencing this HibernateSpeakerRepositoryImpl. If I ever want to change this, even though I have the interface and I've coded my application to the interface, I have to rebuild my entire application based off of this configuration. Also, if you go into our HibernateSpeakerRepository, and I alluded to this in our example, I have code inside of here that stubbed out until we have our database instance ready to go. There's no reason why we couldn't just use a stub here and use Spring to load into our ServiceImpl, that stubbed out class, that we could then swap out when the actual configuration for our database is done, and our application doesn't need to change. We don't need to go recompile code or introduce other changes into our source code, rather just change configuration. Let's look at how we can now bring Spring into our application to help us with those pain points.

## Spring Download

One of the reasons I originally updated this course was because Spring quit offering a direct download for the compile jars. Sure there were ways to obtain it, but nothing as simple as just clicking a link on their project page. With some of my other courses, people had asked that it just focus on one technology and not use supporting tools such as Maven. Well, like it or not, Spring wants you to download their tools using Maven. It will be simple for this course though and candidly any project of moderate complexity should be using Maven or Gradle anyway to manage dependencies. The Maven repo has the source, Javadocs, and the binaries all available for download. And one of the main reason Spring wants you to use Maven is because of their transitive dependencies that are required to run the projects. Spring Boot could be used to set up your application as well, but it's much too complex and a black box for trying to show you just the basic fundamentals of what Spring needs to run in order how to set it up. We will set up a Spring Boot application later though and compare the differences, but let's get our app configured and use Maven to download those dependencies and wire up our app.

## Demo: Spring Download Maven

Getting Spring inside of our application is actually quite easy. All we have to do is open up our POM, and I'm going to give myself some whitespace in between our version from our POM to our build section and add a dependency section. Inside of here, I'm going to add a dependency, and the group ID is org.springframework, and the artifactId is spring-context. From here, all we have to do is add a version, and you can look on their website to see what the current version is. At the time of doing this course, it was 5.2 .0 .RELEASE, and save that. And you can tell that you've done it right inside of your application if you look at your external libraries and see that it's imported all of the Spring libraries with their transitive dependencies. So you can see there's the spring-context 5.2 .0 that we asked for directly to be inserted into our application, but there's the transitive dependencies of Spring AOP, Spring beans, Spring core, Expression, and JCL that it pulled in as transitive dependencies. Now if that doesn't make sense to you, you may want to go refresh yourself on the Maven course like I mentioned earlier, but this should get you up and running for Spring inside of your application. That's really all we had to do to download it and get it into our app.

## Summary

Let's recap quickly what we learned in this module. We covered quite a few things, and it was a fairly large module, and it set up for us all of the upcoming modules that we're going to go through. We discussed why we'd want to use Spring and what it's buying us inside of our application. We went through and looked at the prerequisites that we had for our application and set those up for what we're going to do in future modules. We built a demo app and went through the details of that demo app and discussed the pain points inside of our demo app and why we want to use Spring to remove some of these critical pieces that are hard coded inside of our application. Then we went ahead and downloaded Spring and incorporated that inside of our application, and you saw how quickly and easily it was done using Maven. The next module that we're going to do is the Java configuration module. Now I want to caution you. A lot of people are real quick to want to skip the XML portion of the configuration of this course, and we have even restructured it so that you go the Java section first. I do strongly feel that the XML section, though, helps people understand the separation of concerns somewhat better than doing it in just plain Java, so let's go through Java first because that's the most popular, and then the XML section might help solidify any questions you may still have.

# Spring Configuration Using Java

## Java Configuration Introduction

In this module, we will walk through the configuration of your Spring app using just Java. We're going to take that sample application that we created in the previous module and wire it up using the Java annotations and Java configuration loader. Java configuration is the latest and most preferred method available now in Spring for wiring up your applications. It was introduced simply because some people don't like mixing XML and source code together for a configuration. We're going to take that sample project, wire it up with Java and a few basic annotations just to show you how this works. So let's talk about the key point of the Java configuration. Well, the first thing you'll notice about it is there's no XML. There's no applicationContext file. If you've done any Spring development in earlier versions, you'll have an XML file that contains all of your configuration. With that Java configuration, we have no [00:00:59] applicationContext.xml. We still have a context, it's just not configured in XML. Earlier Spring development got pushed back for having too much XML, and all of a sudden people were making jokes about being an XML developer rather than a Java developer. Later versions of Spring included namespaces to aid in development, but developers still wanted less or no XML, so enter Java configuration. Almost everything in Spring can now be configured with just pure Java configuration. This was not necessarily the case when we first started this course, but since this is an update, we've moved to showing Java configuration first because everything is available to be configured this way now.

## Demo: Copy Project

To get our project ready to copy for the next demo, I've gone ahead and closed any of the open files we had from the previous demo. To copy a project inside of IntelliJ, you actually don't do it inside of the IDE itself. You do it at the filesystem, so I'm going to switch over to my finder window, and you want to right-click on your folder, and just say Copy, and actually Paste. That's it. Just a simple copy/paste. You can do it from the command line, you can do it from your Windows Explorer window. I'm on a Mac. I can do it in my finder window. And you want to go ahead and change the name of this folder to whatever you'd like it to be. So I'm going to name mine conference-java, and then you want to switch back to your IDE. Once in my IDE, I want to go ahead and go to File, Open, and navigate down to that project, and select it, and you have a choice of New Window or This Window. I'm going to choose This Window. It actually replaces the existing contents we have with our new project, just so we're not confused about which one we're doing, any edits or anything like that. And then I've got to do a couple of things before we just get going and dive right in. First off, I want to make sure that the project was renamed, but it never actually renames the conference.iml. To rename that, we want to right-click on it and say Refactor, Rename, and change that to whatever our project name is, so mine was conference-java. If you did something else, you will want to name it that, obviously, and click Refactor. And then I'm going to open up my pom.xml and change the artifactId to conference-java. I'm going to do this just so that there isn't any conflicting Maven artifacts out there. If we did a Maven install or a publish, they wouldn't clash with one another. Now, we've got those changes made. I'm going to go ahead and navigate to my source, main, java, Application file. And you can open it up or you can right-click on it and run it from there. If I right-click on the main method and say Run, the first time we do this, it's going to pull up this Edit configuration dialog. The Edit configuration dialog wants us to change two specific things. The use of the classpath module, you'll see right now it says that no module is selected, and that's because we moved it and renamed it. So I want to choose conference-java, again, if you renamed yours something else, you'll want to choose that classpath module name. And you want to make sure you have the JRE installed. So, you have a default option if you've configured that inside of your IDE or you can choose a specific version. I'm going to go ahead and choose Java 11.0 .2 to make sure that that's the one that I always want this to run with, and I want to click Run. Then our application will build for a second and open up and run, and you should see the output from our previous demo the same as in our new copied project. So we should see that Bryan, or if you used your name there, output, and it finishes with an exit code of 0. If you get anything but this, you have something not copied correctly or configured right, so make sure you fix that before you move on. To begin configuring

## Demo: App Config

our application, we want to create a file where we can Bootstrap everything at. I'm going to close our build section here and navigate over to our source, main, java directory again, and right-click and do a New Java Class. This is a very simple class to start with. We're just going to call this AppConfig. Now it doesn't need to be named this. This is kind of a default standard or a norm. You could call it whatever you like, but we're going to choose AppConfig just to stay consistent with what we're doing, and click Enter. This is it. This is all we need to start Bootstrapping our application. If you've used any other versions of Spring where you did XML configuration, this would be the start of your application context, so this is where you start configuring the context of your application. All of your configuration starts at this point and trickles throughout your application here. So there's nothing special about it. You'll notice it's a default public class. There's no methods. It's a plain POJO. We'll go ahead and start adding annotation to this, but before we do, let's go ahead and talk about what those configured annotations are.

## Demo: @Configuration

To start the configuration of our application, we're going to use an @ configuration annotation. The Java files that have the configuration annotation replace any XML files that we could've used historically. Configuration is a class-level annotation and looks something like the annotation in the snippet above. In that AppConfig file that we just created in our demo, we now put an @Configuration annotation at the top of that class. Methods used in conjunction with the bean annotation are used to get instances of Spring beans. Now, we're throwing a couple of topics at you here really quickly, and there's a short, but steep learning curve. We're going to explain what a bean is and what a configuration file is as we go. We just have to show you these two concepts as we dive into it. We add an @ method level annotation for Bean, and it looks similar to this where we're saying that what this method returns is an instance of a bean and that the bean is now registered inside of Spring and available for us to use inside of our Spring application. Classes and method names can be anything. Spring really doesn't care. It doesn't have to be named getCustomerRepository or anything specific. We can call our methods _____ foo just as long as when it goes to create these beans, we have them marked as a bean and in a configuration file. Let's begin by starting to add some of this configuration to that AppConfig file that we just created. I've switched back over to our IDE. I want to go ahead at the class level and say @Configuration. And you can see that as we're typing it, it wants to go ahead and suggest what it should be. We want to choose that second one, the configuration, not configurable, configuration, and choose it, and it will automatically import that annotation for us. Now this signifies that this file is to be used for configuration purposes. To add a bean, a Spring component down here, we can start off by just creating a method, so I want to call this public, and I want to return an instance of the SpeakerService. And I'm going to just call this getSpeakerService. I'm going to use a no arguments method signature. And I want to return inside of here just a new instance of that SpeakerServiceImpl. So I'm going to say return new SpeakerServiceImpl. And it's even suggesting to return the implementation of it. So I can save that, and now we have our method signature and structure complete, but we still need to tell it that this is a bean. So we want to say @Bean. And we can optionally, we don't have to, optionally give this a name. And I'd like to, and I want to call this the speakerService, and I'm going to do this camel case, just like we would a variable name. This is it. We've literally Spring enabled our application. We've told it to configure this as a Spring app, and we've created a bean, our speakerService, and we're going to show you how we transform that out of the application to be in this registry or this configuration that's now a bean.

## Setter Injection

Setter injection using the Java configuration approach is really as simple as a method call. A lot of the mystery of injection just goes away. A lot of developers I talk to are concerned with the black box fill of dependency injection or inversion of control. And with the Java configuration approach, it's more visible as to what's actually going on. With XML, there's a lot of wondering what's wired up and what's calling in, who and how this autowiring is all taking place, and that just kind of goes away with the Java configuration. Setter injection is simply a matter of calling a setter on a bean. We're going to define a bean using the bean annotation like we did in the previous demo, and you can see here that we've got our getCustomerService method that returns a bean of type customerService or a bean named customerService, and notice, though, that we have a setter injection in here. As we're building our beans, we're going to call this setCustomerRepository method on our customerService instance. To do that, we need to have a method called getCustomerRepository in our configuration that returns the customerRepository bean. This is going to wire the CustomerRepository inside of our CustomerService. Let me say that again. We're not just going to call an instance we create. We're going to call and get an instance of the bean from the Spring configuration file. So when we define a bean such as our CustomerRepository here that returns an instance of CustomerRepository, you can see that those methods now line up. I'm going to call getCustomerRepository that's registered as a Spring bean, and that's going to return an instance for our setCustomerRepository instance on our service instance. Take note, Spring is still doing a lot of magic behind the scenes when a bean is registered such as these beans are all, by default, a singleton and will only execute the method the first time it's called. That's very key point because if I didn't have this set up this way, it would create a new bean every time we called this. Since it is in Spring, it's going to register this as a singleton and only return one instance. Let's implement this in our code now.

## Demo: Setter Injection

To add setter injection to our project, we actually have to change a couple of things that we have hard coded inside of our previous demo. That's the whole reason we're actually using Spring, so let's go ahead and open up our service, and specifically our SpeakerServiceImpl, and see how we have this new HibernateRepositoryImpl in here? Let's go ahead and get rid of that hard- coded instance, and we want to create a setter in here where we actually wire up this configuration. So we're going to right-click and say Generate, and we want to do Setter, and specifically the SpeakerRepository, and click OK. And now we've converted our SpeakerServiceImpl to be injected rather than have hard-coded configured instances in here. We can save this. Then we want to go back to our AppConfig. Now inside of our AppConfig, we can do a couple things here. We want to start off by creating a new bean for our SpeakerRepository, so we want to say Bean. I want to give this a name = speakerRepository. And if you are wondering why I am kind of fanatic about the names of our beans is because when we go to do autowiring, which is coming up here very quickly, we're going to use it to autowire by name to our variable name, so these match up and it automatically happens for us. Let's go ahead and finish out our method signature here, so we'll say SpeakerRepository, and we'll do getSpeakerRepository. [00:14: 37.473 ] Take any no arguments method parameter as well here. And inside of here, we want to return our new HibernateRepositoryImpl. And now I want to go back up to our SpeakerService that we created earlier and reconfigure this a little bit. So I'm going to just delete this line, and we want to do SpeakerServiceImpl, not the interface. We want to do the actual implementation. We call this service = new SpeakerServiceImpl. And inside of here, we want to actually call that getSpeakerRepository bean that we just created, so we'll say service.setRepository, and we can call the getter inside of here where we do the getSpeakerRepository. And we've now configured our service with that repository, and now we can return our service that we just created. So now we changed our repository to be injected through setter injection into our SpeakerServiceImpl. Now a couple of questions before we proceed just to clear anything up that you may be a little bit confused on right now. Why didn't we just create a new instance of the HibernateSpeakerRepositoryImpl like we did on line 20 on line 14? Why did we call this getSpeakerRepository? Because SpeakerRepository is now created as a bean, and I mentioned this in the previous slides, but to kind of reiterate that pointer, make it really stick, the bean is a singleton. Only 1 of them will get created. We can call this getSpeakerService method 50 times now, and only 1 of the SpeakerRepositoryImpls will get created. So it's implemented as a singleton, and that's a good thing. That sometimes confuses or concerns people to start with. This is actually a good thing for us. So now we've got this configured to use beans. They're singletons. It's all wired up. It's using setter injection. Let's go to our application now and switch our application to use Spring because right now we have all this configuration stuff going on, but we aren't actually doing anything with it yet. So let's go inside of our main method, and we want to say ApplicationContext, and we'll call this instance appContext = new AnnotationConfigApplicationContext. Let me make this fullscreen so you can see a little bit more of what's going on here. And inside of here, we want to pass in the AppConfig.class. Now this is our AppConfig that we just created, this file up here where we configured our beans and it's annotated with configuration on line 8. This file is what we're passing into our AnnotationConfigApplicationContext. What this is is this is loading Spring and it's loading the configuration files that we have out there into our ApplicationContext. So when line 9 is ran, it's going to go out and create a basic registry with 2 beans in it, the speakerService bean and the speakerRepository bean. As we just changed on line 13 and 14, it now injects the speakerRepository bean into the speakerService bean, and then we'll return that back to our application when we call it. Now to change that, I'm going to change one more line here, and I wanted to comment out that old one so that you can see it and kind of compare the two, so we'll call this SpeakerService service = appContext.getBean. And we want to use the version that does type casting, so we want to go in here and, first of all, we're going to call it by name and say that we want the speakerService and pass in the class instance of SpeakerService.class. And save that instance, and we're ready to run our application. We have Spring enabled it. We have gone ahead and done the setter injection of our speakerRepository and our speakerService. Let's run this now. When we fire this up, it'll go ahead and compile it, and you'll see that it actually returns our instance just like we had before. I've commented out line 11 so you can see what we had before that was hard coded. Now nothing inside of our application is actually hard coded. It's done through our configuration instances. Just to recap that, we added the configuration inside of our AppConfig that now signified a speakerRepository bean that wasn't there before, and it is setter injected into our SpeakerService. Our SpeakerService, we got rid of the hard- coded configuration inside of here and just added a setter for it, and then we converted our application to use Spring. And from this point forward, we can now inject all sorts of beans and pieces in our application wherever we would like to do that and not have it hard coded. Let's look at what this would look like using constructor injection now.

## Constructor Injection

Constructor injection is just like setter injection. We go through and create our bean instance just like we had before. Instead of calling the setter, we call the defined constructor. Our bean that we had written before where we would call the getCustomerRepository instance would now be used in the constructor of that bean. Very simple. And, again, like setter injection, some of the mystery of what Spring is doing behind the scenes is removed using the Java configuration because it looks just like the Java code that you would do normally inside of an application. The one difference being that now we have this stored in the container, and we're not passing objects around. We can pull them from the framework using the bean, the get bean aliases, with these bean names that we have defined in our objects. Moving back to our IDE, it's really easy to now switch this over to using constructor injection. I'm going to open up our SpeakerServiceImpl and add a constructor inside of here. So I'm going to go ahead and say public SpeakerServiceImpl and pass in a constructor that takes an instance of a SpeakerRepository and give that the camel case naming convention of that instance. And I'm going to assign this to repository = speakerRepository, and save this. Now we have our instance stored inside of here that is going to be injected using setter injection. We can go over to our AppConfig, and you'll notice that on line 13, we broke our configuration down. That's fine because we're going to go ahead and fix that to use that correct injection. I'm going to comment out line 14, and I'm going to comment it out, not delete it so that you can see it for a reference compared to what we did with the setter injection. I want to just go ahead and call getSpeakerRepository right here and save it. So now as we create this bean, we're going to call the getSpeakerRepository bean and inject that in through constructor injection into our SpeakerServiceImpl. Let's go ahead and run our application again. I'm going to switch over to Application and right-click on our main and say Run. And you should see it now export out Bryan, which it does. You may be thinking this is magic and it didn't run. You're more than welcome to go ahead and add some system.outprintlns or debug and step through it, but it's going through and using this constructor injection. One of the biggest mistakes I see people make is in this code for our injection, they start trying to do crafty things and go ahead and create some variables and hang onto some stuff here. That's what Spring does for you. Whenever I see new people doing stuff with Spring, they try to be smarter than the framework and create some instances here and pass some stuff in. One of the guys I recently taught about Spring was trying to create his own singleton when Spring did these singleton services for you. Don't do anything magical. Just wire up the beans how you would use them and let the framework do that for you.

## Summary

To recap what we did in this module, we learned that instead of an application context that we could configure our code using the Java AppConfig file. We use an @Configuration annotation to define files that contain configuration code. We can have multiple files that are containing configuration information. They just need to have that annotation at the top, and then when we wire up our application wherever our main method's contained at, pass in those config classes. We can just define a bean using the bean annotation. After setting up the configuration of the beans, we walked through setter and constructor injection using these configuration types. I like this approach a lot since, as we mentioned before, it seems to remove some of the mystery of what's going on behind the scenes in Spring. It's literally just writing Java code and wiring those beans up in that configuration up, which can sometimes seem like a mystery with some of the other concepts. We started to dig into some of the more wiring concepts that are a little bit advanced with Spring, but let's dive deeper into those because we really just scratched the surface on those, and let's go into autowiring deeper in this next module and focus on autowiring and component scanning inside of Spring.

## Spring Scopes and Autowiring

### Bean Scope Introduction

Now that we've seen some of the basic Spring configuration and overall just how to create and implement beans inside of our application, let's go a little deeper and look at Spring scopes and autowiring, specifically scopes of beans inside of our application. They're a very important part of the bean lifecycle and will show us how to use different things inside of our application and implement behaviors regarding beans. The scopes inside of our application usually are associated with patterns, but they're not really the same thing. They go hand in hand, but a scope does not equal a pattern. Spring implements a lot of patterns for you, which is good because it helps you avoid a lot of the common mistakes and pitfalls made when implementing patterns on your own. And if you aren't familiar with design patterns, I really recommend watching my design pattern series on the Java design patterns of behavioral, creational, and structural patterns here on Pluralsight. Let's dive a little deeper on the scopes and talk about which scopes are available to your application and which scenarios they tie into.

### Scopes

There are five scopes available inside of Spring for us to configure a bean inside of our application. Valid in any configuration is a singleton, which is actually the default. You've most likely heard of a singleton before, but we're going to explain that a little bit more in detail and what that means in regards to a Spring application in our next demo. The other scope, though, is a prototype scope. Prototype is actually a new bean per request. You typically do want the singleton over the prototype. The other configurations that are available for beans that are only valid in web-aware Spring projects are request, session, and global. So if you're implementing Spring MVC or even doing a single-page model application to where you're doing microservices to your front end, those scopes are only available in that context. Let's look a little bit more at the singleton pattern.

### Singleton Java Config

The singleton design pattern restricts the instantiation of a class to just one object. A singleton is the default bean scope inside of Spring. So if you don't give it a scope, it will automatically be assigned the default scope of singleton, which means that there is one instance per Spring container or context. A lot of people say that means one instance per JVM, which is technically true, but you could possibly have more than one Spring container spun up in your JVM. So, really, the correct term is one instance per Spring container or application context. The configuration for adding scope using Java configuration is quite simple. We simply add the @Scope annotation to our code as you can see here in our code sample. And in the previous release of this course, we weren't using Maven, so you were required to add the AOP jar separately. Since we are using Maven, this is just a transitive dependency, and it's already available in our project. Let's add that scope to our project now and show you how you can configure that for your beans. To add scope to our already configured application, I've gone ahead and opened up our AppConfig.java file. I'm going to go below our bean definition for our speakerService and add an @Scope annotation. Now by default, I did mention earlier that it is a singleton, and that's implied, but we can explicitly set it, and there's a couple ways we can do this. First off, we want to go in here and type value, and we can type inside of here singleton. Now I do not like using my own typed out strings for these values because I could easily forget that n off the end of there so it's a singleto not a singleton. So we can use some strings that they have already set up for us inside of Spring by saying BeanDefinition.SCOPE_SINGLETON. You can tell by this context-sensitive help tip that it's just a sting equal to singleton, but that's what we want and we don't have to define our own. So a lot of people go ahead and create their own utils. We'll just go ahead and select value= BeanDefinition.SCOPE_SINGLETON, and we can save this. Now to test this, let's go back to our application, and, right here, we have this SpeakerService service = appContext.getBean. So we have an instance of this bean where it's going to go ahead and tell us what object we have. I'm going to add another System.out .println here, and this is a very simple way to just see what's going on behind the scenes here. And I want to actually just print out service, and it's going to be the object address for service, and I want to grab another instance of that bean. So I'm going to copy this line here, and go down below, and we'll call this service2, and I'm going to copy that System.out .println again. Make sure you change the variable to be the second one. And this should print out these two different object address instances for us. So, let's go ahead and run this. I'm going to right-click on our main method and say Run Application.main. And when this runs, you'll notice that it prints out the exact same object address instance. So down here in our console output, you can see that com.pluralsight .service .SpeakerServiceImpl, and your object address will be different just because it's unique every time you run this through the JVM. Then it prints out my name, and it prints out that same object address instance again. So our singleton is in fact a singleton. Each time we ask for an instance of that bean from the application context, it's returning the same bean for us. Now, this is the behavior we want because we shouldn't be storing any state that is unique to that user inside that object. You can store state unique to that application in there, but you don't want to store state unique to that user inside that object. So, this is working as we planned. Let's look at what it means to be a prototype now.

## Prototype Java Config

The prototype design pattern guarantees a unique instance per request, and, thus, the scope inside of a Spring container mimics that design pattern. Each time you request a bean from the container, you're guaranteed a unique instance. It is essentially the opposite of a singleton. The configuration of a prototype is nearly identical to a singleton, so let's look at a small code snippet of how that's configured and then integrate that into our code. The configuration is very simple. You just set that to a string of type prototype. We'll go ahead and use the bean definition just like we did for the singleton, and everything's already configured in our application to show us that this sample is going to work. Let's take a look at that code now. I've opened up the AppConfig back where we had it before with the BeanDefinition.SCOPE_SINGLETON, and I want to replace that with SCOPE_PROTOTYPE. We've already configured our Application.java to show the object address to see that it's unique per request. So let's now run this and see if our object address has changed per request. When we run this, you'll notice now there is a unique instance every time we ask for that bean from the container. So our object address before was 73700b80. Yours will be different because of your instance, printed out the name, and then it has the new object address of

10d307f1. Again, yours will be different, but now you can see that since changing that to a prototype instance, it is now giving us a unique bean per request, now, per request of the bean from the context. If you hang onto that bean and do stuff with it for the next 10 minutes, it will still be the same bean, but every time we ask for a new one for the container, it's going to give us a unique one back. Just so we don't mess up any future demos, let's go ahead and change that back to SINGLETON. I'm going to select SCOPE_SINGLETON, save that, switch back to our application, run it again and make sure that it's ran, and everything look correct. We've got the same object address as before and after.

## Web Scopes

Web scopes are beyond what we're going to cover in this course since it's just a basic Spring fundamentals course. They are covered more in the Introduction to Spring MVC course that's available here on Pluralsight, and that's simply because we have to set up an entire web application for you to see how they interact with an object. The three other scopes that are available are the request scope, which returns a bean per HTTP request, which sounds a lot like prototype except it's for the lifecycle of a bean request, which is fairly short, but longer than prototype where it's one instance per every time I ask the container for a bean. The session just returns a single bean per HTTP session, and that will live as long as that user session is alive, so 10 minutes, 20 minutes, 30 minutes, however long they're alive on that website a bean of scope session will stick around. And then there is globalSession, which will return a single bean per application, so once I access it, it's alive for the duration of that application, not just my visit to that application. You could think of it as singleton, but it's really the entire life of that application on the server until it gets undeployed or the server gets rebooted.

## Autowired

Autowired. Autowiring is where I would say that most people start to think that there is some magic that's taking place inside of Spring. Autowiring is a great technique used to reduce the wiring up and configuration of code. If you've ever heard the term convention over configuration, this is it. To autowire our applications using the Java configuration, we just simply need to add a component scan to our configuration file with an annotation that looks something like this. The text inside of the annotation, com.pluralsight, just says that this is where I should begin looking for autowired annotations. To use autowiring, you just mark whatever bean you want as autowired. You can choose by name, and that uses the @Bean name or by type, and that uses the instance type. One thing I like about the Java configuration over XML configuration for autowiring is that using Java, I can mix pieces that I want and it feels more natural, so I can have a bean that I defined as autowired with another bean into it, and it makes more sense where those beans are coming from. We'll show you how that configuration plays out in this next demo.

## Demo: Autowired

We need to do a handful of things to get our application set up to use autowiring correctly, and that has more to do with just how we're doing the demos than you need to do this in every application you've done. We're trying to show a bunch of different techniques, and this makes it a little bit more interesting showing some of those. To start with, let's open up our SpeakerServiceImpl, and inside of here, we had gotten rid of our default no args constructor in one of the previous demos. Let's add that back in because we're going to need that for our setter injection demo that we're going to use autowiring on. And I want to add a couple of log statements just using simple System.out .println, so we'll say System.out .println, and inside of here, we'll say that we are in the SpeakerServiceImpl no args constructor, and I'm going to go ahead and copy this because we're going to add some of these to a few different methods here. Let's go to the next one, and we'll say that this is the repository constructor, SpeakerServiceImpl repository constructor. And the last log statement that we'll add is one in the setter for the SpeakerServiceImpl, and we'll say that we are in the setter, and just save that, now, pretty straightforward. All we did was add a no arguments constructor and the SpeakerServiceImpl log statements and setter log statements. Let's go back to our AppConfig. As of right now, we are still using the constructor injection that we've wired up. It's really actually not injecting. It is configuring the bean, but we are just adding that in there. Let's comment this out, and I'm going to leave these in here just so you can see as we go through each instance of these what the configuration was from the previous to the next and say new SpeakerServiceImpl. And if we ran this right now, we are not wiring in the repository anywhere. Since we want to use setter injection to begin with, let's go back to our SpeakerServiceImpl and go to that setRepository and say @Autowired. When I save this, it's going to automatically inject that SpeakerRepository bean into this setter. So when we run this code, we should see the no arguments constructor get ran, and then we should see the log statement for this SpeakerServiceImpl setter also get ran. Let's go back to our application code and run this. And you'll see that we were in the no args constructor, and then the setter was called, and then our other log statements from the other stuff we were doing with our singleton scopes are dumped out. So, very simply, our code was changed to do the no arguments constructor and setter injection inside of here, and then we went to our SpeakerServiceImpl, added that no args constructor in here, and just specified that we wanted this method to be autowired and injected this way. What if we want to do constructor injection though? We were using that before, and we have our constructor still in here. I'm going to show you how to do this where your beans are fully autowired, not just individual methods because it shows that demonstration, that example a little bit more concrete. You'll also notice that we talked about the component scanner before, but we haven't configured that yet, and that's because we haven't configured our beans to be fully autowired, just on individual methods. Let's talk about a technique called stereotype annotations first so that we can fully autowire our beans up.

## Stereotype Annotations

To fully autowire our applications and not just do a hybrid like we did in the previous demo, we need to talk about stereotype annotations. Our last example we had beans still configured in the AppConfig file and then had some things autowired. To remove the need for those bean definitions in the AppConfig, let's discuss the stereotypes. The first one is @Component. It is actually the same thing as @Bean. I referenced component here just so that if you saw it in another example, you wouldn't wonder what it was and why we hadn't covered it. We'd already used @Bean, and they are basically the same thing. The next one is @Repository, and as you might guess, it's used to denote a class that's being used as a repository object. Nothing else changes with these. [00:17: 27.398 ] They're just nice to label beans as such. You could technically just label everything as a bean or a component, but you could use a filter to look for specific types of annotations. That's beyond the scope of what we're going to talk about here. The last one is @Service. This is often confusing to people because a service does not mean a web service or a microservice. It rather means where you would put your business logic. There is actually a fourth annotation that we will cover in the Spring MVC intro course named controller. It's out of the scope of what we're covering here, so I didn't use any real estate to show it, but this is where you would create web or microservices for your application. Now that we've seen our stereotype annotations, let's go ahead and finish wiring up our application the whole way. In our AppConfig, we want to add our component scanner. So we're going to import @ComponentScan. Inside of here, we want to give it the string syntax for an array of the package structures that we want to scan, so we want to do com.pluralsight inside of here. And this just tells it that this is where we want you to start scanning for beans to autowire. Now the other thing we have inside of here is we still have this hybrid approach. On line 15, we have a bean defined. On line 24, we have bean defined. To fully autowire our application, we can get rid of these bean definitions inside of here. Now one thing I didn't mention to you when discussing the stereotype annotations is that the bean annotation is only applicable at a method level. So we've got a method defined on line 17 and a method defined on line 25. I can use the @Bean annotation here. I cannot use the bean annotation at a class level. So to demonstrate this, I'm going to open up our SpeakerService, and if I come out here and type @Bean, you'll see that I get an error, and if you hover over the error, it'll actually tell you that that is not available for that specified type, where I can come in here and say that we want a component, and that will work. That is applicable, but since we're in a service, I'm going to use the service stereotype annotation, and I'm going to go ahead and name this here. So we'll give us our speakerService name that we had before, and save this. Now the other thing I want to do is I want to autowire this bean, and right now it's set up for setter injection. Let's go ahead and leave it that way. I am going to go back to my AppConfig, and now I can actually comment out all of this code for this speakerService bean. We're actually going to do the same thing for that speakerRepository here in just a second, but let's not get ahead of ourselves just yet. I'll open up my HibernateSpeakerRepository, and I will note that we are doing this on the implementation, not on the interface. Don't make that mistake and open up the interface and try to do it on there because it won't work. At the top of our HibernateSpeakerRepositoryImpl, let's do @Repository, and we'll give this a name as well, and this will be our speakerRepository, and save that. So now this bean is set up as a repository using the repository stereotype annotation. Our speakerService is set up as a service with the name speakerService, and then our AppConfig, we can go ahead and finish commenting out all of this. And right now, speakerService is set up with that setter injection for the repository to go into it. Let's make sure we've done everything right here and go ahead and try and run our application. I want to open up our application, right-click on main, say Run, and you'll see that it called the no arguments constructor. The setter ran through our code that showed that we had those objects, printed out the name, and we had the same address. So we've actually kept that scope the same. Now, let's not move too fast here. Let's show you a couple things inside of here really quick. We just got rid of this bean definition that we had to go type all that code out for, wire up the whole thing. This @Scope, if we still want that scope, I can go back to anywhere we define a bean at the top of the code and say, well, let's throw Scope in here. So I can go in my app and still define Scope here if I want. By default, it's singleton, so there's no reason for me to do that, but I can define that at the class level. And we have our autowired setter injection taking place. If we want to change this to constructor injection, we can go ahead and move that up to the constructor and autowire it that way, and let's run our application again and see what it does. We go inside of our app here, go ahead and click Run, you'll see that we have now switched over in our log statement to using the repository constructor injection, and it's still autowired up. So I've effectively removed all of this configuration code inside of AppConfig by just having the component scanner and defining our beans the right way. This may not seem like a big thing to you except for I've worked in applications that have had hundreds of beans, 2, 3, 400 beans. If I have to go through and define 8 lines of code to define each bean and then troubleshoot that, you're talking with 300 beans, a couple hundred lines of code to get that set up for that application, and trying to stay on top of that and maintain that, this really saves you a lot of time and effort, and it's really actually quite nice in how it wires those things up. So it's really simple. You just define the stereotype. You can still define the scope if you want, and you can move around the autowiring from constructor to setter if you want to do it based off of which contract you want to enforce.

## Summary

In this module, we discussed the various scopes that are available inside the Spring container, namely singleton and prototype. We showed examples of how to configure both of those using the Java configuration. We then mentioned the web scopes that are available, but they are outside the scope of this module and of this course really because we have to set up an entire web application just to demonstrate them. Those three scopes are request, session, and globalSession. We then discussed autowiring. We showed an example using setter injection and constructor injection and showed that you can use the component scanner to wire those apps up or you can do a hybrid. Our first example we had with this was just a hybrid of some of the methods being autowired, but still having our beans defined in that AppConfig. If that makes you feel more comfortable in using it, use it that way, but I will say the more convenient way is to go with full autowiring using those stereotype annotations that we discussed. Let's look at what we can do with the XML configuration, and we're going to run through some smaller examples because it's very similar to Java now.

# Spring Configuration Using XML

## XML Config Introduction

Now that we've gone through the Java configuration for Spring, let's talk about configuring your application using XML. We're going to take that sample application that we built in the previous modules and wire it up with XML using the Spring Framework, but just so you don't think that we are done with Java configuration, we are going to compare and contrast the XML configuration with the Java configuration, and we're going to show a couple more techniques on how to autowire your app up. So why would you still use XML? XML configuration was the first method available in Spring, and although it has declined in popularity over the years, some things, in my opinion, are still just simpler with XML. There is a natural separation of concerns that organically happens when configuration code is removed and replaced in a separate file. We're going to copy that sample application that we created in the first module and create a file called an applicationContext to wire up our app. Our file doesn't need to be named the appContext, it's just a standard that's kind of become associated with Spring. We're going to go through and wire it up in XML to help you better understand some of those concepts and configurations that we showed with the Java configuration.

## Demo: Copy Project

Just like we did when we were configuring our Spring Java configured enable application, I went ahead and closed everything, and I actually opened up our original conference app just so can look at it and reference it, although, that really doesn't matter because we're going to copy this and open up the XML- configured app anyway. So I've closed all the files down, got it down to just the basic desktop, and I'm going to switch over to my workspace. Now I'm going to take our conference app, not our conference-java, but our conference app, Copy, and Paste that, and I'm going to call mine conference-xml, and enter that. I'm going to switch back to my IDE, and now I want to go to File, Open, and I want to navigate down to my workspace, and open up the conference-xml project. And I'm going to choose this window to replace the contents that we had there. And now we have our basic project structure up and ready to go. We'll go ahead and run it, making sure that everything is set up the right way. I want to navigate down to my source, main, java, Application, and right-click, and say Run Application.main. Let's make sure that everything still finishes and completes exactly how we expect it to, and it does. Now this, again, should be the basic project that didn't have any of your Java configuration in there. I can't stress that enough. We don't want to try and mix the Java configuration with the XML configuration. So our Application.java file should be just this hard-coded SpeakerService, creating a new SpeakerServiceImpl. It has the System.out .println in there for just the original find. Our model and service are all pretty basic, a repository, in fact, our SpeakerServiceImpl should have that hard-coded HibernateSpeakerRepositoryImpl in here. That's what we're going to do with the XML configuration now just like we did with the Java configuration. Now I can't stress enough, I am going to show you a couple techniques with XML and their equivalent in the Java configuration as we do this to help compare and contrast it. So if you think we're just going to cover XML, we're not. We're going to go through some Java configuration techniques as well to show you how they compare to one another and the power of one over the other.

## applicationContext.xml

The applicationContext really is the root of an application configured with Spring. And just like we had in our Java configured Spring app, the AppConfig is this same thing. They are synonymous. It doesn't have to be named the applicationContext.xml just like our AppConfig.java file doesn't need to be named that, but it's more of a loose standard. I've seen people abbreviate it appContext or App-Context, and this is just kind of a default that you see people do inside of their applications. By default, though, Spring will look for a file named applicationContext.xml without any extra configuration. A simple view of Spring is that it's just a HashMap of objects, and we define that HashMap inside of our applicationContext.xml or our AppConfig.java. The objects that we have inside of here are pretty much just name value pairs. Although it's not the intention of Spring, it can be used as a simple registry, and we can look up those beans out of our context. All of our XML configurations begin here, and for our sample, we're going to call ours applicationContext.xml. You can have other files that will reference this and look up and pull it in. It's a little bit more of an advanced topic, but we can import those things in there. There are some namespaces that the Spring developers have put together that help us in our configuration and validation of our files and really make things a lot easier. We'll look at adding a namespace to the top of our application, but in our example here, we're basically going to put an XML snippet at the top of our application context, and you'll know what it means in our bean namespace, and it helps us configure our files. Let's do that to our application now. Adding the applicationConfig.xml to our application is actually quite easy. You want to navigate down to source, main, but alongside of Java, there's another folder that was created when we created our Maven project called resources. And if you open yours up, there's nothing underneath yours. I have a notes.txt in here just so I don't have to type in a bunch of the XML configuration that we're going to do. This is one of the drawbacks to the Community Edition to IntelliJ. The ultimate paid edition actually has a dialog to help you create this XML file. The free version does not. So I want to right-click on source, main, resources and say New, File, and I want to name this applicationContext.xml, and click OK. And it will just give us a blank file. Now I've gone ahead and searched through the Spring documentation to copy this XML snippet that I'm about to paste in here in. You can do the same or you can pull this out of the resource files for the project, so if you navigate to our course description, there is a resource download section to where you can pull the files down if you don't want to go searching the internet for the Spring XML configuration namespaces, but here it is. So now we have our file. It's our beans namespace already defined in this, and this is the root namespace that you typically will use for a Spring configuration file. What this does as we're going through and doing help and context-sensitive help inside of our XML file, when we start doing a bean definition by typing a left angle bracket, you'll see that it pops up the helper context for the different things that we can put in this namespace. And that's all done by just including this XML schema definition at the top of our file. Let's go through and look at what the namespace is now by us and how that kind of helps out in us developing the XML configuration a little bit better.

## Bean Definition

Here is a snippet of just the beans namespace definition that I just copied to the top of our XML file. I believe that the misunderstanding of namespaces is one of the reasons that people are afraid of XML and using XML configuration. Spring has created this namespace that simply acts like a dictionary for the properties that we can use to create a bean inside of our application. Let's talk a little bit more about these properties. So our XML declaration allows us to define a bean in XML. This bean is called a CustomerServiceImpl bean, and it represents where we want to put our business logic inside of our application. There's a few other properties to find in this example as well that we're going to walk through more in detail, but you can see the pieces that we are creating here. The beans are essentially just classes. The XML configuration is composed of beans just like our AppConfig was as well, and they're just POJOs that we use inside of our application context. Defining beans can be thought of as replacing the keyword new. So whether you are in your application or you're saying something like your _____ CustomerService = CustomerServiceMy, CustomerService = new

ServiceImpl, wherever we're using that keyword new, that's something that should be thought about being put in your XML or Java configuration. Lastly, we always want to define the class, but use the interface, and I'll show you what I mean by that in our demo that's following this.

## Demo: Add Bean

Adding a bean inside of our applicationContext is very similar to adding it to the AppConfig that we had before. We want to do an open angle bracket and use the bean alias. And inside of here, it gives us a bunch of different properties that we can use for this. I'm going to start by choosing name as the first one, and the name that we want to do for this is the speakerRepository. The next thing that we want to do is add the class. And the class that we're going to do for this is com.pluralsight. And we want to specifically choose our repository. And we want this to be an instance of the HibernateSpeakerRepositoryImpl. Now in our code, we are going to reference this using the interface, but we'll create the bean with the implementation. I'll show you more what I mean by that here in a second. But this is actually all we needed to do to create our bean definition. Now we're not injecting anything in this, in fact, that's why I started with this bean is because if you remember, our SpeakerRepositoryImpl has no other dependencies in it. I'm going to open that up so you can take a look at it. It was just a very simple hard-coded one method list of speakers that gets returned from what would be a mock database, so there's nothing being injected in this yet. We're going to go through and look at setter injection a little bit and actually reference our Java configuration that we did to show you the two and come back and wire this up to be setter injected for the SpeakerService with this bean that we just created in the SpeakerRepository.

## Demo: Setter Injection

To set our application up for setter injection, we're going to have to define another bean inside of here just like we did for the SpeakerRepository, but before we do that, I want to open up our SpeakerServiceImpl. And inside of our SpeakerServiceImpl, we're going to get rid of this hard-coded HibernateSpeakerRepository reference just like we did for our Java configuration. Then I'm going to go down and give myself a little extra space and click Generate, Setter, choose SpeakerRepository, and click OK. Before I move on though, I do want to change the name of this from setRepository to setSpeakerRepository. The name isn't essential, but it does make it easier especially for autowiring by name, which we're going to show a demonstration of that later in this course. I'm going to click save. And I do like to clean up my imports. You can see up here on line 4 that we have an unused import. That's why it's highlighted in gray. I'm going to do Ctrl+Shift+O on my Mac and remove that import. You can also do Ctrl+Shift+O on a Windows machine rather than _____ Ctrl option, and it should clean up those imports for you. You can also just delete it manually. I'm going to save this and go back to my applicationContext, and let's create another bean inside of here. As I mentioned before, to do this, we just want to create a bean just like we did for the SpeakerRepository. So I'm going to give myself a little extra space there. Make sure you stay inside that closing bean element that I have on line 12 of my application. And we'll say bean and give it a name of speakerService. And notice I'm doing this as variable names exactly like we would do in the

Java configuration. And I want to define a class, and the class is going to be com.pluralsight .service .SpeakerServiceImpl. And instead of doing the forward slash and closing element, I just want to do a right angle bracket, and it'll give me that closing bean element separate. And the reason we want to do this is because we're going to create a property inside of here now. And the property name is going to be speakerRepository, and there's two options inside of here of reference or value. Since we're referring to another bean, we want to do reference. If we were just putting a value in there, say some string or some numerical value, we would use value. So we want to do a reference, and that reference is going to be to the speakerRepository bean. And then we can close that off with a forward slash, right angle bracket, and save this. Now our beans have been defined, and we can see that we have our SpeakerServiceImpl created with a property that we're setting from the reference to the bean speakerRepository that we've defined on line 7. So we're going to take that bean and inject it in there, and it's calling the setter on our SpeakerServiceImpl that we created inside of here. That's why I wanted to name it setSpeakerRepository. If we had left it as setRepository, then our property name would just be lowercase repository, so it uses the bean naming convention. Now let's make the changes we need to to our Application.java to load this. Switching to our Application.java, we have to do the same thing that we had done before with our Java configuration, and that is to create an instance of the ApplicationContext to Bootstrap our application. So we'll say ApplicationContext appContext = new ClassPathXmlApplicationContext. Inside of here, we will give it the name applicationContext.xml, and then save that. Now from here, I'm going to comment this out just so you can see that it was hard coded before and we've switched it over to our newer style. We'll say SpeakerService service = appContext.getBean, and we want to pass in here the name of speakerService, and we'll pass in the class type so that it can do this without casting. So we'll say SpeakerService.class, and save this. And now our application is set up to run. I'm going to exit fullscreen so I that I can see the build console. And right-click on main, say Run Application.main. And you can see that our application loaded. So let's walk through what just happened here. In our Application.java, we Bootstrapped our application on line 9, as you can see here, by loading that applicationContext.xml that we created. Our applicationContext here is created inside of our source, main, resources folder, and so that means that Maven, when compiling, will actually compile it out to our classes directory, so under target classes, you can see our applicationContext. So when it goes to run our application, it's actually copied that down to our target classes directory, and that's just where our classpath launches from. And then we created beans inside of here. We started off with a speakerRepository, and then we created the speakerService and did a reference to that speakerRepository into our speakerService that did setter injection, then we changed our speakerService to accept that setter injection. And remember, on line 16, we changed that name to setSpeakerRepository, and that referenced back to our applicationContext where on line 11, we said that the property name was speakerRepository. If we hadn't changed that and just left that as repository, it would've just been property name repository, lowercase. And our application is actually all configured now. The only thing we really had to any different was create this applicationContext where we had a namespace in here. We start defining our beans, and our beans actually look quite similar to what we did in the Java configuration. It just may seem a little bit more second nature to do you to do it in the Java configuration, but you can see how we are now wiring up our application the same way. We have that same setter injection inside of our application now.

## Constructor Injection

So we already defined our bean to use setter injection. There's two types of injection just as we did inside of the Java configuration module. There's setter and constructor injection. Setter injection is using exactly what it sounds like, the getters and setters of our bean, and constructor injection uses the defined constructors. Something to keep in mind though is that you can use both setter and constructor injection together. We're going to now show you how to use a constructor injection example, but I just want to point out that I feel that setter injection is often better for existing code. We've seen our simple setter injection and what it's like to use that. Constructor injection guarantees a contract for us. There's a few nice things about doing that, but namely it's that we have our code and our contract defined when we create that object instance because of the constructor. A positive and a negative though is that I need to have a constructor defined for each situation that I want to guarantee. I should also note that you can use constructor and setter injection together. One other slight difference is that constructor injection is index based and not named based like setter injection is. Let's go ahead and add an injection-based example for our constructor inside of our application now. Changing our application over to be a constructor injection example verses setter is actually pretty quite easy. So I'm going to go into our applicationContext on line 11 and change that to constructor-arg, and instead of name, constructor arguments are index based, so I want to do index= 0. They're 0 based, meaning if I have multiple arguments I'm going to pass in, I have 0, 1, 2, 3, and so on. So we start with 0. So I'm going to say constructor-arg index= 0. Then I need to go to my SpeakerServiceImpl. If I try to run this right now, it's going to break. I want to add two constructors inside of here. I want to add a public SpeakerServiceImpl, and I'm going to give this the no arguments constructor, just the default no argument so we don't break any other examples we're going to do in the future. Then I'm going to add a second one inside of here that is public SpeakerServiceImpl. I'm going to make this one take in an instance of our SpeakerRepository. And honestly, I'm just going to go down here and copy this code that we already have created for our setter because it's going to be very similar to that. So, paste that in there and save it. So our two constructors, one on line 12, the default no arguments, one on line 16 that is the SpeakerServiceImpl taking an instance of the SpeakerRepository, and it's just going to assign those variables to one another, so we have our this.repository is equal to our repository we're passing in, and we'll save that. That's it. That's all we have to do. Since we only have one argument, the index for this one is 0. If we had multiple arguments in there, it would be 0, 1, 2, 3, 4 like in our applicationContext where we just have index 0 on line 11. Let's go back to our Application.java and run this now and see what that looks like. So if we right-click on our main, and say Run Application, it'll run just like we expected it to almost like we didn't change anything, although we did. So now it's constructor injection verses the setter injection. You can see the code for both there. All it took to change that in our applicationContext was changing that one property on line 11 to constructor-arg, and we have it constructor injection based.

## Autowiring

Early on, Spring got a bad reputation for having a lot of XML configuration, and people just didn't care for going through and wiring up every bean and all the references and everything that went with it. To counter this, they introduced a mechanism called autowiring for you to autowire beans together. There are four types of autowiring that you can do on a bean. The first type is byType, and this allows a property to be autowired if exactly one bean of that property type exists within a container. So let's say I have a car object that we're injecting in. If there's only one car object of that type of that class, then we can inject it in. But if I have two types of cars, so two car objects with different names, I'll get a fatal exception because it can't choose which one of those two because it's just looking at the class type. The second option is byName, and that's why you've seen in every example I've named my beans. We could actually go through without naming them, and it would've chose byType by default. ByName fixes that problem of byType. They're both good to have, though, because byType will allow us to only create one instance of a class in our container or in our applicationContext. ByName allows us to choose specifically by the name of the object that we're wiring up, so if I have a car and it's of type car, but byName is Honda, it's only going to inject Honda. If I have a Toyota, it's not going to choose that one, and I won't get an exception based off of that. The next option is by constructor, and constructor is analogous to byType, but it applies to constructor arguments. Now it has a little bit of a flavor of byName in there too because arguments can be indexed or named and injected into our object that way. You do have to, obviously, have a matching constructor for arguments that you're trying to pass into it where the other two are kind of setter based. They need a default no arguments constructor. And the last option is no or none for autowiring. So if I specify no, it means that, no, it can't be autowired at all. Now a lot of people early on tried autowiring for just testing because they were concerned about it and how it performed in production. I've used lots of applications with autowiring in production and never ran into a problem with it. I probably have over 100 applications today deployed into production all configured with autowiring, and I've never seen a hiccup with it.

## Demo: Autowired

To switch our application over to being autowired by constructor, I'm going to go ahead and comment out this constructor argument that we have on line 11 and go to our bean definition and just say autowire, and I can now choose constructor inside of here. If I save this, our application is now set up to be autowired by a constructor. Let's switch over to our Application.java and run this again. And it should perform just like we expect it to and it does. You can put a log statement in there if you want to inspect it a little bit further, but all we had to was change our applicationContext to now specify that it was autowire by constructor. Let's look at what that looks like in the Java code. So we had done this in our other application. I've gone ahead and opened it up a new window. To put that @Autowired on there, we do the same thing on line 22. Now because of its location, it's saying that, by default, it has to be done as constructor based because that's the only way it can do that on that constructor. If we move it down to that setter, then we can choose the different types. Likewise, we can do the same thing in our XML application. Let's look at what that looks like now in our code. Switching this to use the setter, I'm just going to go ahead and change this to byType and go over to our SpeakerServiceImpl. Everything's already set up here. We have a default no argument constructor on line 12. If we didn't have that, it would fail. And we already have our setter defined on line 24. And since we're doing this byType, the name on line number 24 doesn't matter. So let's go back to our code and run this, right-click main, Run, and you'll see our application runs. We've got it set up to do byType. We can actually even change it to byName and save that. Because we are referring to the SpeakerService, and we have a bean named speakerRepository, it's going to do setSpeakerRepository, which is the method name inside of our SpeakerServiceImpl. Let's run it again. Should work for us as well. Right-click and say Run Application. It succeeds. Let's show it to you failing. If you remember when we first created our SpeakerServiceImpl, this was called setRepository. So I'm going to save that. I'm going to go back to my applicationContext and make sure it's byName. It is. There is not a setter named setSpeakerRepository. We have our setter defined on line 24 as setRepository. I just changed that. I'm going to save all of this, go back to my applicationContext, and this should fail. Right-click and say Run Application.main, and it fails, and the reason it fails is that it can't wire that up. It's not been wired correctly. You'll notice it's a runtime exception, so that can be a little bit cryptic to debug is that it's just telling us on SpeakerServiceImpl line 21 that it's null. Well, it's null because it was never injected correctly. It didn't wire those beans up. It just failed and went ahead and proceeded on running. To change that to where it would work, we can go back to our applicationContext and change this to byType, and save this, and when I run our application because it's looking for the type not by the name, it will succeed. There you go. ByType will look at the class type, which our SpeakerServiceImpl did have a method that took a SpeakerRepository, and we had defined a bean of speakerRepository on line 7. Those lined up, but if I had the names wrong, it can be a little bit more brittle. I liked names because I like to name all of my beans and I think it's a cleaner, better example of doing it. If you have an error, you should know pretty quickly. Even though it can be a little bit cryptic to debug, I think it's a better choice for you, but you see how byType will go through and wire those beans up for you, and you're guaranteed to only have one instance. I can't have multiple SpeakerRepositoryImpls in here or it wouldn't inject it correctly.

## Summary

In this module, we showed how to do the similar configuration we had done in the Java application using the XML configuration. We created the applicationContext, which was just like our AppConfig.java file. We showed what a bean definition looked like, and the bean definition code is very similar to the @Bean annotations that we used. We then demonstrated setter injection and constructor injection. Then we looked at autowiring and actually compared that to the autowiring that we did in our Java configuration app. We showed the various autowiring methods as well by constructor, byType, and byName, and showed you how your application can break using one approach and succeed using another. Let's look at some of the more advanced Spring configuration concepts in this next module.

# Advanced Bean Configuration

## Advanced Bean Configuration Introduction

Even though this is an update for this course, we've decided to add this additional module on Advanced Bean Configuration to address some questions that people have had over the lifespan of this course. This also came at a request from Spring themselves as it's helping people that are trying to achieve some of the various Spring certifications. These are some of the topics that they go over in there and want to help people by exposing them to them here. We're going to dive into some of the more edge cases. We're going to talk specifically about advanced bean configuration using these various techniques. First off, we're going to look at BeanPostProcessors and init methods and how that ties into the creation of beans and having a hook to have an additional method that acts kind of as a callback. Then we're going to look at FactoryBeans. FactoryBeans are a great way to create a Spring bean that has static methods inside of it. We're also going to look at SpEL, or the Spring Expression Language, and then we'll look at proxies and, finally, bean profiles. They're typically used in a DevOps or configuration per specific environment-type scenario. Let's start out by looking at BeanPostProcessors and how they can aid us in that callback with building out beans.

## Bean Lifecycle

The bean lifecycle is a more advanced topic, but it is worth showing to you even in a fundamentals course. It starts off with instantiation and then it moves onto populate properties. These are read from either a properties file or injected in from other resources. Then the framework sets the bean name and makes it aware to other resources. We can then set that as a BeanFactoryAware context. Next, we do pre initialization utilizing BeanPostProcessors. From here, we can initialize the bean utilizing properties that were just set. Now we can call an init method, and this is actually the piece that we're going to demonstrate that's very useful even in this fundamentals course, and then you wrap up that entire initialization process with another set of BeanPostProcessors. We're going to look at a more common use and that is the configuration of an init method on a bean, and let's do that demo now. Adding an init method is actually quite easy to do in our application. I've gone ahead and opened up our conference-java app. I'm going to navigate to our pom.xml. We need to add a dependency inside of here because Java EE has now assumed an annotation for post construct API calls. So I've added this javax.annotation javax.annotation -api version 1.3 .2. That was the most current version at the time of recording this course, dependency inside the dependency section of our pom.xml. I'm going to save this. When I do, it will automatically import those dependencies inside of our application. You can check that by looking in your external libraries and see if it's pulled that in there, and you can see we have our javax.annotation library that it's pulled in as a dependency there. I'm going to navigate down to my service and specifically the SpeakerServiceImpl. And you can do this in any bean that is configured by Spring. I want to navigate below the constructors that we have implemented in here. We have a no arguments constructor and a constructor that took a SpeakerRepository in. The beauty of this is it doesn't matter which constructor we're calling. It's still going to be called after those have ran. So I want to add in a private void method. I'm going to call mine initialize and no arguments in there. I'm going to add a System.out .println in here that I'm just going to dump out a statement that says that were called after the constructors. Put a semicolon on the end, and now the last thing that makes the magic happen. We're going to put in a @PostConstruct annotation, and that's it. So created a private void method, no arguments, tagged that with a @PostConstruct annotation, added that library inside of our dependencies, and now I can run our application. When this runs, you'll see that it called the SpeakerServiceImpl no args constructor, and then it's called that method where we're called after the constructors and before anything else is ran. So if you remember from our application, we have this debug statement that's dumping out for each one of these services just showing that they were singletons from the previous example and then the name that we're calling. You can see how we can hook that in there. And the nice thing about this is we can have just a simple post construct method that gets called for any type of logging or configuration things that you want to put in there. Now one thing I would caution here is I personally would not put things like obtaining connections to a database in here. I see people want to go through and add things inside of here that I don't believe fit here. Closing and opening connections, they should be handled by Spring not by you programmatically in this post construct or a pre-destroy annotation. That's another one that's available. You get the idea of these being aware annotations in your code. They should be used for configuration things that you just want to run after all of your constructors. Don't be opening and closing connections. Just a word of caution.

## FactoryBean

FactoryBean configuration is very similar to the init method configuration we just did. It actually builds on the factory method design pattern, and at least its functionality is built to work the same way. It's actually one of the strengths of doing this with Spring. The code doesn't actually have to be written as a factory to be utilized in Spring as a factory. It's also a great way to integrate legacy code into your application. You can actually enforce a contract in how your code is configured without creating a specific constructor for it. Typically, creational contracts are implemented in a constructor, but with legacy code, we often don't have the ability to modify it due to existing commitments. Using this pattern enables us to establish a contract without modifying that code. Lastly, it's how you can also work with static methods inside of a class. Let's look at implementing a simple demo of a FactoryBean inside of our code now. For our demo, I've opened up our conference-java project, and I'm going to navigate down to my source, main, java directory and specifically my com.pluralsight package, and right-click, and say New, Java Class. And in here, I want to type util.CalendarFactory. And for our demo, we're actually going to wrap a calendar instance in a factory so that we can manipulate some things and show you some examples with it. So now that we have this in here, we want to type implements FactoryBean, and FactoryBean is typed, so we want to pass in or type it with a Calendar object. And you'll have a red squiggly line underneath the code, and it's because we need to implement the methods that the FactoryBean interface is telling us we need inside of our application. So if I hover over that and choose Implement methods, and I want getObject and getObjectType, click OK, it'll out those default implementations in there for me. Now before we get into editing those, let's start off by saying private Calendar instance = Calendar.getInstance. And then in our getObject, we want to return that instance, and for our getObjectType, we want to do Calendar.class. Now we have the contract for our factory setup. The next thing that we want to do is add some value to our factory wrapped bean, so we're going to add a public void addDays method inside of here, and this is just a simple example to show that we can take this static instance and add some functionality around it that we can then utilize inside the rest of our application. So we're going to take and say instance.add ( Calendar.DAY_OF_YEAR, num). So what this will do is just take and add the number of days to our instance that we pass into this method. So we'll save this, and it's a little bit of a contrived example, but it gets the point across. Now we have our FactoryBean implemented, and yours should look similar to mine. We have our imports and package structure all the same at the top. We have our factory that implements the FactoryBean and it's typed with Calendar. We have our instance. We get our object back, our contract with our Calendar.class, and our method to add some days to that instance. So now we can go ahead and configure this, and we're going to do this a little bit different than you might expect. We're going to open our AppConfig that we had before, and we want to create two beans inside of here. We're going to create one for our actual bean instance and one for our factory. We're going to start off with the first one, and this is an @Bean, and we're going to give this a name = cal, and then we want to create a method that is public CalendarFactory, and we'll call this the calFactory. And it will have an instance of our CalendarFactory factory = new CalendarFactory. And this is where we can say factory.addDays. We're going to pass in just the number 2 here and return our factory instance. So there's our first bean. The second bean we're going to do as @Bean, and we'll do public Calendar, and this will be our actual calendar instance, cal, and this throws Exception. And inside of here, we'll return calFactory.getObject. So this is where we're actually getting the instance from our factory itself. Now, we need to add all of our imports, save this, and our bean's configured to use. Now you're wondering, what did we just configure? Well, let's implement this into our code and actually see how it's used now. You actually don't need to worry about line 12, that method for the calFactory because we're just setting that up to use. We're actually going to look like we're using it from inside of our code as an autowired bean. I'm going to open up my HibernateSpeakerRepository. And inside of here, now I can just say @Autowired and do private Calendar cal. And inside of my code, to make this a full- fledged example, I'd go ahead and add a date or something like that to my speaker object. I'll just put a System.out .println here where you can see what this instance actually is. And we'll say cal, and we'll do cal.getTime, and save this, and now we can run our application. So all I did was autowire this bean in, and as you can maybe guess, it's going to have been called from the FactoryBean of our calendar object. We have our static code that was called from that calendar object, added days to it, worked with it, manipulated things, and now we can reference it inside of our code. Before this, there was not a way for us to really reference a calendar object and pass it in this way. Let me run our application, right-click on our main method, and say Run. And you'll see when this runs down below that we have our no args constructor, we have our post construct still in there from our previous example, our service instance, but you can see I have a calendar object and it's added 2 days to my object and printed that out to the screen. Then I have our name and that other object address. So you can see right here that we have our cal: Fri Nov. That came from our SpeakerRepository instance, and that's hinged on the day that I ran this where it's printing this out to the screen. So I have my calendar that's dumped out, and it's going to head through that CalendarFactory, added those days to our years from our configuration. So we added two days to it. This is a great way if you had to pass in a bunch of dates, or times, or objects of that nature into your application for you to utilize a static instance like this and be able to manipulate that and then inject that wherever you want in your code. Maybe you have audit columns on your database tables. There's other ways to do that, but just shows you a good example of how you can utilize this.

## SpEL

Spring Expression Language, or SpEL as it's often abbreviated, is a powerful expression language from the Spring Framework. From my experience, it's typically used in libraries, but it can also be used to do some convenient changes to your code. Using it, you can manipulate the object graph. This simply means that once an object is created, you can use the expression language to manipulate that object. You can also use it to evaluate values at runtime and change the behavior of your code accordingly. It can also be used to evaluate and manipulate your configuration. Let's look at how we can inject some values at runtime into our application. Without building a more complex application, let's add a seed value to our model object. I'm going to open up our Speaker. And a lot of times as we are doing security and other things like that, we'll have values that get seeded at runtime. I'm going to add a private double, and I'll call this the seedNum. And then I'm going to come down below and just generate some getters and setters for this object. So I'll right-click, say Generate, and we'll do the Getter and Setter. Click OK and save this code. And now I want to go up to my HibernateSpeakerRepositoryImpl where I create this. And inside of here, I'm going to actually use the Spring Expression Language to grab a value out of the java.lang .Math library. So I'm going to type in here the annotation @Value, and inside of here, I want to do the pound sign, the open curly brace, and say T( java.lang .Math ), and I'm basically grabbing the math object here and saying .random for a random number, and this gives me a very small decimal, so I'm going to times that by 100, and then close that curly brace off. And now I want to go ahead and put that in an object, so I'm going to say private double seedNum and save that. And now I can, at compilation time, take that value that's added at runtime and inject it into my code. So, I'm going to go ahead and say speaker.setSeedNum, and I'll pass in seedNum and save this. So now this object's been initialized on line 19, and we can store that value inside of our speaker object. And a use case for this is, like I mentioned before, when you're doing security and you create passwords, a lot of times those are salted or seeded with a value that you pass in, and they'll often times use a calendar day or some random number, so that's what we're going to do here. We'll save this, open up my Application to run this, and I'm just going to grab this line. And since we're getting a lot of stuff in here, I'm going to comment out these two System.out .printlns. In fact, I'll even comment out this SpeakerService where I'm getting the second instance, and we'll do getSeedNum here. And let's run our application and see what it generates. Now that we've got everything compiled, you can see here where we've still used our no arguments constructor. We have our post construct in there. We have our name and our calendar instance that we're grabbing from up above and the value of our seeded values. We have 94.888, and it continues on. That's a seeded value that we passed into our application, using the Spring Expression Language. So a very powerful way to grab a random number, and you can do a lot of stuff with this. We can manipulate stuff out of a list or pull specific things, we can evaluate something to false or true at a runtime value, but just as a basic example of how easy it is to get this inside of your application using this expression language.

## Proxies

Spring often utilizes AOP proxies for its code base. Proxies are a great way to inject behavior into the code base without modifying the underlying code. In the code snippet that you see on the slide, this simple POJO is the class that we want to proxy to, and the POJO is the interface that we will make calls on through the proxy and allow the calls to be intercepted. To be honest, this isn't something that the average developer typically needs to work with, but it's a good thing to know. When should you use this? Well, you probably shouldn't be looking for opportunities to use it. It's usually when you are doing something framework or library based. One of the most used cases in the code base is when dealing with databases and using the annotation @Transactional. The @Transactional annotation is used to be the starting point of your code base in which you would like the following code to be surrounded with the transaction. Writing one is really beyond the scope of this course, in fact, I think you could have a whole course on proxies and the best way to implement them, but I wanted to show you what they are and point out that they are used often in the Spring code base. Let's look at one last concept and that's bean profiles.

## Bean Profiles

Bean profiles were implemented later in the Spring code base to help you adapt to environments. They allow you to set up specific code that gets ran only in a specific environment, so we can swap out configurations at runtime. It's a great way for us to have effective runtime configurations. I'll be honest, the first time that I saw these, I wasn't really a fan of them because I don't like it when my configuration changes in environments and then it's not the same. I've always worked on my code base to not have that be the case. But when dealing with sensitive data or that type of information that I'm going to go pull something from a dev or sample database until I get into production, it's a great way for me to set up a profile for that. Let's see how we want to implement this inside of our application. Adding bean profiles is actually quite easy to do inside of our application. I'm going to go ahead and open up one of our bean definitions. We'll start with the speakerRepository. I want to go right above the class declaration and add in an @Profile annotation. And I'm going to give this a keyword of dev, and that's a word that we chose. It's not a keyword specific to Spring. This can be anything you want. It could be staging_42. It's whatever we've chose. We'll do @Profile dev for the HibernateSpeakerRepositoryImpl. And we'll open up our SpeakerServiceImpl and do the same thing here. So we'll say @Profile, and, again, we're going to choose dev here as well. And we want to go inside of our application and edit the run configuration. This is how we tell it to choose which profile we're going to run. So in the VM options, I want to do a -Dspring.profiles .active =dev. Now, don't make the mistake of putting it in the environment variables or in one of the other fields. It will not work. It needs to be a VM option. Let's click Apply and hit OK. And now when we run our application, it will run with a specific profile. You'll see that everything runs. Well, great. What happens if it didn't have a profile for that? Well, that bean wouldn't be available. So if we change this to prod, and save this, and run it again, we'll get a null pointer exception because it can't find that bean. So you see how those beans will allow specific code to only be ran in specific environmental configurations. I'm going to change this back to dev. And I want to show you that this HibernateSpeakerRepository really is a good example of where you'd want this profile. We have hard-coded values in here that eventually we want to pull from a database, but we don't have a database set up. So, I want to set this up to only run in dev. It's kind of like a to do statement if you use that in your IDE or in some of the other tools that are out there saying, hey, I want to come back and fix this. Well, by marking this as dev, it's not going to get ran or accidentally deployed into production where you break things.

## Summary

We looked at a bunch of advanced bean configuration techniques in this module. We started off by going through the bean lifecycle, and the key point with that, there were a lot of steps in there, you want to pay attention to the init method step. That's what allows us to hook into that post construct annotation and add that functionality to our beans. Next, we configured a

FactoryBean from scratch and implemented some static code inside of our application, and that's really one of the main takeaways there is that when you're needing to integrate some legacy code or maybe some code with a static initializer, the FactoryBean and wrapping that is a great way to go about that and be able to autowire that and still inject it into your application now. Then we went through Spring Expression Language and showed how you could go ahead and interface some other code and libraries and manipulate the object graph at runtime. It's a very powerful way for you to go about doing that. Then we talked briefly about proxies. Proxies are a very advanced technique. Just wanted to whet your appetite a little bit with them and show you that they're out there and make you aware that Spring uses them, but they're not something that you would use on a day-to-day basis and you shouldn't be looking for opportunities to integrate them inside of your application. The last thing we did was implement bean profiles inside of our app, and it was a great way to show you that you may have some code in there that you want to make sure doesn't get into production. If you use it just for that, it's a great tool. So what's next for you? Well, this is the end of this course, and if you enjoyed it, I would really appreciate if you would leave me a review. If you liked my course, there are a bunch of other courses that I've produced out there. In fact, one of the next ones I would recommend is the Spring MVC intro course. There is also a great course on Spring Boot fundamentals that Dan Bunker has put together that is a great course as well. I've taken it myself, and I would recommend both of those to you as a next step in your journey and continue to follow along the learning path for Spring, on Pluralsight. Thank you.