**Introduction**

**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 are going to be learning about developing with Spring, and we will use the various configuration methods to illustrate some common problems we often face in software development. We will develop part of an application without Spring and then show how to configure that part of an application using XML in Spring, Java configuration, and then lastly, annotations in Spring.

**What is Spring?**

So, what is Spring? Well, the Spring framework started out as just an inversion of control container. It was conceived to reduce or replace some of the complex configuration of earlier Java Enterprise Edition development. Spring was later built around using Java without EJBs, so its original concept was how to work better with EJBs, and then they realized that they just kind of need those for a lot of situations, so it kind of transitioned into, Okay, how do I do the same development without using EJBs? 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 EJBs and typically, 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, it's just a web server. This is one of the reasons that Tomcat has taken over for the Java development standard container. It's easy to use, it's lightweight, and until Spring, you either weren't using enterprise features, or you had to use more complex, harder to use application servers. I should also mention that Spring is also completely POJO-based. Any code you write in Spring can be written without Spring. POJO, as you may or may not remember stands for a Plain Old Java Object. The Spring framework really isn't doing anything magical behind the scenes, it's just helping us write better, cleaner code, and doing things POJO-based and interface-driven. So, we've already talked about being lightweight, but we should also point out that Spring was built out of the frustrations of J2EE so it's really unobtrusive. It shouldn't be getting in your way. If it is, you're probably 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 code. 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 those best practices are built into our code inherently, just from using Spring. 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 singletons, factories, abstract factories, and namely, the template method pattern. Spring uses the template method pattern a lot, and although it's really not a pattern, it is design-based methodology or best practice that'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 these details.

**Update**

This course has actually been updated, and since its initial release, we have used Maven to do the builds in our course. The first release of this course did not use Maven, as a request from some developers, saying that they just wanted to focus on Spring. Well, the Spring framework has decided that they want to allow you to only download the individual libraries while using Maven or Gradle. So, in this course we will use Maven to manage our downloads which is a change from the previous course, if you watched that. If you didn't watch it, it will not matter and should flow just fine for you, but it is a significant change from the previous version.

**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 this is actually buying me. These are all the big questions I ask. What is it 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 things like caching and other tools in there without having to rewrite our code base. That's a little bit beyond the concepts that we're going to discuss in this class, but these are all the focus that Spring and the principles behind Spring help us deliver on. It also helps us reduce the complexity of our code. Lastly, though, and this is my biggest point that it tries to solve for us, is that it puts the focus on the business. The business doesn't care what framework I'm using, they care that we're getting code done, and Spring helps us get complex code done faster, and makes it more maintainable, testable, and helps us remove that complexity, just helps make everything easier for us as a developer.

**Business Focus**

Let's talk about business focus for a minute. If you've ever done much database development in Java, you've used JDBC. And JDBC, at first glance, doesn't look that bad, but then you really start picking it apart, and realize it's really not that pretty. Look at all that extra stuff we have in our code, just to do a simple select statement. Candidly, I had to make the font smaller for this demonstration, just so I could fit it all on this slide, and I've even left out some extraneous parts to get it to fit on here. We have connections, prepared statements, result sets, and then we have a try-catch-finally block, we have a driver manager, all this stuff in there, and look at the great big finally block of if, or else, equals, null, if statement equals null, if connection equals null, all this empty curly-brace catching exceptions, there's just a big bunch of code in here, when really all our business cares about is this line here, where we say that we select whatever field we want from the CAR table with an ID, and then the other line that the business really cares about is where we grab that actual result and add it to our CAR and return that. So, out of all of this, there's two lines that the business cares about: that we've got our CAR based off of the ID, and we've created that CAR object and return it. I'm going to show you a few things that Spring does to make this easier, so that all these places where we're assigning things, and handling, opening and closing connections, and of this type of stuff can just be handled for us.

**The Solution**

So, the solution we're looking to get out of Spring, or from using the Spring framework, is that we can remove configuration code, or lookup code, and then developers can focus on the business needs. Like I said, that's a big one for me, is that the business 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, on that previous slide, it really wasn't that testable, as far as things are hard coded in there, and I have a driver manager that I'm grabbing a connection, where that connection's coming from, that type of stuff. Spring also helps us with doing annotation or XML based development. If we want to annotate our code, rather than having so much configuration code in there, Spring can help us with that focus as well. So, we make things easier to test, make things focused on what the business needs, and remove configuration lookup code through doing annotations or XML based configuration. Let's look at that business focus again, and see how Spring can help us clean that up.

**Business Focus Revisited**

Here was our code before using the get connection, and driver manager, prepared statement, the huge finally block that essentially does nothing but close connections for us. And now let's look at what that code could be, using the Spring framework. Now, I will say that this one particular library in Spring, not necessarily Spring core, is enabling us to condense this code, and that is the Spring JDBC template, but it's the same concept. Notice that nowhere in here is it mentioned that we need a connection, a result set, prepared statement, or using the driver manager, nowhere it talks about opening and closing that stuff. In fact, there's not even a try-catch-finally statement at all in here. We have our find and our return statement, and this is using an entity manager in there as well, but this is an example of the template method and Spring utilizes this a lot to help us clean up our code. This is one of those cases where Spring is using a pattern, and we really don't have to know about what that pattern is, or how it's doing it. This may scare some of you, that there is a black box with Spring, and I hope to alleviate some of those concerns, but look how at much smaller and more condensed this code is, and it's really just doing what we want it to do. It's got a great emphasis on business focus. There's no reason we should have to rewrite what's on the left for every query we need to make inside of our application.

**How It Works**

So, how does this all work? Great, we've seen that it can reduce business focus for us. 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 almost as a glorified HashMap. Now, I don't want to dumb it down 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 that there's too much going on behind the scenes, there's too much magic happening with Spring, because there really isn't. Spring can also be used as a registry, and that's how I'm going to show you to start with it because we're just going to use a little main method in our application to run it, but we really get the benefits of Spring through its wiring constructs, and using auto-wiring. And I wanted to show you this image on the left, of what the Spring container would look like if you graphed it all out. You can see all these little squares in there are just beans, and you can see their references. Some of them are standalone, some of them are referencing other beans. Just how they're all wired up, and how it makes all these objects that we're using together. So, to take that JDBC example that we were looking at, one of these might be a statement, one of them might be a prepared statement, one of them might be a connection that's behind the scenes, an entity manager of some sort, so all of our beans get stored in this little container and then we access them out of that container and that's where that HashMap kind of comes into play.

**Summary**

In this short module, we talked about what is Spring, and the problems that we're trying to solve with Spring, and what we're going to get out of it, and we took a look at what the business focus is, and you're going to see me drive towards that in all of our examples of how we want to do this, and what the business gets out of us by using Spring. Let's go ahead and dive in to our application now, and set up that sample project that we're going Spring-enable through the various configuration methods.

**Architecture and Project Setup**

Sample App Intro

Let's talk about the architecture of Spring and the application we will build in this course. We will walk through the basic project setup and historical approach to what Spring solves. 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.

**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 to you about why Spring was developed the way that it was. Spring was developed to make existing tasks easier. Before it 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 things that we are trying to fix. This problem is referred to as write once, run anywhere, and is often abbreviated with W-O-R-A, or 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 hardcoded inside your application. This is what Spring's architecture is going to ultimately help us with.

**Prerequisites**

There are a few prerequisites you'll need to do with the examples in this course. I am using Java 8. With Spring 4, you are required to use Java 8, and this is one of the reasons for releasing this course. I'm assuming that you'll already know how to install Java, so we won't be covering that in this course. I am also going to use Maven in this course. Previously, we did not use Maven and had some requests to have this as a standalone course not using any build tools. Spring now forces you though to use Maven to download your dependencies for your application. I do have a module at the end where I'm going to show you how I would use Spring Boot with this, but people have requested to have just Spring standalone by itself. There is a Pluralsight course on Maven if you have more questions about that. You can go research and follow it. The next thing you'll need is an IDE. I am using Spring STS, which is just a flavor of Eclipse. In fact, it's just rebranded Eclipse. If you already have Eclipse installed, you can use that, it'll work just fine. I like Spring STS a lot because it has some of the common plug-ins already installed, and you don't have very many, if any at all, configuration issues. The steps for Eclipse and Spring STS should be nearly identical, if not the exact same. And finally, we're going to use Tomcat. You won't need this for any of the examples that we're going to be walking through, but the sample that I have at the end that will also use Spring Boot, I will deploy to Tomcat as well. So if you have that, great. If you don't, don't sweat it, you can follow along my example and see what I'm going to do. So mainly, Java 8 and the latest version of Spring STS and Maven.

**Sample App Setup**

Let's go ahead and create a sample project now to show the problems that we currently have in our code and how Spring can help us fix those. I'm going to go ahead and open up Spring STS. And if you haven't run this before, it may take a minute to load. The first thing it's going to do is ask us for our workspace. I'm going to put all my code under C, dev, workspace. Really shouldn't matter where you select your workspace as, but just make sure that you remember what that location is because you may want to reference it from the command line as well. This should load up fairly quick. Sometimes the first time you load it, it does take a little bit longer. If you have just barely installed Spring STS, there is oftentimes a dashboard that'll pop up. You can go ahead and close that out. The first thing I'm going to do is go over to our Package Explorer, I'm going to right-click and say New, Other, and choose Maven, Maven Project. Now we just want to create a simple project, so I'm going to go ahead and check that check box. If you don't do this, it'll take you through an entire archetype selection, which is not what you want to do. It's a lot more complex and will add a lot of extraneous code that you don't necessarily want in your application. So let's go ahead and click Next. And for our Group Id, we're going to enter com. Pluralsight. For the Artifact Id, let's do spring\_sample. And then our version can still stay as 0. 0. 1 SNAPSHOT, and the packaging can stay as jar. If this stuff is confusing to you, you may want to look at the Maven course, but this is the basic, the basis of what our project is and what you need setup for this to run. We click Finish, this goes ahead and creates our project for us, and there's going to be a few things in here that you'll see. We've got our folder structure for us, src, main, java; our properties and other source files directory, such as src, main, resources; it also created some test directories for us, src, test, java and src, test, resources; as well as our pom. If you open up our pom, you'll see inside of here that it's just a standard Maven pom file and there's not much in here yet.

**Sample App pom.xml Demo**

The first thing that we're going to do inside of our pom. xml is fix the compiler version that our IDE is using. If you look over on the left, you can see that our compiler is currently set to J2SE 1. 5, which is the default for our IDE. Since we're using Java 8, let's go ahead and step that up to using Java 8 for our project. I'm going to go ahead and make this full-screen so you can see it a little bit better. We want to go in between our version element and our closing project tag and add a build element. Inside of the build element, we want to add a plugin section, and make sure you do plugins, not plugin management. We're going to add the Maven compiler plugin in here. Now the first thing you need for the compiler is to add a group IDE, and we are using org. apache. maven. plugins. And then we're going to add our artifactId. An artifactId is the actual element pointing to the plugin that we want. If you're a little unclear on Maven and Maven's use, I would recommend, as I mentioned before, going through the Maven course on Pluralsight. So we have our Maven compiler plugin, and then we're going to add the version that we are currently using, and that is 3. 2. And lastly, we need to add a configuration section. So we say configuration, and inside of here, it has two elements: a source, which we want as 1. 8, and a target that we want as 1. 8 as well. Once we have this in here, we can save it, and a lot of times, Eclipse or Spring STS will go ahead and automatically update your project for you once you save it. So if I save this, you'll notice it's still stuck at J2SE 1. 5. We need to right-click on our project, scroll down the Maven and say Update Project. It'll make sure that you've selected the right project if you have multiple ones open. It may not be this one. So choose our project, go ahead and click OK, and you'll notice that it's now bumped it up to J2SE 1. 8.

**Sample App Add Model Demo**

Let's now add a class file to our application. To do that, we're going to go ahead and right-click on src, main, java and say new class. We want to change the package up here because it'll default to our application name or our artifact Id. I want to change this com. Pluralsight. And then for this particular class, we're going to put it in the model package. So com, Pluralsight, model. And this is the customer object. You're going to see that we'll break everything out by the individual packages that we have inside of our application. So let's click finish. And now, inside of here, we're going to add a couple things. We are going to add a private string, firstname, and a private string lastname. And this is just going to be a basic object, a standard DTO, if you've ever worked with some of the blueprints stuff before. And to do so, we're going to right-click and say Source, Generate Getters and Setters. And we want to select firstname and lastname and click OK. Now you'll notice a couple of things. It'll stick those getters and setters wherever it wants to, and it doesn't put a lot of white space in between them. So I like to get in the habit of right clicking in my source and selecting, Source, Sort Members, and this will organize them the way that I want them to be alphabetically. And then I also want to do either Control + Shift + F, or you can also go back to the Source menu and choose Format as well. And that'll add the white space in there and any other spacing or tab characters or things that you have set up. If you have that team preference set up wherever you're working, or just your personal preferences, this will go ahead and format that code to be standardized how you like to see it. Now one thing I'm going to add about this before we move on to the other tiers is that notice we're currently just using the default no arguments constructor. It'd be the same thing as me saying public Customer, and just having an empty no arguments constructor in there. I only point this out because by default, this is what Spring is going to use, and we're not going to be doing any constructor injection this early in the course. We're going to talk about it later, but I did want to point out that it is using that default no arguments constructor right now. So let's go ahead and save that and move on to our next tier.

**Sample App Add Repository Demo**

The repository tier is a little bit more complex than the model that we are working with, but not much. We're going to right-click on src, main, java, and start off by saying New, Class, and we want to go ahead and change the package up here. Don't miss this step. We want to change it to repository. And for the name, we're going to start off by doing an implementation specific repository. So we're going to say HibernateCustomerRepositoryImpl. Now I'm doing this on purpose to illustrate that one of the reasons we want Spring is to be able to swap out this underlying implementation and code to the contract. So let's go ahead and click Finish. It should have created you another package over here with that object in there. Now to build this, we're going to go ahead and put a default method in there just illustrate what we would be doing with this object. So let's add a public list, and let's use Control + Spacebar to go ahead and have it import Java util. And we want to return our customer object. And this method is going to be a findAll method that will return a list of customers. So since we are in a separate package, you may need to use Control + Shift + O to import that customer object. And you'll notice that it added that line in there for us. Now we're just going to put some basic default implementation in here. You could actually almost call this repository a stub rather than the hibernate repository, because we're not going to wire it, clear up, to hibernate in this example. But you'll get the point that we're trying to illustrate. So let's create a new instance of a list. I'm going to say customer, and go ahead and call this the customers list, and it is a new ArrayList. Now inside of here we are going to just add some default implementation and return it. So I'm going to create a customer object and set inside there just some basic parameters. I'm going to use my name actually. If you want to customize it for yourself, go on ahead and put your name in there. And we'll set customer. setFirstname, Bryan, and customer. setLastname, Hansen. Add that to our customers list, and there's our customer. And return our list. Return customers. Now this, let's make this full-screen so that you can see all the code that we added in there, just a basic stub. method. But to use this inside of Spring, we want to code this to the interface. So what we're going to do is now extract interface out. And there's two ways we could go about this. You could go ahead and right-click over here and say New, Interface. That's one way to go about doing it, and then you could code all that stuff in there by hand. The other thing that you can do though is go through and extracting an interface out. And I'm going to go ahead and do that. So to extract that interface, I want to take and right-click in my application and go to Refactor and say Extract Interface. And the interface name is CustomerRepository. And the methods that we're going to include in this is the findAll method. Go ahead and select that and click OK, and notice that it created a customer repository interface right alongside our customer repository implementation. I'm going to open it up, and you'll see that it's adhered to all the constructs that we have in place in the pieces that we want for our customer repository. So when we reference our hibernate customer repository, we can now do it through just our customer repository interface.

**Sample App Add Service Demo**

The next piece we need to add is the customer service tier. And so this is where all of our business logic would be created and stored and those types of things. We're going to right-click on src, main, java, and say New, Class. We're going to go up to the package and say com. Pluralsight. service, and then for the name, we want to do the CustomerServiceImpl. And click Finish. And just like the customer repository, we want to go ahead and add a few things inside of here. What you're going to notice is we're starting to hardcode some things in our application, and that's what Spring is going to help us fix. Start off by creating a new instance of that customer repository. Let me say private CustomerRepository, customerRepository = new HibernateCustomerRepositoryImpl, all right? Now this is going to be a very simple service tier. I've seen these get very complex because this is where you want your business logic to reside. This is also where, if you were creating a web service, where the heart of that web service would be. The actual interface would be in the web tier, but the business logic resides in the service tier. So if you're wondering why I have a pass-through from the service to repository, we'll talk more about that. But for this, we're going to just add a simple pass-through method that's public List, and it's going to reference our findAll method that we have in our customer repository. So we'll say find all, and we're going to return customerRepository. findAll. Now this is a very basic application. You may need to do Control + Shift + O if you have a red line over your customer object that just organizes your imports, the very basic pass-through of our service that's going to go through and reference our repository object. But notice here that we have a hardcoded reference to our customer repository. This is where Spring starts to step in. So we don't have these pieces hardcoded inside of our application. Just like our repository tier, we want to extract an interface out of this, so we're going to right-click, we're going to say Refactor, we're going to Extract Interface, and we want to call this the CustomerService. And we're going to select the findAll and click OK. And that should create a service interface for us here as well. Now that we have all these pieces done, let's go ahead and create a small sample application to illustrate all of these working together.

**Sample App Run Application Demo**

To create our application, let's just right-click on src, main, java and say New, Class. And we are going to use the default package for this just because this is going to be a sample application to run our app. We could also do this as a unit test or some other means to run this, but this will work for what we're trying to illustrate. So we're going to just call this application, and I do want to select public static void main for it to create one. So let's go ahead and click Finish, and it should have created an application. java under the default package. And inside of here, we're going to just add some code to illustrate what's going on inside of our application. First, I want to start off with creating an instance of our customer service tier. So we'll say CustomerService, and we'll say service equals new CustomerServiceImpl. And again, here we are hardcoding this in here. This is what Spring is going to help us remove. And as we get that instance, let's go ahead and just do a System. out. println. We want to do service. findAll. And just for sake of this example, we'll say. get, and we're going to do just an index of zero. We know we have one record in there. We're just testing that everything works right. In fact, we'll oftentimes call this happy path testing, in the code sense because we want to just test it. What should be working is in fact working. So now when we save this, if we right-click on our main method and say Run As, Java Application, it should go ahead and return Bryan. Or if you put your name in there, your name. And now all of our tiers, all of our pieces are all wired together. We have our application that creates an instance of our customer service object. Our customer service object then calls our repository. The repository has a findAll method inside of there that will go ahead and create an instance of that object and return it for us. From here on out, we'll work on cleaning up this code and showing you the different injection methods that we can go through to not have hardcoded references in there to make things more easily testable, and how we can go ahead and swap pieces out without having to rewrite entire portions of our application.

**Configuration**

Spring is all about removing configuration code from your application. Why is configuration code such a bad thing in your application? Well, it makes things brittle. And by brittle, I mean hard to move to different environments. You maybe haven't experienced this in your career yet, but if you ever had to recompile a code because you are deploying to a different server environment, what about testing? One of the things I often 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 base. We have a really complex code. I'll tell you, it's usually not the complexity of the code that makes it hard to test, it's the way that, that code was written. It's just a perfect example that if you opt to not test code because it was too hard, it's not the testing that's typically hard, it's the configuration of that code that's hard. Maybe it has a reference to JNDI or database or some other thing like that. But more often than not, there's 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 within your application. We've already developed part of an application without Spring, now let's look at some of those pain points that we're trying to overcome by using Spring inside of our application.

**Pain Points Walkthrough Demo**

To go through and illustrate what some of the pain points are of our application as we're going through and applying Spring to it, these are things that we want to remove from our application, such as this new HibernateCustomerRepositoryImpl that we can see inside of our application right here. The CustomerService shouldn't know that it is using hibernate specifically. We've already abstracted that out to an interface, and this is a good example of where we could use a design pattern like a factory pattern to abstract that out of this code, but Spring makes this even easier for us. And if we go inside of our CustomerRepositoryImpl, we can see that we have some hardcoded data in there, and things are just too smart about what's going on right now. If we drill down to our application inside of our application, we know specifically that it's using this customer service, and we can see that specific things are being called and loaded from inside of our application. So wherever you see us creating an interface and it's tied to a concrete implementation, we should try and abstract that out of our code so that our application isn't hardcoded anymore.

**Spring Download**

One of the reasons for re-releasing this course was because Spring quit offering a direct download for the compiled jars. Sure there are 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 they just focused 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 to manage its dependencies. The Maven repo has the source Java docs and binaries all available for us to download and integrate into our IDE. One of the main reasons Spring wants you to use Maven is because of the transitive dependencies that are required to run projects. This is where a lot of people get confused about which jars they need to add into their application. And Spring has made this simpler through using Maven. Spring Boot could be used to set up our application, but it's a little bit too much of a black box if you're just learning Spring, and it doesn't show people what Spring needs to run nor how to set it up. We will set up a Spring Boot application later though, and compare the differences inside of our application. Let's get our app configured and use Maven to download the dependencies that our app will require.

**Spring Download Maven Demo**

Let's begin adding Spring to our application by downloading it from their website. It's not quite what you're thinking because we said we were going to use Maven to download Spring, and we are. Let's go ahead and pull up a browser and go to spring. io. Pull up their basic webpage and you'll notice there's a projects link. If we click on that, there's a section for the Spring Framework, and this is the one that we're looking for. If we click on that, it takes us to the webpage for the Spring Framework, and you'll notice that they've made it to where we can already grab a dependency detailing for the current Spring release. At the time of recording, that current one is 4. 3. 2, but Spring is pretty good about releasing current updates all the time. We currently don't have a dependency section in our application, so we're going to go ahead and grab that entire dependency section. Let's copy that and come over to our application and double-click on our pom. xml. It should be at the bottom of your project structure. And let's click this pom. xml tab. And in between our version and build section, let's just paste that entire dependency section in there. Now I'm going to save that and hit Control + Shift + F to format my code and clean up our XML. Now when we click Save on this, you may have noticed a couple things happen on your application. You may have noticed some downloading or some other things going on in the background, and you'll notice we now have a Maven dependency section down here. And look at what it's downloaded for our application. We have our spring context, spring aop, spring beans, core and common loggings downloaded into our application, as well as spring expression. Believe it or not, that's all it took to add the Spring dependencies to our application. Now there's more than one way to add dependencies to your project, and we'll talk about those other ways later as we grab other things that we might need to complete the buildout of our Spring dependencies.

**Summary**

Let's quickly recap what we learned in this module. We covered a lot of things, and this was a fairly large module, and it set us up for all the upcoming modules that we're going to go through. We discussed why we would 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 up, and discussed the pain points inside that demo app of why we want to use Spring to remove some of these critical pieces that are hardcoded 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 XML configuration module. Now I want to caution you, a lot of people are real quick to want to skip the XML portion of configuration, and I will tell you that there's things you can do with the XML configuration that aren't as easily done with Java. So if you're thinking, Oh, I don't want to use the XML configuration, I want to skip ahead and go straight to the Java configuration, I would encourage you to make sure that you don't do that because there's things that XML makes really easy for configuration that it's a little more difficult inside of the Java configuration section.

**Spring XML Configuration**

**XML Config Introduction**

Let's talk about configuring Spring using XML. We're going to take the sample application that we built in the previous module and wire up that application with XML using the Spring framework.

**Why Use XML?**

Why use XML? XML configuration was the first method available in Spring and still one of the more popular approaches. Some things are, in my opinion, still simpler using XML. There is a separation of concerns that organically happens when configuration code is removed and placed in a separate file. We're going to copy our sample application that we created in the last module. And create a file called an applicationContext. xml to wire up our new application. Our file doesn't need to be named that it's just a standard that's kind of become associated with Spring.

**Copy Demo**

This is where we left off form the previous module where we created a basic application to illustrate some of the problem areas or pain points that we have within development using hard code references. I'm going to go up here to our file bar and right-click and say close all. Then I'm going to come over and minimize our project so we don't have that exploded directory view. And right-click and say copy and then go right down below that in the package explorer window and right-click and say paste. And this will bring up the copy project dialog for us. Now we're going to do a couple things in here. We're going to change the name of our project to Spring\_Sample\_xml and then we're going to use the default location as well. So this will copy this project right alongside our other project, not necessarily down inside of it. So let's go ahead and click OK. And you'll see we got both of these projects out there now. We only have to make one minor change and this is an advantage of using Maven for this. Is open up our pom. xml and you'll notice inside of here our artifact ID is still spring\_sample. Now you can change it right here or you can go over to the pom. xml tab for the source and change it in there as well. So we've got the artifact ID. We can just change this to Xml and go ahead and save that. Now we've got our project in there that is a copy of our spring sample and our libraries are already setup for us. We want to go ahead and open up that application. java and right-click on it and say run as. It will run as a java application and we can verify that everything's working and our application is good and it's stable and we've made all the changes that we wanted to. If you wanted to verify that you could go change the output from the model directory but we should be okay. Now, I'm going to do one thing so we don't make any mistakes going forward. I am going to close this Spring sample application. I get a lot of people asking questions on the support forum and more often than not it's because they have made change in the wrong directory. So, I'm going to right-click on Spring sample and go ahead and say close project and that will close that up.

**Application Context**

XML configuration in Spring begins with a file that we are going to name the applicationContext. xml. It really is the root of the application configured using Spring and XML. It doesn't have to be named the applicationContext. xml. It does need to be an XML file but that's just more of a loose standard that we use that name. I've seen people abbreviate it as appContext or application-Context and this is just kind of a default that you'll see people use inside of their own Spring applications. If we use this, Spring will do things using convention over configuration by default to find and discover pieces of our application on its own. A simple view of Spring is that it's actually just a HashMap of objects and we define that HashMap inside of our applicationContext. The objects that we have inside of here are pretty much just name, value pairs. Although it's not the intention of Spring ir can be used as a simple registry and we look up those beans that we define in that HashMap out of this applicationContext file. All of our XML configuration begins here and for our sample, we're going to call ours applicationContext. You can have other files that this will reference and lookup and pull in. It's a little bit more of an advanced topic but we can import those things here. There are some namespaces that Spring developers have put together that help us in our configuration and validation of files and really make things a lot better. Using Spring STS will actually help us setup these files and make that transition easier. We will look at adding a namespace to the top of our application in our example here. But basically, you put an XML snippet at the top of your applicationContext and it knows what our bean's namespace is to help us configure the rest of our file. Let's add this file to our application now.

**Application Context Demo**

Spring STS makes it quite easy to add configuration files to our project. But we will walk through how to add them without using their built-in wizards. Since we are using Maven in our project we have a nice directory structure setup for where to put our XML files at. And that's underneath the source main resources directory. I'm going to go ahead and right-click on that directory and select new, Spring Bean Configuration file and it'll bring up this dialog for us to create that new XML configuration file. Now make sure you're under the source main resources directory as I get a lot of questions in the support forums. And fairly often it's because people have created a file in the wrong directory and your application can't find it. So let's go ahead and create this file and name it applicationContext. xml. And you have two options down here to select next or finish. I'm going to go ahead and select next just to show you what these options are on the next page and those are to choose XSD namespace declarations to go at the top of your file. We're not going to choose any of those right now so let's go ahead and click finish and see the XML file that it creates for us. You'll see up at the top of our applicationContext that we have a XML namespace defined which is the default namespace of beans and xsi, which just says this is an XML scheme instance. And then the schemaLocation. The schemaLocation is added to our XML file and is what gives us Context sensitive help inside of our application. So now if I come down here and do a left angle bracket. You'll see that my IDE offers up suggestions for me because we have those namespaces defined. So I have bean, beans, description import, and these aliases. Notice that we don't have one here called context. Context is something we'll discuss in the next module and the way we get that functionality in here is by adding that namespace declaration to the top of our project. This also helps us define well-formed and valid XML. Valid in the sense that I can put this element in this location. So if I want to throw a bean element in here I can do that and notice I don't have any errors or warnings about that location. But if I put something that it's not looking forward to in here. Something like context. It is going to let me add it but it's going to warn me that it doesn't know what this is at this location. So it's saying that it's well-formed but it's not valid XML here. So that's what those namespace declarations are at the top of our application. We'll talk more about that as we go through development though. I'm going to go ahead and delete that out of here. Now this is really what you need to get started with your applicationContext. And if you weren't going to use their wizard I would have to type all of this in there by hand. And you can do that, there's nothing magical that that wizard does for us. I'm going to go ahead and create another XML file here and you don't have to do this. I'm just going to show it for an example. I'm going to call this applicationContext2. xml and click finish. And you can see, it's just a basic XML file. Now if I want that context sensitive help though I'm going to have to go through and type all of this information in there by hand. And that's why I do like to use their wizard to do this because this just makes it a lot more painful for me to do. One other thing to note is when I save this it's still treating this like it's an XML file. Notice I just have a design tab and a source tab at the bottom of here. Because it's not saying that it's a Spring configuration file. If I go over to my applicationContext you'll see I have a source namespaces overview beans and beans graph tabs at the bottom of that file because the ID has registered this as a Spring configuration file. So if you sue their wizard although some people don't like using wizards. It does give you a lot of added features and easy just grunt work taken out of there for you just by registering it as a bean configuration file with the Spring IDE. Before we move on though I'm going to make sure I've deleted this applicationContext2 file just so it doesn't mess anything up. And if you created one too, I would do the same. So I'm going to just get rid of that file and now we have just the applicationContext inside of our application and it looks just like this.

**Namespaces**

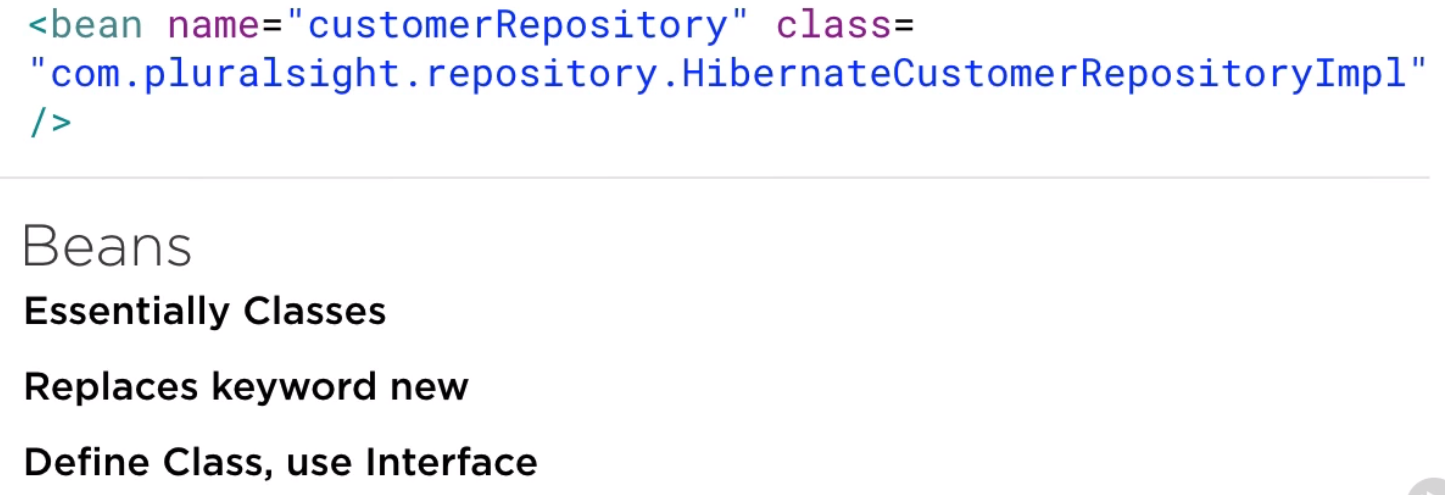
Here is a snippet of just the beans namespace definition that resides at the top of your XML file. I believe that the misunderstanding of namespaces is one of the main reasons that people are afraid of using XML and XML configuration. Spring has created this namespace that simply acts like a dictionary for the properties that we can use to create and inject into a bean. Let's dive in deeper to talk more about those properties.

**XML Declaration**

Here's the definition of a bean in XML. This bean is the customer service bean that represents where we want to put our business logic inside of our application. I have a few other properties to find in this example that we're going to walk through in more detail. But you can see the pieces of what we're creating here. Let's dive into that detail now.

**Beans**

The XML configuration is composed of beans and beans are basically classes. They're just pojos that we use inside of our applicationContext. Defining beans can be thought of as replacing the keyword new. So wherever you are using the keyword new in your application something like customer service my customer service equals new customer service Impl. Where you're using that keyword new that's somewhere you can look at removing that configuration and placing it into an XML file. Lastly, we always want to define the Class but use the Interface. And I'll show you what I mean by this in our demo that's following this. Just a quick reminder though. We discussed why you want to separate configuration from business logic in the first module. But to mention it here again we can now change configurations without re-compiling our code. We could switch from environments like dev to test and we would just use a different configuration file and not have to re-compile pieces of our application. This technique is called separation of concerns.



**Bean Demo**

Let's add the first bean into our application using XML. I want to go after the closing element of the beans declaration and our, and the file. So let's go ahead and enter in our bean definition. And inside of here, I want to give it a name and set that name equal to customerRepository. And you'll notice, I am using camelCase just like I would variable names. I alwso could use the property ID instead of name. ID has to be valid XML so I can't use some characters that I might want to inside of our name. Next thing we need to add is our Class. For our Class, if you have created your file using Spring STS and Intellige does this as well. I'm not sure about some of the other IDEs. I'll get Context sensitive help while entering this in here. So I can say com. Pluralsight and hit control space bar and it will automatically fill that in for us. If I now want to use the HibernateCustomerRepositoryImpl and this is what I was discussing earlier. That our file inside of our file we want to use the implementation but reference the interface. So as I'm defining my bean, I'm going to come out here and use the actual implementation of it as our Class. And you'll see what I mean when we go to reference this in another file. We have now created our first bean and that's really as simple as it is to define a Class inside of our Spring configuration using XML. Let's do something with this bean now.

**Setter Injection**

Now that we have a bean defined. How do we go about using it? Using injection, that is how. There are two types of injection. Setter injection 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 start with an example using setter injection and I also feel that it's better for existing code.

**Setter Injection Demo**

To show an example of setter injection let's go ahead and make this file full screen. We're going to create a new bean. We're going to name this bean customer service. And this is going to represent where we would put our business logic inside of our application. I'm going to create that bean configuration so that we can wire these two together. I'm going to use the Class com. Pluralsight. service. customerserviceImpl. And you can see we're just using the context sensitive help here inside of our application to help create those beans for us. Now, one thing we didn't talk about much earlier is that we can clean up this code by using the forward slash to close off our bean rather than having a closing ending element. So right now, we have the closing bean element. If I put a forward slash in there. It goes ahead and gets rid of that trailing bean element. And this is just if we don't have any extra properties that we need to add in there. We can go ahead and do this to clean up and save some of that space inside of our application. Now, we don't have our repository bean and our service bean wired together yet. So let's go ahead and open up our customer serviceImpl. And you'll notice inside of here we still have that hard-coded reference to our customerRepository equals new HibernateCustomerRepositoryImpl in here. First thing we want to do is get rid of that HibernateCustomerRepository hard-coded reference in here. Now we also need to create a setter so that we can use setter injection inside of this. So I'm going to right-click on the CustomerRepository and say generate getters and setters. And I'm going to de-select or un-select getCustomerRepository because we're only doing setter injection and if you're trying to go for 100% code coverage or something like that inside of your application. Or just not have unused methods, we can un-select this. Click OK and that will create that setter for us with everything defined and ready to go for setter injection. Now before we leave this file, I want to do two things. I want to right-click and say source, sort members. And I like to clean up the code because it will insert code wherever you're currently selected inside of your application. So that will organize those. And then we have this yellow caution up here on the left. And this is an unused import from that HibernateCustomerRepositoryImpl that we had earlier. I'm just going to click Control Shift O and have that remove that from that file. You could also right-click and select organize imports from the source menu as well. Now this file is setup to where everything can be injected through setter injection. Let's go ahead and click back to the applicationContext. We can now add the property element to the customer service bean that we created earlier. And we're going to do this by name. So we're going to wire these beans together by name. We're going to set the customerRepository setter and we're going to pass a reference of the customerRepository bean. I'm going to go ahead and select that text and copy it and paste it in here. And we can also use a forward slash to get rid of that trailing property element. Everything here is now wired up correctly and we have done setter injection wired by name. So it's going by the name of that setter method. I could go ahead and change the name of that bean customerRepository to foo and pass that reference in by foo. So you can see we're using the setter and passing the reference of that element in there. I wanted to show that that is referring to that bean so that you get the idea of we're calling the setter of customerRepository. Because sometimes when those three fields are all named the same, it's a little confusing as to which piece we're injecting where. So that customerRepository is now set using that bean that we have defined and it's by reference from one to the other. Now we can go ahead and open up our application. java and run this application. But before we can do that, we need to import our applicationContext. xml. So we're going to comment out that customer service bean and create a new instance of the applicationContext Class. We can use context sensitive help here to help us pull in the correct interface for that. I'm going to just call mine appContext and we're going to use some tools inside of Spring to look up this Class path file. So we're going to say new Class path XML applicationContext. We're going to pass in the name of that file just by a string. So applicationContext and then this will look on our Class path for us and find our applicationContext. xml file. And since we are using Maven it will automatically put that file in the right place. And let me show you what I'm talking about here. Our applicationContext is under source main resources. Maven will compile that into our application at the root of our application. So in our Classes directory applicationContext. xml will be copied at the root of that directory structure so we don't have to put any path information. And that's why we can just put this string in here pointing to that applicationContext file. Now inside of our appContext we want to go ahead and get a reference to the customer service bean that we created. To do this, we're going to create an instance of the customer service object and then use that appContext to find this bean. So we're going to go ahead and say appContext. getBean. And we're going to use the lookup method that we can pass the name of the bean in and the Class type, so that we don't have to cast it and suppress any errors or warnings that we have. So we're going to go ahead and put in a string here of the class name, which is customer service and then we're going to tell it that the Class it's looking up is customerService. class. And notice this is where again we are using the interface instead of the concrete Class so that we can swap out those implementations behind the scene without having to re-compile our application. So now, we have our service that we're going to go ahead and find this and lookup the first object that gets returned by first name. Everything's defined, setup, configured here, ready to go. So now we can go ahead and run our application. So I'm going to exit full screen and then right-click and say run as java application and you'll see that our application runs. Let's open up that console and you can see all the stuff that's going on here now. Our Class path, XML applicationContext has been loaded and it's gone ahead and found the bean that we've had and started up our application to where we're now able to run this application and have it pull those resources out of those bean definitions. And a lot of people get confused about what's going on in the log files here. Spring definitely dumps out a lot of logging context. But you can see that our application is wired up and producing that bean from our configuration. We've wired up our customerRepository, our customerService. We passed a reference in from one to the other. We went ahead and our customerRepository is still just the same bean that it was. We returned that instance from a repository but our customerServiceImpl we got rid of that HibernateCustomerRepositoryImpl that was hard-coded in here and enabled setter injection. And to do that, we created that set CustomerRepository method and eliminated that hard-code reference that was in here. So really, the meat of that CustomerRepository setter injection was done in this file by eliminating that reference, creating the setter. Cleaning up those imports that we had inside of here and then going ahead and wiring that up in the applicationContext that now has that pass by reference to the customerServiceImpl. Our application could now have different types of repository injected in. And we wouldn't have to change pieces in our customerServiceImpl. This is just some of the value of using setter injection and moreover, dependency injection.

**Constructor Injection**

We have seen how simple setter injection is to use. But what about constructor injection though? Constructor injection guarantees us a few nice things using simple constructs in java. But namely it is that we have a defined contract when we create each object. A positive and a negative though is that I need to have a constructor defined for each situation. I should also note though that you can use constructor and setter injection together and I often see people argue of one over the other. Use them both, they work well together. One other slight difference is that using constructor injection, it is index based and not named-based, like setter injection is. Let's add a constructor injection example into our application now.

**Constructor Injection Demo**

Here's where we left off in our application using the setter injection. I'm going to go ahead and make this applicationContext full screen. You can see where we have the property that we injected in fusing the CustomerRepository into the customer implementation. Let's change this to using a constructor argument. Now, the first thing we want to do is change this to constructor-arg. And name is no longer applicable. I'm going to switch this to being index-based equal to zero and that's our position. Now when I save this I should immediately get an error message, which I do. You can see all the red underscores telling me that it can't find this constructor. So what I need to do now is open up our Customer Service Implementation. Before we were using the default no-args that looks very simply, like this. CustomerServiceImpl. If we don't define one in java this is automatically there for us. Well, now since we have one that we're expecting it to pass in the CustomerRepository object to. We're going to say, public customerServiceImpl and we're going to pass in the CustomerRepository. And we can name this the same thing and pass a this reference to it. So when you say this. CustomerRepository is equal to CustomerRepository and save it. So now, I have gone away from setter injection and done to constructor injection. And all I had to do was add this constructor and change this line from being a property to being a constructor argument. Let's go ahead and run this code and see if it still runs the same. We can go back to our application. java right-click on it, or it should be the last thing that we ran in there, I can click the run button. And you'll notice that it still runs. Just like we would expect it to. A few things to note, when defining constructor injection as I mentioned before and I mentioned it again. It is index-based. So if I have three arguments to pass in I would start with it being index at position zero then one and two, and pass those objects into that constructor. Really no advantage other than in this example, I have to guarantee that that constructor is there. So I am forcing that contract which does give me a little more sense of security with knowing that I have initialized things the right way. I've seen people make mistakes before using Spring where they have done things in configuration for initialization and ran into problems because something was null. So I do like the options of using the constructor argument but setter injection works just as well. And you can use those two together. All we would do is go inside of our closing element and say property inside of here if we wanted to inject something else in there. We don't have any example that we're going to now. So this works just great.

**Autowire**

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 into it. To counter this, they introduced a mechanism called autowiring, for you to autowire beans together. Let's briefly go through each type and then we'll go through them in more detail on the following demos. There are four types of autowiring that you can do on a bean. The first type is by type and this allows a property to be autowired if exactly one bean of that property type exists within the container. For example, our customerRepository was a bean of a specific type. The HibernateCustomerRepository Impl. If I had two beans of the same Class but with different names, I would get an exception using this approach because it couldn't tell which bean to choose. The second option is by name. And by name fixes the problem of by type. They're both good to have though because by type will allow us to only create one instance of that Class in our container in our applicationContext. Where by name, will allow us to have multiples and choose that specific object based odd of the name. The next option is by constructor. As we saw earlier, constructor injection is very similar to setter injection. Well, constructor autowiring behaves very similar to setter injection by type. It looks for an object of that type to inject into the arguments of the constructor. And the last option is no, or none for autowiring. So if I specify no, it means that no it cannot be autowired at all. A lot of people early on tried autowiring just for testing because they were concerned about how it would perform in production I have used lots of applications with the autowiring and production. And never really ran into a problem with it.

**Autowire by Constructor Demo**

Let's take our previous example that we had used to illustrate constructor injection and change that to autowire by constructor. So first thing we're going to do is take our element that we had defined down here and comment it out. We have already setup everything we need for it to be injected into our constructor. If you remember, we added that constructor into our customer service implementation. and now all we need to do is come over here inside of our element and make sure you're in that element tags. We're right after our Class definition. We're going to say autowire equals. And it should pull up our options for us. We have constructor here. Let's go ahead and choose that. And now if we got to our application and run it you'll see that it autowires and does run exactly how we would expect it to. Now, just to verify that this isn't smoke and mirrors let's go back into our definition and get rid of that autowire and save it. When we run it again, we should get a null pointer exception, and we do. It's saying that when we're trying to reference that CustomerRepository object inside of customerService that it hasn't been defined. So you can see that it's autowriring and find it for us. It just saves us from having to go through and name out all of those elements and tie those references together. If there's only one of those inside of our application. So let's go ahead and add that back in there just to make sure we're not leaving this in a broken state. Save it. And run our application again and it's back to where it's working. Great way to get rid of a lot of busywork inside of your application. And I really like it. Haven't had any problems using this inside of a production application. Although I can see why it does instill some fear in people because they're not quite sure how things are wiring up. But that's the way we have the various options of by name, by type, constructor and no to choose what's the best for your situation.

**Autowire by Name and Type Demo**

To show you autowriring by type. Let's open up our applicationContext. xml. And we're going to go ahead and go to the autowrie section and that customerService bean. And change that from constructor to by type. Now let's say that we're going to go ahead and open up our customer service Impl and verify that we have a couple of things here first. We have a default no arguments constructor which we do have, and a setter public void set CustomerRepository. Now by type, it's going to go through and look at the arguments for that setter. CustomerRepository, CustomerRepository. And so type of object and then it's named CustomerRepository. Now let's run this, and that's really all we had to do to change this to used by type. You'll notice that it still works. Works just fine. We changed one autowiring setting to by type from customer. And we could switch it to name and it will still work by name. We run this, it works by both type and by both name. This may seem a little confusing to you. If you're not that comfortable with autowiring you may not necessarily trust that what we're doing is working. And it adds a degree of skepticism to it. If I changed this bean name form CustomerRepository to foo and then go back to my application and run it. You'll see that it breaks because there is no setter by name of foo. So we get a null pointer exception. Now to make this work by name, we can go ahead and open up our customer service impl and changed that set customer repository method to set Foo. So just looks of this, for the bean naming convention of lower case foo. It's going to have a setter of set uppercase foo. That being foo is going to tie with us, setter foo. So now, if we go ahead and run this application it will find those by the bean naming convention and line up one another. To show you this breaking the opposite direction though. If I change that Dean then back to CustomerRepository and we have that setter of set foo. If I run my application now I'm going to see this break because our code is looking for a setter named foo inside of the customer service impl. So, our bean name is CustomerRepository. We switched this to by type here in the applicationContext. xml. It doesn't care that that name is still foo and the customer service. I run it, it will find it. Now that we've seen this work both directions let's go ahead and clean everything back up. So we have our bean named CustomerRepository. Our autowiring is currently configured by type. Let's fix that method name in the customer service impl to set Customer Repository. And now we've got everything back to where it originally was stating with. The right names with the right method names. Let's go ahead and save that. We have our setters correct, our bean name correct and our autowiring method correct. With all of those changes and all the things that we've tried. Let's go ahead and go back through our application. Just verify that everything now works how we thought it should. We'll do that by going to our application java and opening that up. Running it and making sure that our application acts how we think it should. And that is by outputting the name of that Element. Everything looks good.

**Summary**

In this module, we talked about what the applicationContext. xml is and how it's the root of our application. And we looked at Bean Definition. Bean Definition doesn't just apply to XML in fact, it applies to anything in Spring. So the concepts we learned from this module are just going to apply to the other modules as we go forward. We then looked at setter injection and how we can tie that into our application and how setter injection works well with existing code that we maybe want to Spring enable. And covered, lastly constructor injection and autowiring inside of our application.

**Spring Annotation Configuration Using XML**

**Annotation Config**

In this module, we'll walk through the configuration of Spring using XML and annotations. We're going to take the sample application that we had created in the previous modules, and wire up that application using annotations configured through XML. Annotation configuration was the second method available in Spring to wire up your application. We're going to copy our sample project, and wire up our application with a little bit of XML, and most of the configuration done in Spring using annotations.

**Copy Demo**

At the end of our last module, we had two projects that we had completed. Our Spring sample app, which was our introductory project that just had basic code inside of it, and really not much else. It wasn't wired up with Spring or any of the other configuration, but it did have all of the libraries and Maven pom file that we needed to finish our application. Then we had created our Spring sample XML project, which we had wired up using Spring and just XML configuration through an application context. Here you can see, we've got our spring XML project and our customer repository, and service classes that we implemented throughout. And in the last samples we did, we had used autowiring to configure those up, so that they could be dynamically discovered through our XML classes. Now, let's do the same thing again, and create this project, but this time, we're going to use annotations to help us discover these features, and not have to go through and create these beans like we've done inside of our application context. I'm going to go ahead and exit this class, and I'm going to close the Spring sample XML file, and the associated project, so that we're not confused and copy the wrong thing. So close that project, because the one that we want to copy and configure from is spring\_sample. So let's right click, say Copy, and just come down below it, and paste that in, and we're going to name this spring\_sample\_xml\_anno, for annotation, just short for annotation. And go ahead and click OK. Now, the first thing I want to do is go inside of here and open up our pom file, and we want to change the artifact ID to spring\_sample\_xml\_anno, and save it. And now, let's just verify that this runs how it should. We can see that it's already added our Maven dependencies, and all of our class paths should be set up for us. So to test that this is running correctly, we can go ahead and open up our src/main/java, default package, and we have our application. java file. We can just right click on this and say Run As, Java Application, and it should go ahead and display Bryan at the bottom of this. So this is just making sure that everything we have in there is configured correctly, and it runs how we expect it to. So now, we can go through and start setting it up to be configured, using those annotations.

**Application Context**

Just like in the XML configuration module, we need a place to bootstrap the annotation scanner. And we do that in an applicationContext. xml. The difference being is that we just have an annotation scanner in there, and that goes out and looks for specific items. So we'll have our configuration to bootstrap and an annotation scanner, and then it uses the same conventions as the XML configuration. At the very top of our application context, we're going to have a beans schema and definition, and all of our references to our xsi and xsd to go out there and build our application up.

**Application Context Demo**

Let's add the application context to our new project before we move on to try and add any of the annotation features in there. To do this, I'm going to go ahead and right click on my source main resources directory, and say New, Spring Bean Configuration File. Now, I am using Spring STS. If you are using Eclipse or another IDE, you could just create a simple XML file, and go ahead and add the XML namespace and declarations at the top of it. But this makes it a lot easier. So I'm going to go ahead in here, and put in here applicationContext. xml, and just click Finish. I'm in my src, main, resources directory, or you can verify it up above, but let's click Finish here. Now, this file will go ahead and define our namespace and our schema locations for us. This is just our starting point, or bootstrap point, for our application. I'm going to go ahead and save this. If you aren't using Spring STS, you could copy the one we've created from the previous module, and just delete anything outside of it that's not looking like the file that I have here. So any of your bean definitions or anything like that, you could just go ahead and remove out of it.

**Component Scanner**

To start up the component scanner, we need to add a couple of things to our application context, and one of those is the context namespace. So as part of the context namespace, there is the schema defining context, and the xsd location. Just like you saw with the beans specification, we can add the context specification and schema to the top of our document. Then there's two elements we need to add out of this schema. We need to tell our application that it is configured using annotations, and then, where to start scanning for those annotations. So the first element says, hey, I'm configured with annotations, and the second element says, okay, this is the package you should start looking for these annotations in. You can drill down to a specific package, and tell it to only load things from that specific location. Let's add these to our project now.

**Component Scanner Demo**

Adding the annotation, configuration and location scanner is really quite easy inside of our application context. First, we want to go down to our namespaces, and tell it to use the context namespace. I'm going to click on the context namespace, if you don't have one of these selected, it's not going to have this namespace version up here for you, and I'm going to tell it to choose specifically the spring-context. xsd, this one at the top. If you haven't chosen this, it's going to be selected by 4. 3, or whichever version you have installed by default. I want it to be the non-version specific one, because if you upgrade, it's not forcing it down to an older version. This way, by using spring-context. xsd, if we install 4. 4, it's not going to be limited to 4. 3. Using the non-version specific one uses the latest version it can find on our class path. Once we have that selected, again, we added context and chose the spring-context. xsd, I'm going to switch back to our source tab. Inside of here, you'll now see that it's added the context URI, and the context XSD to our application. Let's go ahead and start adding the annotation config and component scanner inside of our code. By adding that namespace in our URI, we can go ahead and now say, context, and we want to do annotation-config. Now, by default, this goes ahead and gives you an opening and closing element. I want to get rid of that trailing element, so I'm just going to put a forward slash right after config, and it will remove that last element. Then we want to go ahead and add the context:component-scan. This is going to do the same thing, it's going to give me that closing element. You can leave it, but it's just cleaner to have it removed from your code. And inside of our base package, I want to do com. Pluralsight. Our application is now configured to start scanning for annotations. We haven't talked about what annotations are available to us yet, so let's go ahead and look at those now.

**Stereotype Annotations**

There are three main annotations for core Spring that help us to find components, or beans, inside of our application. They typically refer to these as stereotype annotations. The main stereotype annotations are component, service, and repository. Semantically, they are all the same. Service and repository both extend component, but really don't add any features or functionality. It's just how you think of them, and how you use them. Component is used for regular components or beans, basically, any POJO. So anything inside of our application that we want to be a bean, we mark with a component. Service does not stand for web service. That's what everybody thinks of when they first hear the word service. It's actually the tier where our business logic should be contained. So if you're dealing with a Spring MVC application, we don't want to add business logic to be contained inside of our controllers. We instead put that in our service tier, or a business logic layer, and mark that with @Service. And the last one is repository. Repository is our data access tier, or our DAL layer. It's basically where we do any of our interaction with the database. Whether we're using Spring JDBC template, or JPA, hibernate, iBATIS, whatever we choose to use, that tier gets labeled with @Repository. Let's go ahead and start annotating our code with some of these annotations now.

**Stereotype Annotations Demo**

Let's begin with our repository tier for annotating our code. I'm going to go ahead and open up our src/main/java, com/Pluralsight/repository, HibernateCustomerRepository impl. So open up our hibernate customer repository. And in between our import statements and our class definition, I'm going to go ahead and type @Repository. Now, almost any ID that you're using, you can hit Control+Space bar, and have it suggest things to you. That was the only repository that was available. If you forget that, you can do Control+Shift+O. Now, in sight of this repository, I'm going to give it a name, and this is giving us our bean name. So I'm going to do customerRepository, and save this. That's actually all we have to do, to define this as a repository bean. Let's do the same thing to our service tier. I'm going to go ahead and open up CustomerServiceImpl, and same thing, in between our import statements and above our class definition, I want to say @Service, and there are multiple services available inside of this one, so by typing Control+Space bar, the first one they suggest is @Service from springframework, and you'll notice, it's defined it as one of the stereotypes, that is the one that we want to select. And we want to go inside of here, and give this a name of customerService. Now, we actually have all of our code annotated, and discovered as beans. But before we proceed, I want to talk about one line of code in here. Now, we could go into our application context and start wiring these beans up, they're discovered through one another, but we're going to take it a step further, and actually use autowiring to configure these. So you'll notice on this line right here, we have a customer repository that's still equaling the new hibernate repository, because we had copied this demo from our Spring sample. So in our old Spring sample XML, we had wired this up with a bean, and got rid of this. We looked at constructor injection and setter injection, and then just member variable, and how we could inject in that as well. So let's go ahead and discuss autowiring, and come back to this code, instead of showing you three ways to do it, we'll show you the preferred way to go about it.

**Autowired**

A lot of people feel that autowiring and using annotations is much more straightforward. That you autowire it where the code is going to be injected at. Autowiring a method is hidden because it's tied to where you place that annotation at. It seems to be a little less confusing, and other people like the XML approach, where they see it wired up from step to step. We can autowire in three places, at member variables, constructor, and setter injection. Let's discuss each of these in a little bit more detail.

**Autowire Member**

To autowire at the member variable level, we just place an annotation on our declaration, just like we have in this code snippet here. Using the component scanner, Spring will look for any code marked with the stereotype annotations, and the autowired annotations, and wire up your application appropriately. Let's try this out using member annotations in our code now.

**Member Injection Demo**

Let's work on getting rid of that hard coded new hibernate customer repository impl. We can do this by going ahead and eliminating this code off the end, and having the @Autowired annotation at the top of our declaration. Now, you'll notice I have a yellow squiggly line telling me that the hibernate customer repository impl is no longer used. Go ahead and click Control+Shift+O, and tell it to remove that from our code. Now, our application is wired up using the customer repository from our hibernate customer repository, where we have defined it as a bean over here. Let's test this inside of our application. We need to go to our application. java, and we haven't ran this yet. It would have ran when we first copied it over, but let's now Spring-enable this. Now, we can open up our spring\_sample, and copy the application. java where we've already defined this customer service code inside of here. Rather than type it all out, I'm going to do that. So I just open up our spring\_sample\_xml application. java, and we go ahead and overwrite what we have inside of ours, in our spring\_sample\_xml annotation. So now, we can save this, and it's going to give a warning that we don't close that app context, that's fine. Let's right click and say Run As, Java Application, and you'll see that our beans are now autowired, and displaying Bryan through that configuration. So, to step through those pieces, we have our hibernate customer repository, which we annotated as an @Repository. We have our customer service impl, which we annotated as @Service, and then we went ahead and autowired our customer repository, so that it automatically pulled that bean in, here on this line, and everything was injected. There was no wiring code, if you look at our application context now, you'll see there's just the scanner and the annotation config marcation, and we're good. That's all the pieces we need to autowire our application. Let's talk about the other types of injection that we can do, using autowired inside of our application now.

**Setter Injection**

Autowiring at the setter level is also very straightforward, and in some ways, is closer to how you would develop your application without using Spring. We generate the setter and then just place an annotation at our method, just like we have in this code snippet. Like the member variable autowiring, Spring will, using the component scanner, look for any code marked with the stereotype annotations and the autowired annotations and wire up our application appropriately. Let's try setter level injection instead of the member injection now.

**Setter Injection Demo**

Let's open up our customer service impl, and get rid of this @Autowired at the member level. I'm going to just comment it out, so you can see the differences as we go through there. First thing I want to do is right click on customer repository, and say Source, Generate Getters and Setters. Now, all I want for this example is the setter, so I'm going to uncheck getCustomerRepository, and this will create the setter for me, for us to do setter injection. Very simply, all we have to do now is say @Autowired, say this, and our code is done. The setter injection's already configured, we're good to go. I'm going to go ahead and add a System. out. println to this, just so you can see that we're actually calling this method to do that setter injection. We are using setter injection, and let's go ahead and save that, and now, we can go ahead and go back to our application. java, and run this. And we should see our Bryan and our we are using setter injection inside of our application. So here we go. We've converted our application to now use setter injection, and it's very simple. Like I said, the approach with this I like in some regards because it's how we would wire up our application doing test-driven development, or test-first development. We would call these setters to build out these pieces as we go. Either way works well. I do like the member level injection, because I don't have to have a bunch of setters for no reason, but this is how I would use test-driven development, without having Spring integrated into an application. So both approaches are definitely acceptable, and it's really a matter of personal preference as to which one you like over the other.

**Constructor Injection**

Autowiring at the constructor level is quite similar to the setter approach. We create a constructor for the type of object that we want to inject in, and then simply mark that constructor as autowired. You do need to be a little bit more cautious with this approach if you switch back to setter or member injection, because you will have eliminated your default constructor. Let's try this example out now.

**Constructor Injection Demo**

Constructor injection is very similar to setter injection. I'm going to start off by commenting out this autowired annotation on the setter, and give myself some white space, to create that new constructor. I'm going to go ahead and call this public CustomerServiceImpl, and pass in an instance of the customer repository. And I'm going to call this variable customerRepository, and then, we go through and assign this. customerRepository = customerRepository. And just like the setter, I'm going to add this little debug line in there that says System. out. println, just, we're going to tell it that we're using constructor injection. Now that we have all of this in here, we have to annotate our constructor with @Autowired, and we are actually done. That's all we had to do, to use constructor injection. Define our constructor, add the assignments in there, and mark it as @Autowired. Now, we can go ahead and come over to our application. java, and right click Run As, Java Application. And you'll see that we are using constructor injection, and it printed out the statement Bryan that we were expecting it to. So all of this code is now defined inside of our application. Notice there's no beans wired up inside of our application context, so all we had to do is define it as the annotation config, and the base component scanner, and this is a great launching point for having configuration done inside of our XML, and leave our java code as similar or as exact as we could make it without having Spring in our app, but still wiring it up with Spring, so we would normally use constructors, we would normally use setters to do things, we can just annotate those, but keep some of that configuration out of there, so it's a good hybrid approach. And then, you can see of course what we did to our CustomerServiceImpl, just to show the various methods. We did member level injection, we did constructor injection and setter injection, to go through and show how we wire these things up. But notice how you still don't have that customer repository, the actual implementation defined anywhere in here. So we have that true loose coupling, and we're doing everything through the interfaces. Even when we access our customer service inside of our application. java, we're doing that through the interface and not doing that through the implementation. So we really get that loose coupled code, where we can inject in whatever configuration we want inside of our application.

**JSR-330**

I wanted to mention just for a second the JSR-330 specification. Java started to see the benefits of dependency injection, and decided to incorporate into its core some very lightweight simple dependency injection. JSR-330 is not as feature-rich as what Spring provides. It is just a simple dependency injection specification for Java. What it does is beyond the scope of what we're covering in this course, even though it isn't very feature-rich. It has simple annotations, but does not provide to us what Spring does. So why use Spring? It provides a lot more than just this basic dependency injection, and it's a lot more feature-rich libraries to help us make things easier. So Spring is all about making our life easier, by coordinating and helping orchestrate different libraries together, to work better.

**Summary**

To recap what we did in this module, we learned that we still needed to have an application context, but it was just to bootstrap our component scanner. The component scanner is what was used to go through and look at all of our package structure and classes, to figure out which annotations should import, and how to assemble our application. Then we went through our basic stereotype annotations, component, service, and repository, and talked about how they get used at each tier inside of our application. After that, we wired up our application using autowiring, and showed the different autowiring characteristics of setter injection, constructor injection, and the member or field level injection. We briefly talked about JSR-330, and how it is a simple implementation for dependency injection in Java, but it's not as feature-rich as what Spring provides us. Next, we're going to look at configuring Spring using just Java, and no XML inside of our application.

**Spring Configuration Using Java**

**Java Configuration Introduction**

In this module, we will walk through the configuration of Spring using just Java. We're going to take the sample application that we had created previously and wire that application up using just the Java annotations and the Java Configuration loader. Java Configuration in Spring is the latest method available for wiring up your Spring application. It was introduce simply because some people don't like mixing XML and source code together for configuration. Like in previous modules, we're going to copy our sample project and then wire it up using Java and a few annotations to show you how all of this works together.

**Copy Demo**

Let's go ahead and get our project copied so we can begin to do the Java Configuration. I've gone ahead and closed all of the other projects out and I have a filter set so that it only shows open projects. Let me go ahead and right-click on spring\_sample and say Copy and then go down below on the Package Explorer and say Paste. This will bring up the Copy Project dialog. I had to change the name to spring\_sample\_java and click OK. This will create the project for us just like we've done in the other samples that we've looked at. The only thing we need to do in here is now open up that pom. xml and change the Artifact Id to spring\_sample\_java. Now we can save this. The reason why we're doing this with the Artifact Id is just so that there's not the same artifact being generated by a Maven with the same ID. We have different projects with the same ID. Unless we'll go ahead and make those unique projects, it really isn't an issue because we're not deploying this to a web server or anything like that that's going to make a difference. But that's why we are changing that Artifact Id each time. We should now be able to open up the Application. java file underneath the src/main/java and right-click on it and say Run As Java Application and it should output Bryan. It does, so we know that our application is set up correctly. Now we can go ahead and start doing the Java Configuration inside of our project.

**AppConfig**

The first thing you'll notice about the Java Configuration is obviously the lack thereof an ApplicationContext file. We have no applicationContext. xml. Early 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 to see less or no XML at all. So, enter the Java Configuration. Almost everything in Spring can now be configured with just pure Java Configuration. This was not necessarily the case when this course was first released but since we've re-recorded this course, everything has since caught up now. Instead of not having a ApplicationContext, we now create an app config file and start our configuration in there. Let's go ahead and add that configuration to our project now.

**AppConfig Demo**

We need a file to bootstrap our application config for our application here. We were doing that before in our applicationContext. xml. Using Java Configuration, we're going to go ahead and right-click on this default package alongside our Application file and say New, Class. Inside here, we can go ahead and name it AppConfig and that's really all we have to do. It's public. There's not a main method inside of here that's contained in our Application. java file and we go ahead and click Finish. Inside of here, there's nothing special going on just yet, and that's what we like is that it's just plain Java code. Let's look at the annotations that we need to add to this though to make this a configuration file.

**@Configuration**

To replace our XML configuration, we're going to use an @Configuration annotation. The Java files that have the @Configuration annotation replace any XML context files that we have previously used. Configuration is done at 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 can 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. We add a method level annotation for @Bean and it looks similar to this where we're saying what this method returns is an instance of a bean, and that bean is now registered inside of Spring and available for us to use in our Spring application. Classes and method names can be anything and Spring 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 add this to our AppConfig now.

**@Configuration Demo**

As we noted earlier, to make this file a configuration file, we simply just have to add the @Configuration annotation at the top of this class. I'm going to go ahead and import that just using Control + space bar. This will tell Spring as we run this code to go ahead and look for any configuration information from this file. To create a bean, we're going to do something quite similar and that we're just going to create a method and call it as public CustomerService. Let's go ahead and return the interface here and I'll use Control + space bar to import that. I'll call this getCustomerService. This doesn't take any parameters in here. We're just going to return a new instance of the CustomerServiceImpl. This is where we are implementing an instance of the concrete class but returning the interface. Let's go ahead and organize all of our imports, so I'm going to click Control + Shift + O to have it pull in anything that we don't have yet, and let's finally mark this as a bean. I'm going to say @Bean, Control + space bar, and you'll see the import for org. springframework. context. annotation. And inside of here, we can give it a name. I like to name these just as I would variable names, so I'm going to call this customerService and I'm going to use camelCase, the lowercase C, capital S. This looks just like it would any instance variable inside of a class. The reason I like to do this is it follows the bean naming convention and as we use autowiring later, this will already be named for the appropriate method that we're going to inject it into. Let's look at setter injection now and see how we want to tie this in using setter injection inside of our application.

**Setter Injection**

Setter injection using the Java Configuration approach is really as simple as a method call. A lot of the mystery of dependency injection just goes away. A lot of developers I talk to are concerned with the black box FILO dependency injection. With the Java Configuration approach, it is more transparent as to what is going on. With XML, there's a lot of wondering what's wired up and what's calling who and how this autowiring is all taking place, and a lot of that just seems to go away whereas more visible using the Java Configuration approach. Setter injection is simply a matter of calling the 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. Notice though that we have setter injection in here. As we are building our bean, we're going to call the setCustomerRepository method on our customerService instance. To do that, we need to have a method called getCustomerRepository that returns the CustomerRepository bean. This is going to wire the CustomerRepository inside of our customerService. We would define a bean such as our CustomerRepository here that returns an instance of that bean and you can see those methods now line up. I'm going to call the getCustomerRepository method that registers it as a Spring bean and that's going to return that instance for our setCustomerRepository on our service instance. Take note though, Spring is still doing a lot of things behind the scenes when a bean is registered. Such as these beans by default are a singleton and will only execute the method the first time it's called and then returning the cached instance after that. Let's implement this in our code now.

**Setter Injection Demo**

Since this was a copy of our original spring\_sample project, there's a couple of things we need to do to get this set up to utilize setter injection. The first thing I'm going to do is open up our CustomerServiceImpl and change the hardcoded instance in here to be passed in through a setter. I'm going to get rid of that new HibernateCustomerRepositoryImpl assignment and I'm going to right-click on customerRepository and say Source, Generate Getters and Setters. Then I'm going to uncheck getCustomerRepository just because we don't need it for what we're doing. We just want to do setter injection. I'm going to go ahead and save this and I'm going to go ahead and type Control + Shift + O before I leave here to get rid of any of those unused imports that were in this file. So we had that HibernateCustomerRepository in here from before. Let's go ahead and clean up those imports. Now let's switch back over to the AppConfig and we need to do a couple of things inside of here. First, I want to go ahead and create a new bean down here. I'm going to say @Bean. I'm going to give it a name equal to customerRepository. I like to follow the bean naming convention as I mentioned before where I have it camelCase. Lowercase C, uppercase R for repository. Then I'm going to create a public CustomerRepository and for the method name, as I mentioned before, it can be anything you want but I like to follow the standard convention of the bean naming convention here as well. This just helps as we create autowired methods in the future to not have it trying to guess which methods tie into which bean that we're trying to inject. Now, we can say return new HibernateCustomerRepositoryImpl and save that. Before we go much further, we need to now get rid of this hardcoded instance here of the new CustomerServiceImpl and create a variable that we now do setter injection on. So I'm going to say CustomerServiceImpl service equals new CustomerServiceImpl and we want to go ahead and say service. setCustomerRepository, and now we're going to use that getCustomerRepository method that we just created. You might be asking yourself at this point how is this any different than just calling the setter and the getter on our own not using Spring? As we mark that as a bean, Spring will now look in its repository and its context to see if there's any other beans already created with that name or of that instance and inject those in here. It may not actually create a new instance of that bean every time. We get that power of Spring by just having that annotation here. Even though we call it the same way in our code, behind the scenes, it's looking to see if it is a Spring bean and it's already been injected somewhere else. Now we can go ahead and return that instance of our service and save this file. Everything looks good here. One last thing before we run it is we need to tell our application how to look up beans that have been configured in a file like this. I want to go ahead and open up our application. I'm going to make this full screen just for real estate on here and we're going to create a new instance of the ApplicationContext, and we're going to call this appContext. This is where Spring wins the award for the longest class name ever known. I'm going to say new AnnotationConfigApplicationContext and we're going to pass in AppConfig. class and save that. I'm going to bring this down to a new line so that it will all fit on the screen and you can see what we have to tell it now instead of looking for this XML file to go ahead and look for this AppConfig class, but all of our configurations are now done in a class rather than using any XML. As we've done in the past, let's go ahead and tell it to look up this instance of CustomerService to not use that hardcoded reference in there. We'll say CustomerService service equals appContext. getBean. I want to use the string for customerService. And we also want to use the instance that it goes ahead and passes in the class so that we don't have to cast that afterwards. We'll say CustomerService. class. We can save that. Now, we have everything tied in here to go ahead and look up our new configuration as well as look for our bean inside of that configuration. Let's run this and see if everything works how we think it should. I want to right-click on this. I'm going to make it towards not full screen so you can see the output while we run it. Right-click, say Run As, Java Application. You should see that it outputs Bryan as well. A few changes that we had to do there. Let's recap that a little bit. Inside of our AppConfig, we've now added our configuration annotation. We've marked beans as beans but you can tell by looking at this Java code, that's a lot closer to what we would do without using Spring and that's one of the powerful features of using the Java Configuration is I don't have it looking like I'm doing some stuff in XML and some stuff in Java. This looks very similar to how I would go about doing it without using Spring at all for getting that new instance passed in there.

**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 that we have for that instance. Our bean that we had written before where we called getCustomerRepository would now be used in the constructor of that bean. Very simple. 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 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 around objects. We can pull them from the framework using the bean and the getBean aliases with these bean names that we have defined on our objects.

**Constructor Injection Demo**

To convert this to constructor injection versus setter injection, I'm going to begin by opening up the CustomerServiceImpl and create a constructor inside of here. I'm going to say public CustomerServiceImpl and I'm going to pass in an instance of CustomerRepository and I'm going to follow the bean naming convention and pass in the camelCase variable name of repository. We're going to assign that to this. customerRepository equals customerRepository. Now, this class is now set up for constructor injection. When I save it, it's going to break our AppConfig. The reason why is we got rid of our default constructor. That's fine because we're doing constructor injection now. We're going to click over into AppConfig and we're going to change this. To start with, I want to get rid of this service. setCustomerRepository and I'll leave it there for reference so you can see it. And just inside of this constructor, we can now say getCustomerRepository up here. Now this is converted to setter injection. I got rid of the setter here or just commented out the setter, rather, and added that constructor over here. Now we're ready to try and run this. I'm going to move over to our Application. java and right-click and say Run As, Java Application. You'll see that it runs as well, so we're now running that through constructor injection. Really simple. You can see inside of our code that all we had to really do is besides create that constructor is just convert our bean configuration to now take advantage of that. I just used the getter that we had in there before. It works great. We don't need to store instances of these because Spring will register them in the ApplicationContext and as we call it, it will be efficient for us. I see a lot of people try to get crafty with doing some initialization in here and caching some things. That's what Spring's doing behind the scene for us. So by registering as a bean, every time we call that getCustomerRepository instance, it only gets ran the first time and returns that cached instance for us from every time after that. Just set up the code how you would normally run it, Spring will do all of that stuff for you.

**Autowired**

Since it was only one more line of configuration to use autowired inside of the Java Configuration project, I opted to just include it in this module rather than create a whole separate module just for demonstrating autowiring. To autowire our applications using Java Configuration, we just need to add the @ComponentScan annotation that looks something like this. The text inside of the annotation, com. Pluralsight just says that this is where I should be looking for @Autowired annotations. Very similar to the component scanner that we had set up inside of our XML project. And also like the XML configuration, you just mark whatever you want as autowired by name--or by type. One thing I like about the Java Configuration over the XML Configuration is that using Java, I can mix pieces that I want and it feels more natural. I can have a bean that I defined and autowire another bean into it. It makes more sense where these beans are coming from, and we'll show you what that looks like in this demo now.

**Autowired Setup Demo**

To configure our code to use autowiring, it really is as simple as saying @ComponentScan at the top of our configuration file and then passing in using the array string syntax of com. Pluralsight, thus being the package name of the code that you want scanned for autowiring. That's it. That's all we have to add to our code to get it to go through and start scanning for annotations and inject those in. Let's walk through a couple of different scenarios and different variations in how you can use autowiring inside of your code. Right now, this is configured to run and it'll go through and look in any code underneath com. Pluralsight for things that it wants to autowire up. Let's go back to our CustomerServiceImpl and start with autowiring a few things there. Before I make any changes though, I want to add a few pieces of System. out. println just debugging statements, just some logging statements in here so you can see what's going on. So I say System. out. println and we'll just say "We are using constructor injection. " Now that we're inside of here, we can go ahead and run this guy, and this is, we haven't made any other changes to our code so far. We did tell it to start scanning for components but as far as our configuration's concerned, right now it's still just using the configuration that we had before. Let's go ahead and run our application. We're going to say Run As, Java Application. You'll notice inside of our console here it says that, "We are using constructor injection. " That's fine. We expected it to print that out. We don't currently have anything set in here for our setter injection. Let's copy that, go in our setter method and paste that in there and change this to saying setter. Now, we also need to add a default no arguments constructor. We got rid of that in the previous demo. To convert this back to setter injection, I'm going to add that back in here. We'll say public CustomerServiceImpl and we'll just open and close and save that. And go back to our application and if we run this, we're still using the constructor injection that we had defined before. Let's switch to our AppConfig and go back to the line that we had previously. We want to say CustomerServiceImpl and we want to use the no arguments constructor. Say service equals new CustomerServiceImpl. Now, we can go ahead and go back to using this instance we had before of the service. setCustomerRepository using that method. That's why I left that instance there for us before to use so we can just uncomment that. All I did was convert it from using the constructor injection to using the setCustomerRepository and it calls that getCustomerRepository instance. Let's save this. Open up our Application. java and right-click and Run As inside of here. It should be converted to using setter injection. We'll say Run As, Java Application, and we'll see that we are now using setter injection. Just to walk through those steps again. I added the default no arguments constructor back in, converted our config to now use the setter but we aren't autowiring anything in there. Why did I have you do all this work? So now we can go through and show the variations of using autowiring.

**Autowired Configuration Demo**

Let's go ahead and open up our CustomerServiceImpl back up and go up to our CustomerRepository and say @Autowired. I'm going to use Control + space bar to import that. Save this and go over to our AppConfig and comment out that service. setCustomerRepository, and save that. We're not setting the CustomerRepository. It's going to be autowired in there. Let's go back to our Application. java and right-click and run this. We'll say Run As, Java Application. You'll see that our application runs. Let's go ahead and convert this over to setter injection. To do so, I'm going to open up our AppConfig and you will see inside of here that we now have our default no args constructor. We would, not using autowiring, uncomment that, but let's go ahead and now open up our CustomerService and move that from the method level @Autowired to the setter. We'll go down to this line of code here and paste that @Autowired in there. You'll see that when we run this, it's going to tell us we are using setter injection. Let's go back our Application. java and run this. Now you'll see that we are using setter injection statement gets exported to our console. Really powerful. We can start using all of those features. But if you look at the Java code, it looks similar to how we would write it without using Spring. Now when we start autowiring things in there, it gets a little bit more cryptic or other things get passed in behind the scenes, but still, this is just Java code that's taking place inside of our application. I want to take it a step further even and really start to show you some of the power of autowiring. Let's comment out this CustomerRepository bean altogether and discover a HibernateCustomerRepository bean and have that autowired in there. I'm going to open up our HibernateCustomerRepositoryImpl. If you remember from the previous module there were some things called stereotype annotations. We talked about those, and one of the stereotype annotations was @Repository. I'm going to go ahead and mark our HibernateCustomerRepository implementation as an @HibernateRepository. I've gone ahead and left this as setter injection and I have this CustomerRepository instance out there. I do like to name these. So let's go ahead and call this customerRepository. It's not necessary but I'd like to do that just from a good practice. I'm going to go back and run our application now. Let's go back to our Application. java, right-click, and say Run As, Java Application. You'll see that we are still using setter injection but we have completely got rid of that bean from our application config where it was defining that CustomerRepository, and rather let it be discovered through that scanner here. That scanner we defined at the top of our AppConfig, component scan that looks at com. Pluralsight, you'll notice this HibernateCustomerRepositoryImpl is under com. Pluralsight and @Repository is one of the beans that's automatically defined. Now our application will go, "I've got a bean that will work for that, I'll go ahead and inject that in your code. " We come over into our CustomerServiceImpl and tell it, "Hey, I'm a bean that accept autowired instances into me. " You can see right here, we are now injecting that in. We can take autowiring even a step further if we want to and go back to our AppConfig and comment out this configuration altogether if we want to. I want to show you the true power of autowiring. There is no beans to find here. I'm going to go to my CustomerServiceImpl and I'm going to use one of the stereotype annotations at the top up here of @Service and I want to pass into here, you'll notice that's one of the org. springframework stereotype annotations. I want to pass in here the name of customerService. The name of this bean is customerService and it's a stereotype of @Service. I honestly could use @Component because it doesn't do anything over service, but it's a good practice to get into annotating them as the types they are. If I go back to my Application. java and run this, it should run and autowire our entire application. And you'll see that we are still using a setter injection but if you look at our application config, there is no bean defined in here and everything is done using autowiring. Our CustomerServiceImpl, we just say that you are a CustomerService bean and we autowire in that CustomerRepository. You don't have to. Defining those with the Java code exactly how we had done before works just fine. That's why I commented out so you can see the difference. But I wanted to show you the true power of autowiring inside this application.

**Summary**

To recap what we did in this module, we learned that instead of an ApplicationContext, we configure our code in Java using an AppConfig Java file. We use an @Configuration annotation to define files that contain configuration code and we can just define a bean using the bean annotation. After setting up the configuration of beans, we then walked through setter and constructor injection using these configuration approaches. 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. Lastly, we looked at autowiring using this approach and saw a bunch of variations of how powerful autowiring can be but you don't have to use that if you don't want to, and we left that commented code out so you can compare and contrast the two alongside one another. Next, let's look at some of the Spring bean scopes that you can use while working with inside your code such as singleton and prototype.

**Bean Scopes**

**Bean Scope Introduction**

In this module, we will discuss the various scopes that are available inside the Spring framework, and how you can use those while configuring your beans. Bean scopes are a very important part of the bean life cycle, and we will show you how you can use the different scopes to get different behaviors from your application. We will be editing the existing projects that we have created, instead of copying them and creating new ones.

**Patterns**

Although beans scopes and patterns are not the same thing, they usually go hand in hand. Spring implements a lot of patterns for you, which is good because it helps you to avoid a lot of the common mistakes and pitfalls made when implementing patterns on your own. If you aren't familiar with design patterns, I recommend watching my design patterns in Java courses on the behavioral, creational, and structural design patterns.

**Scopes**

There are five scopes that are available inside of a Spring application for us to configure a bean for use. Valid in any configuration is a Singleton, which is actually the default, and if you haven't heard of a singleton before, we're going to explain that in more detail in the next demo. The other scope that is available in any configuration is the Prototype scope. Other configurations that are available only for beans in a web-aware project are Request, Session, and Global.

**Singleton**

The singleton design pattern restricts the instantiation of a class to 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 to the default scope of singleton, which means that there is only one instance per spring container or context. A lot of people will say that means one instance per JVM, which is typically true, but you could possibly have more than one Spring container spun up in your JVM. So really, the correct terminology is one instance per Spring container.

**Singleton Java Config**

The configuration for adding a scope using the Java configuration approach is quite simple. We just need to add an @Scope annotation to our code, as you can see in this snippet here. In the previous release of this course, we weren't using Maven, and we were required to add an AOP jar separately. Since we are now using Maven in this course, it is already a transit of dependency and is available in your project. Let's add the scope to our code now.

**Singleton Java Config Demo**

To demonstrate adding a scope instance to our class, I've gone ahead and closed everything else and just opened up our spring sample Java application. I'm going to navigate down to src/main/java, com. Pluralsight. service, to the Customer Service Implementation. Inside this Customer Service Implementation, I'm going to go ahead and create a new line underneath our @Service annotation, and add the @Scope annotation. Now from here, you can hit Ctrl + Spacebar, and have it imported, or do Ctrl + Shift + O. Either way, we want to make sure that it's imported the org. springframework. context. annotation. Scope for that scope annotation. It should be the only annotation of type Scope that is available on your class path, but just to verify that you have the correct one in there. Now I want to go ahead and type in here singleton. If I didn't this text in there, it would default to singleton, but we do want to illustrate that we can specify different instances in there, so we'll go ahead and add singleton inside there for the type that we want. Now before we run this, I want to go ahead and open up our application, and inside of our application, I'm going to do two things to show you that we are, in fact, getting a singleton. The first thing I want to do is add a System. out. println that will go ahead and print out the object address of service. Now, what this will do is it will just print out that address, so if we type CustomerService again, and another System. out. println, we will go ahead and see that we get the same object back by the same address each time we run this. Now that we've added grabbing another service instance and another System. out. println in here, let's run this application and see what the output looks like. I'm going to right click, and say run as Java Application, and you'll see that we get the same object address each time, the CustomerServiceImpl@10c2d26. Now, your object address will be different when you run it because it's going to create that new instance each time the JVM runs, but you can see that we are getting the same object back, even though we've created two service instances, or gotten two service instances back from that container. That is, in effect, a singleton. It's giving us the same object back.

**Singleton Java Config Using a Constant Demo**

I have had people comment in the past about not liking having just these raw strings inside of our annotations, and you can definitely create your own enum, or your own static constants to have the string in there and use that inside of this scope annotation, but I do want to point out that Spring does have a public static final string called the ConfigurableBeanFactory, that is available for you to use inside of your code for that string. So, we have this ConfigurableBeanFactory. SCOPE\_SINGLETON, that's available for us to use. You see, I just imported it there, and I'll go ahead and run our application, and it'll still work just the same, but it's kind of a long class name, and you may want to make your own constants file or enum for that anyway, but there is one available for Spring for you to use if you don't like having that string jut in your code saying that it's a singleton.

**Singleton XML Config**

The configuration for adding a scope using XML is just as simple as what we did in Java. We simply add the attribute for scope to our XML declaration, and you can see that we have it in this code snippet above here already. Unlike the Java definition though, the AOP jar is not required in the XML declaration, but Spring keeps it as a transit of dependency in Maven anyway. Let's add this functionality to our XML project now.

**Singleton XML Config Demo**

To demonstrate the scope attribute inside of our XML configuration, I've gone ahead and closed all the other projects, and just opened up Spring sample XML project. I'm going to navigate to our src/main/resources directory, and open up our application context. I'm going to make this full screen so you can see this attribute a little bit easier because it is over on the end of our declaration. I'm going to go inside that closing angle bracket, and just add the scope attribute, and set that equal to singleton. You'll notice as I'm doing this, the type ahead will go ahead and offer up these various scopes for me to choose and select for my application, so I'm going to arrow down and select singleton. Let me go ahead and bring these other attributes over so you can see that more in the center of the screen. That's all I have to do to add the scope to this XML declaration. Now, I've mentioned it a couple of times now, but I'm going to mention it again. All beans by default are a singleton, so our customer repository that we have defined above, that is already a singleton by default because we haven't told it to be anything different, but like the Java configuration example we did, let's go ahead and run this application to show you that that is in fact what it's doing. So, I'm going to open up our Application. java and do the same thing that we did before. I'm going to go ahead and add a System. out. println here that we print out the object address for Service. And then like we did in our other example, I'm going to copy this customer service and the System. out. println, and paste it below here to demonstrate a second instance of that coming out of our container. Make sure you change the object address down below here just to verify that you are, in fact, getting the same object twice. Now that we have that customer service being pulled out of the container twice, and printing out the object address twice, let's go ahead and run this again. I'm just going to right click and say run as Java Application, and it should print out the same object address, which is exactly what we were expecting it to do. To show you that this is, in fact, working through, let's now go look at the prototype request scope and see how we get a different object with each request that we ask for from our container.

**Prototype**

The prototype design pattern guarantees a unique instance per request. Each time you request a bean from the container, you are guaranteed a unique instance. It is essentially the opposite of a singleton. Since the configuration of a prototype is nearly identical to a singleton, let's forego the code snippets, and just dive into the code to see how it works.

**Prototype XML Config Demo**

I still have our XML project open from our previous example, so let's start there. I'm going to go back to our applicationContext. xml, and just go to where we defined the scope attribute, and change that to prototype. That's all I have to do to change this from a singleton to a prototype, and what this will do is guarantee me a unique instance with each request. Let's exit out of fullscreen, go back to our application, and we already have our configuration where we're asking for two different service objects, and printing out those object addresses, and let's go ahead and run this. Now, you'll see that we have two different object addresses coming down below. We have 6451e down here in the console, and 1494225. Again, yours will be different addresses because it's going to change with each time that you run this container, but you see that each request now gets a different object address from this.

**Prototype Java Config Demo**

To show the prototype configuration in the Java project, I've gone ahead and closed our XML project, and opened up our Java project back up, and I'm going to navigate down to our service object, and go to where we defined our scope annotation, and change this from SCOPE\_SINGLETON to SCOPE\_PROTOTYPE, and there is that public static final string that's in the ConfigurableBeanFactory again that we can go ahead and utilize. I'm going to save this, and open our Application. java file backup, and as you can see in here again, we have our two service objects that we're printing the address out on. Make sure you change that second System. out. prinln to be service2, and we can run this and verify that we're getting different object addresses each time we request them. Let's right click and say run as on our application. We're going to run that as a Java Application, and just like with the XML configuration, you can see we get different object addresses. If you aren't using that public static final string, and are still just using the strings inside your code, just to show you that it works just the same, we will go ahead and add that string inside there, and run that code as well, and go back to our Application. java, and you can see that it works with that approach just the same. Either way, it's very easy, but look at the power that Spring provides you with these annotations. I can go ahead and change the annotation from singleton to prototype, and I can completely change the behavior of my application. I don't need to be an expert on design patterns, but I can utilize all the power of them with some basic configuration inside of Spring.

**Web Scopes**

Web scopes are beyond what we're going to cover inside this course. They are covered more in the Introduction to Spring MVC course that's available here on Pluralsight, and that's simply because we just 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 a prototype, except it's for the life cycle of a request, which is fairly short, but longer than the prototype, where it's one instance per every time I ask the container for a bean. The Session scope just returns a single bean per HTTP session, and that will live as long as that user session is alive, so if we've set our timeout to 10 minutes, or 20 minutes, or 30 minutes, however long they are alive on that website, a bean of scope session will live that long. And then lastly, there's GlobalSession, which will return a single bean per application, and it will be shared across all users on that application, so once I access it, it's alive for the duration of my application.

**Summary**

In this short module, we discussed the various scopes that are available inside the Spring container, mainly the Singleton and Prototype scope. We showed how they can both be configured using Java configuration or XML configuration, and then we mentioned the web scopes that are available, but they are outside the scope of this module, and really outside the scope of this course because we have to set up an entire web application to demonstrate them. Those other scopes are Request, Session, and GlobalSession. If you want to learn more about them, there are courses on Spring MVC on Plurasight's website, and you can see how to implement those, and the different types of mechanisms for those in those courses. Next, let's talk about how we can read in properties files inside of our application, which is an incredibly powerful and useful thing when we're trying to separate out configuration and our business logic.

**Properties**

**Properties Introduction**

In this module, we will discuss how to import properties files using the Spring framework. Properties files are a great way to abstract out values that can change with each environment. Oftentimes, we have different things like passwords, URLs, and various connection information that needs to be extracted from our source code. Using properties files, coupled with Spring, is a great way to abstract out that configuration information. If you've ever had to recompile your source code to move from one environment to the next because something changed that was hard-coded inside your application, you're aware of what I'm referring to when I say different environmental information. Let's look at that configuration now.

**XML Config**

The XML configuration for loading a Property file into our application is quite simple. I, personally, feel that the XML configuration is simpler than its Java configuration equivalent, but they're pretty close in complexity. To load a Properties file, we simply just need to declare a property placeholder in our XML. Behind the XML snippet, this creates a property placeholder configurer that is loaded from the location attribute. So, in this case, our app. properties file that we will create inside of our application. To use these properties, we now simply just utilize the dollar sign curly brace (${) syntax, to inject that value into our bean. So you can see in this example here, we've used ${dbUsername} to pull a property out of our properties file, and inject that into a value in our "HibernateCustomerRepositoryImpl". Likewise, if we are using annotations, instead of XML wiring we can use the @Value annotation to load that property, and inject it into a field inside of our code. Let's implement this in our application now.

**XML Config Demo**

To add a properties file inside of our application, and have it load it, I've gone ahead and closed all of our other files out. And I'm going to open up our Spring sample XML application. Inside of here, I'm going to start off by creating the properties file just so we can see what we're going to try and load. Go to our Source, Main Resources directory, and right click on it, and create a new file. I'm going to call this file app. properties, I'm going to click finish on this, and inside of here, we don't have anything yet. So let's go ahead and create a dbUsername property, and we can set it to whatever we want because we're just going to use it as a sample. I'm going to create mine as mysqlusername. Again, we're just going to use this as a dummy data, just to show that you can import these in, but you'll get the idea from it. So now that we've got our properties file here, I'm going to open up my application context. Now, inside of our application context, currently this is what it looks like. We have our customerRepository bean, and our customerService bean. If you're not seeing these here, it's because you're probably in the XML annotation project that we did. I chose the straight XML project, because we still had these beans configured here, and I wanted to show you how to hardwire it first, and then how to do it using annotation second. So to begin with, I need to come in here, and go down to the namespaces tab. And choose context. And inside of here, you're probably going to get an error saying it's going to add more tabs to the bottom of your file as you do this, that's fine, go ahead and click okay. I'm going to click on context and choose the namespace without a version number tied to it, and it just does this so that we have the most recent one that's available, and it will do that by default on how our jars package. So I like to use the ones without a specific version because I've had times where we've removed a version, and it's caused me problems down the road. Let's go ahead and click back to our source now, and you can verify that we've added that context inside of our namespaces up above. So now, what we want to do is define our context, and I'm going to go ahead and use the context-sensitive help here to say context:property-placeholder, and I want to define the location of our file. Now, this does take a folder structure into account, and we're going to have ours at the root of our class path since it's in our Source, Main Resources directory. I'm just going to go ahead and type app. properties here. And that's it. This has gone through and will load up a properties file from our class path, and make it available for use inside of our application. Now let's go ahead and start using that though, we don't have it really doing anything yet. I'm going to get rid of this closing slash here, and go ahead and add a property to our bean here, that we're going to inject that value into. So, I want to go ahead and create a new property, and I'm going to call that property dbUsername, and right now it's going to give us a red line, because we don't have that in there. So I want to create a value, and I want to inject in here at the dbUsername but I want to use the ${ syntax. So I want to say that value, and then inject in dbUsername. And that dollar sign curly brace is what tells it to go ahead and pull that property out of our properties file, and inject that into our code. So now, to get rid of that red curly brace, I need to go ahead and open up our HibernateCustomerRepositoryImpl, and create that field inside of here. So I'm going to now say, private String dbUsername; and I want to right click on that and create a setter. So I'm going to right click and say Source, Generate Getters and Setters, and I just want to select the set dbUsername, and click okay, and save that. Now, if I switch back to my application context, you'll notice that that red underline has gone away. Go back to our HibernateCustomerRepositoryImpl, and add a field inside of here to just do a System. out. println little debugging code, and we're just going to dump out that dbUsername and have that display on our console as we run our application. So now, we have everything in there that we need to, I'm going to go ahead and go to my application, right click on it, and say Run As, Java Project or Java Application, and you'll see that it has now printed out my mysqlusername for us on the console from our application. So it's really quite simple, we created this app. properties, we have our application context that we added that context namespace to, and then injected that value into it, and then we created a field inside of our HibernateCustomerRepositoryImpl to store that value, and displayed that out to our console. Pretty simple to run this, and it's a great way for us to have those values extracted outside of our application, but be able to still inject them in, and not have them hard-coded.

**XML Config Injection Demo**

To inject that value in using annotations, instead of hardwiring our XML, we only need to make a couple of small changes. The first thing I'm going to do is get rid of this set dbUsername we just created, and come up to our field and say, @Value, I'm going to import the Spring framework beans factory annotation, and inside of here, we're going to tell it which instance we want to use, using that dollar sign curly brace syntax. We're going to say $dbUsername, and save that. As far as our Java file is concerned, that's all that needs to be done here. So I got rid of the setter, and added the @Value annotation. Now, I want to change two things inside of our application context, so I'm going to switch back over here. Since we're no longer hardwiring that in there, I can get rid of this property, and if you don't like that closing, trailing bean, we can come up here and get rid of that and just use the slash, angle bracket /> there to get rid of that. But I also, since we are switching this application now to being annotation-driven, if you'll remember back to the XML annotation demo, we did this exact thing. I have to now tell it that this specific one is configured with annotations. So I'm going to tell it that it is a, context:annotation-config. So, load those values in through annotations. Now, we're all ready to run this. Let's go ahead and get back out of full screen, switch back over to our application. Java, and run this, and you should see that our mysqlusername is in fact imported in there, and it's done so using an annotation. If you wanted to, you could go ahead and just attest yourself, put this same property code, and that same value annotation in your Spring sample XML annotation project because it already had the annotation config set up in here, but to keep it so that we didn't have to configure our HibernateCustomerRespositoryImpl bean in both of them, and make these minor changes across both projects, I just chose to do it in our XML project. It'd be a good sample to for you to try out on your own, though.

**Java Config**

The Java configuration for loading a property file in your application is very similar to the XML configuration. I feel that you see a little bit more of the detail of what is being created when you do this using Java configuration than you do with the XML equivalent. To load a properties file, we simply just need to declare an annotation for the property source. This tells the container to look for our app. properties file that we will create. We then need to make the properties available to our container, when we do that, we'll be creating an instance of our PropertySourcesPlaceholderConfigurer. And this is just a bean that Spring uses to return an instance of this PropertySourcesPlaceholderConfigurer. It's the equivalent of what we were doing in the XML, with that property-placeholder XML element. So it creates an instance of a bean, to store those values in. Now, once we have this done, we simply go through, and use that same @Value annotation to inject that into our code. We've already seen that part of it, so the really the big difference is, how we specify the location, and then, creating the bean to return that instance of it. Let's go ahead and implement that in our Java project now.

**Java Config Demo**

I've gone ahead and closed down all the files we had open from the XML demo that we did, and I'm going to just go open up our Spring sample Java project. The first thing I'm going to do is come down to our Source, Main Resources directory, right click, and create a new file for our app. properties. Now, inside of here, just like we did with the XML equivalent, we're going to create an instance of our dbUsername, and we want to go ahead and store that to some value. Just for fun, I'll maybe call this the derbyusername instead of the mysqlusername just so you see that it is a different demo and that we're not getting something cached for whatever reason from one project to the next. So now we have our app. properties, the next piece I need to implement is to open up our AppConfig and inside of here, make sure yours is looking like this, because we did a bunch of stuff with annotation config before. I'm going to go ahead and add the @PropertySource, and this will define that bean for us of the location of our properties file, now, you may have noticed as I was using that context-sensitive help that there is a singular property source, and a plural property sources, if you have multiple properties files, simply use the @PropertySources annotation instead of the singular one. So we want app. properties, inside of here, and now, for the Java PropertyPlaceholder bean configurator, I have created a snippet of this already so that you don't have to watch me type it clear out. It's a long, verbose class name, and you can use the context-sensitive help to help you create it, I'm just going to paste this in here, and you can stop the video and take a look at it. Going to import this bean in, and you can see this code that we have here. So we have a bean, public static, PropertySourcesPlaceholderConfigurer, and my getter doesn't need to be named this, I did use it this way, get PropertySourcesPlaceholderConfigurer, and we just simply return an instance of it. So what this class does, is takes all those properties that were loaded in from the property's source location, and puts them in this class and makes them available in the context for our application to use. So this is what was behind the scenes of that XML snippet. Now, I'm going to open up our HibernateCustomerRepositoryImpl, and do the same thing that we did before. Just, create a private string, dbUser and we want to use the @Value annotation, to inject that value into it, and it uses that ${ syntax. And, we'll go ahead and add that system. out. println at the bottom, so that see that we are in fact injecting a value into that. And, save this. And now we just need to run our application. So I'm going to go back to our application. Java, and right click, run as a Java application, and you'll see that that value is in fact, getting injected in there. We can see our derbyusername being output on the console. So really straightforward, we created our app. properties, have our values in here, we have our AppConfig that we've gone ahead and added that PropertySource annotation, and that really long classname for the PropertySourcesPlaceholderConfigurer, and then we have our HibernateCustomerRepositoryImpl, basically just the class that we want to inject something into, and all we had to do was add that @Value annotation in there, to inject that in. The reason I chose this example is because this is a case where I may want to inject in a username, possibly a password, or some other configuration information into a database tier to pull something specific back. If you've ever used some of the various database implementations out there like Oracle, for example, I can tell it which user I want to run a query under. Not necessarily what user I want to connect to the database as but, literally, what user I should implement a query under. And so, it's for a logging purposes, or configuration, there's a lot of reasons why you might want to do this, but that's one of the reasons we might want to inject that value in at runtime. Then we just ran our application to see, that it was exported out. So, once we have it configured, really simple to use, it is a long classname, but not that bad.

**Summary**

In this short module, we discussed how to import and use properties inside of Spring. Properties files are a key way of configuring Spring without having hard-coded values stored inside of your application. We configured our app both XML configuration, which I think is a little bit cleaner, and using Java configuration, to fully utilize the power of a properties file, and having those values extracted out.