As a specification, the Java Persistence API is concerned with persistence, which loosely means any mechanism by which Java objects outlive the application process that created them. Not all Java objects need to be persisted, but most applications persist key business objects. The JPA specification lets you define which objects should be persisted, and how those objects should be persisted in your Java applications.

By itself, JPA is not a tool or framework; rather, it defines a set of concepts that can be implemented by any tool or framework. While JPA's object-relational mapping (ORM) model was originally based on Hibernate, it has since evolved. Likewise, while JPA was originally intended for use with relational/SQL databases, some JPA implementations have been extended for use with NoSQL datastores. A popular framework that supports JPA with NoSQL is EclipseLink, the reference implementation for JPA 2.2. sss

A repository is a mechanism for encapsulating storage, retrieval, and search behavior which emulates a collection of objects. sss

Strategies re what are important for different caches

What are the basic three dependencies for Spring needed in the pom.xml? <!-- https://mvnrepository.com/artifact/org.springframework/spring-core --> <!-- Provides IOC Container --> <dependency> <groupId>org.springframework</groupId> <artifactId>spring-core</artifactId> <version>5.3.0</version>

</dependency> <!-- https://mvnrepository.com/artifact/org.springframework/spring-beans --> <!-- Provides the BeanFactory, dependency injection features --> <dependency> <groupId>org.springframework</groupId> <artifactId>spring-beans</artifactId> <version>5.3.0</version> </dependency> <!-- https://mvnrepository.com/artifact/org.springframework/spring-context --> <!-- provides package scanning and annotations -->

<dependency> <groupId>org.springframework</groupId> <artifactId>spring-context</artifactId> <version>5.3.0</version> </dependency>

Xmlns means xml namespace

What is Spring? What does it do? 1. Spring is a module-based, open source java framework

2. It provides support for enterprise-level applications

3. It provides infrastructure so you can focus on your application's business logic

What does module-based mean? Individual Modules of spring ... libraries packed on top of framework to let framework do more for us ... these modules made up foundations of framework ... will add more modules to give it more functionality

What is a module? A grouping of libraries that work together to achieve some goal using reusable abstracted code

What is a module in Spring? A spring dependency

What are some of the core modules of spring? Core,

beans,

context,

mvc,

orm,

aop,

security,

boot,

data,

session, etc.

Do Spring tools such as those for security overlap with those of AWS? Not really ...

Not a lot of overlap between AWS and what you would do custom ...

Spring security provides security but not for inside your app ...

eg. AWS has firewall but the concept of AWS is that you take care of your stuff and we'll take care of our stuff ...

When someone tries to break into server, there are security groups that help that ... but we can't help your application's security

Spring security, tools do different things than AWS services

How does Spring achieve its goals? 1. Inversion of control (IOC)

2. Model View Controller (MVC)

3. Abstraction API

4. Aspect-oriented Programming (AOP)

What are the scopes of a spring bean? 1. Singleton -- creates one instance for all references

2. Prototype -- creates an instance for each reference

3. Request -- creates an instance for each http request

4. Session -- creates an instance for each http session

5. GlobalSession -- creates one instance for all http sessions

Which version of Spring should you know? Spring 4

What are the scopes of Spring 5? Same as 4:

1. Singleton

2. Prototype

3. Request

4. Session

New to 5:

1. Application ties the instance to the lifecycle of the servlet context

2. Websocket ties the instance to the lifecycle of a websocket

What's a web socket and what are some facts to know about them? A websocket is a prolonged communication channel between computers

Http request one way and one way back

A websocket keeps it one

When you create a websocket it creates a diff protocol but you initiate it over http

Will get a message about switching protocols, and then you can say yes and it opens up a different website

There's a websocket api in servlets

A good example of a websocket is like in instance messaging/chatbot

What happened to Global Session in Spring 5? Application essentially does the same thing; it may be a change in the implementation but it has the same effect

What about Inversion of Control? IOC Container has many different implementations

BeanFactory ... makes beans

ApplicationContext

All bean factory can do, application context can do

Application context has more functionality because it inherits from bean factory

Application context is reference to ioc container

When we are referencing application context iths the bean inside the ioc container

IOC is a main context inside of spring ... simply means I'm giving control over something to something else ... giving up part of my application management to spring

Beans.xml is configuration file

Dependency injection is a subset of IOC

Framework for out entire java app

How we layer abstraction on top of our program

Spring is going to manage everything for us

The four different types of dependency injection and whether they are supported 1. Constructor injection (supported)

2. Setter injection (supported)

3. Field Injection (unofficially supported)

4. Interface injection (not supported at all)

Diagram of Spring design pattern Dispatcher servlet takes request ... handler mapping knows what routes go to what controllers ... dispatcher servlet is sending to handler mapping and it goes back to dispatcher servlet to tell it what to go next ... then sends to controller and from that controller is going to update the model and view and send it back to the dispatcher servlet

Totally restful api for project 2 ... diagram shows all you need

<!-- injecting primitives <bean class="com.springy.service.MockService" autowire="byType"> <constructor-arg index="1" type="int" value="5"/> </bean> --

<!-- this is a constructor injection --> <!-- <bean class="com.springy.service.MockService" scope="singleton"> <constructor-arg index="0" type="com.springy.repo.MockDao" ref="mockdao"/> </bean> -->

<!-- this is setter injection --> <!-- <bean class="com.springy.service.MockService"> <property name="md" ref="mockdao"/> </bean> -->

<!-- autowiring --> <bean class="com.springy.service.MockService" autowire="byType" />

Bean xml file set up <?xml version="1.0" encoding="UTF-8"?><beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.springframework.org/schema/beans https://www.springframework.org/schema/beans/spring-beans.xsd">

</beans>

Uses no arg constructor because of constructor injection

If using setter injection ... later on gets dependency

Web XML Spring MVC <?xml version="1.0" encoding="UTF-8"?><web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd" version="2.5"> <display-name>Spring-mvc</display-name> <welcome-file-list> <welcome-file>home.app</welcome-file> </welcome-file-list> <listener> <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class> </listener> <servlet> <servlet-name>dispatcher</servlet-name> <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class> </servlet> <servlet-mapping> <servlet-name>dispatcher</servlet-name> <url-pattern>\*.app</url-pattern> </servlet-mapping></web-app>

applicationContext.xml <?xml version="1.0" encoding="UTF-8"?><beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:mvc="http://www.springframework.org/schema/mvc" xmlns:cnx="http://www.springframework.org/schema/context" xsi:schemaLocation="http://www.springframework.org/schema/beans https://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc.xsd http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd "> <cnx:component-scan base-package="com.webby"/> <mvc:annotation-driven/> </beans>

Usual setter way or dep. Injection: think of in terms of eager and lazy

When to use each ... if use a class where only half of methods use setter method use setter injection ... because don't need as much

What is bean wiring?

Creating an association between Spring's container and an object you want spring to manage.

This is how we define what beans are a dependency on another and leave spring to manage them.

Benefits to Dependency Injection

Maintainability

Cleaner code

Scalability

Testability

Reduces complexity

Decouples our code

What is Application context?

A Beanfactory. It inherits from BeanFactory interface. It adds more functionality such as Internationalization, annotation support, enterprise services, etc.

<?xml version="1.0" encoding="UTF-8"?><beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.springframework.org/schema/beans https://www.springframework.org/schema/beans/spring-beans.xsd"> <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver"> </bean></beans>

Provides beanfactory, dpeenendcy injectionf features

Spring context

Provides package scanning and annotations

<!-- autowiring an automagical process of bean wiring 4 types

no does no autowire, sets it to off

byType spring will autowire by looking at the class' property datatypes, then matching them to the bean in the ioc container. This uses setter injection.

constructor works similar to byType, except it uses constructor injection and checks the type inside the constructor params and matches it to the bean in the ioc container.

byName Looks at the class's property variable names and finds beans in the container with the same name. it uses setter injection. -->

Only time see these autowiring types is when you configure

Stereotypes are annotations that tell spring that the class is going to be a spring bean and what responsibilities they are going to have.

\* There is a hierarchy to stereotypes \*

@Component the parent of them all \*

@Service denotes a service class \*

@Controller denotes a controller class \*

@Configuration denotes a configuration class \*

@Repository denotes a data access class \*

@RestController denotes a controller for a RESTful api.

RestController is a controller with a couple of other things mixed in ... makes it simpler to write rest apis but you don't get the granular control that you co with the controller

IOC container — implementation of how Spring is managing everytnhign for you

Md mockdao bean

Ms mock service bean

Md has instance of mock service

Changes between Bean Validation 2.0 and 1.1

Bean Validation 2.0 focused on the following topics:

support for validating container elements by annotating type arguments of parameterized types e.g. List<@Positive Integer> positiveNumbers. This also includes:

more flexible cascaded validation of container types

support for java.util.Optional

support for the property types declared by JavaFX

support for custom container types

support for the new date/time data types (JSR 310) for @Past and @Future

new built-in constraints: @Email, @NotEmpty, @NotBlank, @Positive, @PositiveOrZero, @Negative, @NegativeOrZero, @PastOrPresent and @FutureOrPresent

leverage the JDK 8 new features (built-in constraints are marked repeatable, parameter names are retrieved via reflection)

You can consult an HTML diff between the 1.1 specification and the 2.0 specification here.

For more information, check out the full list of changes.

JSR validation is part of Java javax.validation.constraints.Email, etc.

Dependency injection:

Instantiation objects but not just that

It's managing everything for us ... decouples everything because we're no longer piecing it together

Now it can be dynamically changed to another ... now it's coupled to the dependency of another

Now we can use spring to actually inject all the values we need in there

If specify a regex pattern and it doesn't match, you can do some kind of exception handling because apparently it doesn't work

What happens when spring makes a bean? The lifecycle of a spring bean

Instantiation

Populate Properties

Set Name

Set Factory

Set Application Context

Before Post Processing

After Populate Properties

Custom Init

After Post Processing

Use

Destroy

Custom Destroy Instantiation

Populate properties

Set name

Set factory

Set application context

Before post processing (just accept it ... use of "post" is weird"

After populate properties

Custom Init

After Post processing

Use

Destroy

Custom destroy

What happens when a spring makes a bean

Everything has a lifecycle

The lifecycle of a spring bean

Instantiation ... instantiate being populating properties, set name (unless do it before), set application context, process bean and decide where to inject it ... after populate properties can do some kind of action ... then after post processing ... after its normal flow of operations Created, everything set on it, processing, then use it, then destroy it with whatever custom destroy method you have itg

Don't need custom method

Can tap into the lifecycle by using different interfaces and methods from those interfaces

There may even be annotations that can use for lifecycle methods ... or can just use the methods

Need to remember this lifecycle and explain it

Read spring docs for this

Baeldung is no. 1 spring resource

You register a servlet; register is the term

Whenever we make a war, we need to use a servlet

Springwebmvc already has that ... has a servlet, has it set up ... all you need to do is include it ... register their servlet with your application then register their servlet with your application and view in dispatcher-servlet

Bean id=viewresolver tells which view to use

<cnx: etc.

<mvc: etc.

These are the only two tags you need

Spring MVC

No more requestdispatcher ... use get mapping

Automatically makes it JSON

Tie lifecycle of Tomcat to Spring ... add <listener><listener-class>org.springframework.web.context.ContextLoaderListener (then end tags)

For fuller architecture, needs viewresolver ... from dispatcher servlet to viewresolver to decide what view to send and then send it back to dispatcher servlet, grabs the view, and then sends response to client

Viewresolver takes a template (like a templating engine) and generates whatever needs to be done for html and the html generator would know what to do

AOP is a way to decouple spaghetti code

AOP Aspect-oriented Programming

AOP is meant to complement OOP

Spring AOP is an implementation of aspectJ

AspectJ is the specification for Srping AOP

Aspect In Java, a class is the embodiment of an aspect

Conceptually, you can think of it as a collection of advice.

Advice A procedure for interacting with the program during runtime. These take the form of methods in Spring AOP

"Advice" may be associated with methods, functions in different languages

Example of Aspect being used in diagram notice the need for @Aspect annotation

diagram of aspect imports

Use of AOP is to reduce what is known as crosscutting concerns

Two kinds of crosscutting concerns tangling -- spaghetti code ... spring helps you decouple it even further (through AOP)

scattering

Crosscutting concerns are concerns that cut across your application ex. if you have a problem in class a, you may have a problem in class b

ex. logging code smells, which is scattering, duplication of lines and other code smells

Spring decouples and it uses AOP to help it do that Will be using Spring all over AOP

jp.getsignature()

Joinpoint is a point in runtime that we can execute our advice

our advice takes in the joiunpoint as a parameter, it includes information about the runtime of the program is a point in runtime that we can execute our advice

our advice takes in the joiunpoint as a parameter, it includes information about the runtime of the program

Inforamtiona bout the runtime; about what is happening; like a hook kind of

Pointcut used to select certain joinpoints

used to target certain subsets of the joinpoints of the program

Pointcut expressions 1. execution - determines that we are looking for joinpoints associate with method executions

2. \* - wildcard that stands for anything

3. .. - Used for parameters for any number, any data type

No matter how many arguments you have, it's going to hit it

Joinpoints, etc. are used for logging ... and reduces duplicated logging points

Advice timing 1. @Before

2. @After

3. @AfterReturning

4. @AfterThrowing (There was an exception, so now I'm running)

5. @Around (can determine if it's going to run)(uses ProceedingJoinPoint type, not JoinPoint)

When would a method not return something? When it throws an exception

Java gen knowledge: void has an implicit return value

For example, the Single Responsibility Principle (SRP) suggests that a class should have only one reason to change.

DIP suggests that high-level modules should not depend on low level modules. Both should depend on abstraction.

DIP Definition

High-level modules should not depend on low-level modules. Both should depend on the abstraction.

Abstractions should not depend on details. Details should depend on abstractions.

The second rule in DIP is "Abstractions should not depend on details. Details should depend on abstractions".

Dependency Injection (DI) is a design pattern used to implement IoC. It allows the creation of dependent objects outside of a class and provides those objects to a class through different ways. Using DI, we move the creation and binding of the dependent objects outside of the class that depends on them.

The Dependency Injection pattern involves 3 types of classes.

Client Class: The client class (dependent class) is a class which depends on the service class

Service Class: The service class (dependency) is a class that provides service to the client class.

Injector Class: The injector class injects the service class object into the client class.

The following figure illustrates the relationship between these classes:

Dependency Injection

As you can see, the injector class creates an object of the service class, and injects that object to a client object. In this way, the DI pattern separates the responsibility of creating an object of the service class out of the client class.

Types of Dependency Injection

As you have seen above, the injector class injects the service (dependency) to the client (dependent). The injector class injects dependencies broadly in three ways: through a constructor, through a property, or through a method.

Constructor Injection: In the constructor injection, the injector supplies the service (dependency) through the client class constructor.

Property Injection: In the property injection (aka the Setter Injection), the injector supplies the dependency through a public property of the client class.

Method Injection: In this type of injection, the client class implements an interface which declares the method(s) to supply the dependency and the injector uses this interface to supply the dependency to the client class.

Service Layer

A commonexamplewould be to have the implementation you will use in your productioncode, and then a mock implementation for unit testing yourcode. This is the power ofdependency injection. It allows you to change the behavior of yourapplicationthrough configuration changes overcodechanges.

What is JSR? JSRs are Java Specification Requests, basically change requests for the Java language, libraries and other components. It's all part of the Java Community Process, whereby interested parties can put forward their ideas for enhancements and (hopefully) have them taken up and acted upon.

What is a Spring container otherwise known as? ApplicationContext

What is a Spring Bean? A "Spring Bean" is simply a Java object.

When Java objects are created by the Spring Container, then Spring refers to them as "Spring Beans".

Spring Beans are created from normal Java classes .... just like Java objects.

When we pass the interface to the method, behind the scenes Spring will cast the object for you.

context.getBean("myCoach", Coach.class)

However, there are some slight differences than normal casting.

From the Spring docs:

Behaves the same as getBean(String), but provides a measure of type safety by throwing a BeanNotOfRequiredTypeException if the bean is not of the required type. This means that ClassCastException can't be thrown on casting the result correctly, as can happen with getBean(String).

So what exactly are scopes?

Well, a scope refers to the lifecycle of a bean,

for example, it tells you how long the bean will live,

how many instances will be created

and also how is the bean shared in the Spring environment?

So the default scope for a bean is singleton,

so here's an example here of just some bean code,

that we had earlier and we didn't explicitly give a scope,

so by default, the scope is singleton.

But now you're probably wondering,

well, what exactly is a singleton? (laughs)

we know it's a default scope, but what is it?

Well, for a singleton, the Spring Container

creates only one instance of the bean, it's cached in memory

and then all requests for that bean

will return a shared reference to the same bean,

so the end result is that there is only one bean

and everyone will share it.

Here's a nice diagram to kind of show you this example.

In this example, we have theCoach

equals context.getBean myCoach

and it'll give you a reference to like a TrackCoach,

that you have defined and then later on in the code,

if you would also do a similar thing,

saying context.getBean myCoach, the same bean id,

then it'll basically give you a reference to the same bean.

We have these two object references here

and they point to the same area of memory

or they point to the same bean,

so again, Spring makes use of a singleton,

it'll create only one bean and then share it

for everyone who requests that bean,

so the singleton scope is default

and the best use case for this is for a stateless bean,

where you don't need to maintain any state.

You can explicitly specify the bean scope,

I mean, by now you know that by default,

you have the singleton scope,

but if you want to explicitly specify,

then you make use of the scope attribute,

so you say scope equals singleton

and that'll make it a singleton bean

and that's kind of the preferred approach

to minimize the number of beans, that are created.

But now, there are additional Spring bean scopes,

that you can make use of.

We've already covered singleton,

there's also the prototype scope,

which creates a new bean instance for each container request

and we'll see examples of that coming up

and we'll also use that as a demo in the video coming up,

then the next three items here,

request, session and global-session,

these scopes are only used in a web environment,

so request is for a given web request,

session is for a HTTP web session, for like session tracking

for like maybe a shopping cart or something

and then global-session is scope application-wide,

but we'll talk more about these later,

when we get into the Spring MVC section,

but for now, we'll simply focus on singleton and prototype.

So here's an example of using prototype scope,

so again, remember, prototype scope,

a new object is created for each request,

so in this example here, I have my bean id of myCoach

and I have scope equals prototype,

so that means that every time

I make a request for this myCoach,

they'll create a new instance each time.

So, a nice little diagram here.

So the line of code at the top,

theCoach equals context.getBean myCoach,

it'll create a new instance of that bean,

I'll get a reference to it and then a similar thing here,

when I say alphaCoach equals context.getBean myCoach,

it'll create a new object for you

and you'll have your own reference.

So the prototype scope is good for keeping track

of stateful data, so again, whenever you see prototype,

just think of the new keyword, it's gonna create a new bean

for each request for that component or that object.

Alright, so this is some really good stuff.

Let's go ahead and move forward,

in the next videos, we're gonna dive into Eclipse

and we'll actually write some code,

that'll make use of the singleton scope

and the prototype scope,

so you can see everything in action.

Alrighty, I'll see you in the next video.

bean lifecycle methods.

So when the Spring container first starts

there are a couple of things that happens.

First off, the beans are instantiated,

and then the actual dependencies are injected,

next you have some internal Spring processing

that occurs with the bean factory,

and then you have the option

of adding your own custom initialization code,

and then at that point the bean is ready for use.

So you can call methods on it,

do work with the bean,

so on and so forth.

At a certain point the containers actually shutdown

meaning your application is shutdown

like what context.close,

then you also have a chance

to call your custom destroy method,

and that code'll execute before the actual application

is stopped or before the actual beans lifecycle is over.

The one thing you want to take a look

at are those two orange sections

here adding your own custom methods.

What you can do is you can add your own custom code

that happens during bean initialization.

So you can call custom business logic,

you can set up handles,

so like databases, or sockets, or whatever.

You can also do a similar thing

when a bean is actually being destroyed or destructed.

So again, you can call any custom business methods,

or you can clean up any handles

that you may have to resources

like databases, sockets, or files.

So basically what this provides here

is that during the bean lifecycle,

Spring allows you to call some of your custom code,

and these are what we call hooks,

where you can actually hook in codes

to execute during bean initialization or bean destruction.

All right, so now, how would you do this?

Well, you simply make configuration entry in your XML file.

So for bean initialization,

you make use of this attribute called init-method,

and then you give the actual method name

that you would like for Spring to call on your bean.

Now this method name can be any method name.

Here I called doMyStartupStuff.

It could be called fubar or whatever,

and we can also do a similar thing for the destroy method.

So again, you simply make

a configuration entry here in your XML file.

You simply give the method name

you want Spring to call for destroy,

and again, it can be any name.

All right, so this all looks kinda good.

What's the basic development process?

So again, I love my step-by-step.

So the first thing you do is you simply

define your methods for the init and destroy

in your bean class, then the next step

is you simply configure those method names

in the Spring configuration file,

and in the next video, we'll dive into Eclipse,

and we'll actually walk through this development process,

and we'll test out this feature.

So I'll see you in the next video.

In this video,

we're gonna discuss bean lifecycle methods.

So when the Spring container first starts

there are a couple of things that happens.

First off, the beans are instantiated,

and then the actual dependencies are injected,

next you have some internal Spring processing

that occurs with the bean factory,

and then you have the option

of adding your own custom initialization code,

and then at that point the bean is ready for use.

So you can call methods on it,

do work with the bean,

so on and so forth.

At a certain point the containers actually shutdown

meaning your application is shutdown

like what context.close,

then you also have a chance

to call your custom destroy method,

and that code'll execute before the actual application

is stopped or before the actual beans lifecycle is over.

The one thing you want to take a look

at are those two orange sections

here adding your own custom methods.

What you can do is you can add your own custom code

that happens during bean initialization.

So you can call custom business logic,

you can set up handles,

so like databases, or sockets, or whatever.

You can also do a similar thing

when a bean is actually being destroyed or destructed.

So again, you can call any custom business methods,

or you can clean up any handles

that you may have to resources

like databases, sockets, or files.

So basically what this provides here

is that during the bean lifecycle,

Spring allows you to call some of your custom code,

and these are what we call hooks,

where you can actually hook in codes

to execute during bean initialization or bean destruction.

All right, so now, how would you do this?

Well, you simply make configuration entry in your XML file.

So for bean initialization,

you make use of this attribute called init-method,

and then you give the actual method name

that you would like for Spring to call on your bean.

Now this method name can be any method name.

Here I called doMyStartupStuff.

It could be called fubar or whatever,

and we can also do a similar thing for the destroy method.

So again, you simply make

a configuration entry here in your XML file.

You simply give the method name

you want Spring to call for destroy,

and again, it can be any name.

All right, so this all looks kinda good.

What's the basic development process?

So again, I love my step-by-step.

So the first thing you do is you simply

define your methods for the init and destroy

in your bean class, then the next step

is you simply configure those method names

in the Spring configuration file,

Special Note about init and destroy Method Signatures

When using XML configuration, I want to provide additional details regarding the method signatures of the init-method and destroy-method .

Access modifierThe method can have any access modifier (public, protected, private)

Return typeThe method can have any return type. However, "void' is most commonly used. If you give a return type just note that you will not be able to capture the return value. As a result, "void" is commonly used.

Method nameThe method can have any method name.

ArgumentsThe method can not accept any arguments. The method should be no-arg.

With what scope can you not use the destroy lifecycle method destroy-method? There is a subtle point you need to be aware of with "prototype" scoped beans.

For "prototype" scoped beans, Spring does not call the destroy method. Gasp!

---

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

Thus, although initialization lifecycle callback methods are called on all objects regardless of scope, in the case of prototypes, configured destruction lifecycle callbacks are not called. The client code must clean up prototype-scoped objects and release expensive resources that the prototype bean(s) are holding.

---

This also applies to both XML configuration and Annotation-based configuration.

---

QUESTION: How can I create code to call the destroy method on prototype scope beans

ANSWER:

You can destroy prototype beans but custom coding is required.

You can download the example source code from here: destroy-prototype-scope-bean-with-custom-processor-xml-config.zip

Development Process

1. Create a custom bean processor. This bean processor will keep track of prototype scoped beans. During shutdown it will call the destroy() method on the prototype scoped beans. The custom processor is configured in the spring config file.

<!-- Bean custom processor to handle calling destroy methods on prototype scoped beans -->

<bean id="customProcessor"

class="com.luv2code.springdemo.MyCustomBeanProcessor">

</bean>

2. The prototype scoped beans MUST implement the DisposableBean interface. This interface defines a "destory()" method.

public class TrackCoach implements Coach, DisposableBean {

...

// add a destroy method

@Override

public void destroy() throws Exception {

System.out.println("TrackCoach: inside method doMyCleanupStuffYoYo");

}

}

3. The Spring configuration does not require use the destroy-method attribute. You can safely remove it.

<bean id="myCoach"

class="com.luv2code.springdemo.TrackCoach"

init-method="doMyStartupStuff"

scope="prototype">

<!-- set up constructor injection -->

<constructor-arg ref="myFortuneService" />

</bean>

4. In this app, BeanLifeCycleDemoApp.java is the main program. TrackCoach.java is the prototype scoped bean. TrackCoach implements the DisposableBean interface and provides the destroy() method. The custom bean processing is handled in the MyCustomBeanProcessor class.

So what exactly are Java annotations?

Well all they are, they're simply special labels

or markers that are added to Java classes

and they actually give you meta-data about the class.

So like I have here in this photo of a shoebox,

we have meta-data or a label on this shoebox.

So we have the actual size of shoe, the style of shoe,

the model number, and so on.

So that's meta-data about this shoe.

So, again, Java annotations are simply

meta-data about a class.

So we can actually have annotations that are processed

at compile time or at run-time,

What is Spring Profiles for? Multiple implementations

Remember: Functionally, Service and Component are the same functionally ... you use the annotations differently to show what they're doing intentionally, what they're behavior is

Are Service and Component annotations the same functionally? Yes, but you use the annotations differently to show what they're doing intentionally ... what they're behavior is

What responsibility does an injected class have in instantiating the object being injected? None. The framework is creating all the objects for you — a tenet of IoC — the class being injected has no responsibility in instantiating the object being injected

Example of public static void main(String[] args) {

ApplicationContext ctx = SpringApplication.run(DiDemoApplication.class, args);

MyController controller = (MyController) ctx.getBean("myController");

This is where Spring is making an instance available instead of using the new keyword

System.out.println(controller.hello());

What annotation will be the output for all of the run services if your services don't have qualifiers? The one you set a primary annotation on

https://www.udemy.com/course/spring-framework-5-beginner-to-guru/learn/lecture/17852650#questions

What is the best way to work with legacy JDBC in a Spring environment? While JPA is clearly the preferred way of interacting with a database, there are times where you need more fine grained access. Maybe you're just extracting data from a legacy database for some type of data conversion or reporting need. It may not be worth going through the work of setting up JPA.

This is where Spring's JDBC Template steps in. If you're working within a Spring environment, JDBC Template is the preferred method of access the database directly. I highly recommend you use JDBC Template over rolling your own JDBC connection. Using JDBC Template will help you from preventing common errors.

https://docs.spring.io/spring-framework/docs/current/reference/html/data-access.html#jdbc

What command do you use in Spring Boot to get help? mvn spring-boot:help

What is Spring Security? Spring Security is designed for securing access to Java based applications. The most common use case is with web based Java applications, but it actually can be used with any Java application, whether it is a GUI client or some type of exposed API.

Spring Security looks at two areas when allowing access to an application. One is authentication, the other is authorization. The first is authentication, which asks the question who am I? This establishes identity. This could be a user id password from a database or LDAP server. Or in other cases from a 3rd party via OpenID. The second area is authorization. Is the user allowed to take the requested action. This asks the question are they allowed to do this?

Spring Security is a mature product and has been around sometime. It is common to see used for building enterprise applications in Spring. Spring Security can easily be used with LDAP servers or Microsoft Active Directory. Having this integration with centralized identity servers is important to companies because of compliance issues. When an associate leaves the company, the company need to have established procedures to terminate the associate's access to its systems. And being able to integrate with LDAP / Active Directory is the perfect tool for corporate compliance.

What is the Spring @Component annotation do? It tells the program to detect it as spring component (bean); it allows for dependency injection

What does Spring commandlinerunner do? It tells spring to look for instances of the type and run them

What is Thymeleaf? A template engine

When was Thymeleaf created? July 2011

What is a natural template engine? A template engine that lets you view templates in the browser

Is Thymeleaf a natural template engine? Yes

Does Spring have curated dependencies? Yes

How does making a component annotation relate to Spring Beans and SpringContext. When you make something a @Component, the component becomes a Spring Bean and you register it with the SpringContext (lifecycle)

What are four kinds of data layer implementations in Java? Map

Spring Data JPA

Old fashioned JDBC

Working with raw Hibernate

What is the args for in this code?

@Override

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

} optional environment parameters

What are the configuration options for the Spring Framework? 1. XML

2. Annotations

3. Java

4. Groovy

5. DSL (Domain-specific Language)

Is XML Configuration still

supported in Spring Framework 5? Yes, XML configuration is still fully

supported in Spring Framework 5

Which type of Dependency Injection should you favor and why? By Constructor - This requires the

dependency to be injected when the

class is instantiated.

What type of annotation does a component scan look for? Component scans look for beans annotated with Spring's Stereotype annotations.

@Controller

@Service

@Component

@Repository

What annotation do you use to declare a Java class is for Spring configuration? @Configuration

What annotation do you use to declare a Spring component inside a Java configuration class? @Bean will declare a Spring Bean (component)

If you have two beans of the same type, how do you specify a preference for one over the other? The @Primary annotation can be used to designate a primary bean

What two annotations can be used to access the Spring Bean lifecycle? 1. @PostContstruct

2. @PreDestroy

Is @RestController a Spring Stereotype? Technically, its a convenience annotation which combines @Controller and @ResponseBody

Why should you allow your Spring Boot Maven projects to inherit from the Spring Boot parent POM? This allows your project to inherit curated dependencies which are known to be compatible.

Which annotation do Spring Stereotypes inherit from? All Spring Stereotypes inherit from @Component

Can you build a Spring Boot project using Ant? Yes. Ideally you should configure Ivy for dependency resolution.

What is a Spring Boot Starter? A Spring Boot Starter is a POM which declares

a common set of dependencies. Spring Boot

Starters are available for most Java projects.

What is special about the @Repository stereotype? Spring will detect platform-specific persistence exceptions and re-throw them as Spring exceptions

The Spring Boot @SpringBootApplication annotation includes 3 other annotations. What are they? 1. @Configuration

2. @EnableAutoConfiguration

3. @ComponentScan

How can you display an auto-configuration report from Spring Boot? Start Spring Boot with the command line parameter -- debug

Can you disable specific Spring Boot auto-configuration classes? Yes, pass the class name to the exclude parameter of the @EnableAutoConfiguraiton annotation

What is the default scope for beans in Spring? Singleton Scope - Only one instance is created

Can you create custom bean scopes in Spring? Yes, Spring is extensible and you can define your own scopes

How are beans created for the Prototype scope? Spring will create a new instance of the bean for every object requesting it

SpringBoot is a configuration wrapper for the whole spring framework. It encompasses SpringMVC but it is NOT a web application such as SpringMVC

What is component scan in Spring? Using component scan is one method of asking Spring to detect Spring-managed components. Spring needs the information to locate and register all the Spring components with the application context when the application starts.

The curated list contains all the spring modules that you can use with Spring Boot as well as a refined list of third party libraries. The list is available as a standard Bills of Materials ( spring-boot-dependencies ) and additional dedicated support for Maven and Gradle are available as well.

What are factory methods used fo? More complex objects

Do Spring programmers usually load bootstrap data on startup? Yes

What are the advantages of Autowiring in Spring? 1. By using Autowiring there is a significant reduction in the explicit configuration needed to specify properties or constructor arguments.

2. By using autowiring, maintaining code and evolution of code is easy as configuration is updated automatically. For example, if you need to add any new dependency to a class, that dependency can be satisfied automatically, you do not need to modify the configuration explicitly.

What are the limits and disadvantages of Autowiring in Spring? 1. If you are using autowiring it should be used consistently across a project. If autowiring is used to wire only one or two bean definitions, it might be confusing to developers and make code less readable where developer has to refer multiple sources to see the relationships among beans.

2. Autowiring can't be used to autowire simple properties such as primitives, Strings, and Classes (and arrays of such simple properties). This limitation is by design.

3. One of the limitations of autowiring is that explicit dependencies in property and constructor-arg settings always override autowiring.

4. Autowiring is less exact than explicit wiring. In the case where Spring container can't determine the exact bean to be autowired, it won't try to guess and an exception is thrown instead.

5. For dependencies that expect a single value, if there are multiple bean definitions matching that dependency type within the container then this ambiguity is not arbitrarily resolved in case of autowiring. If no unique bean definition is available, an exception is thrown.

6. Wiring information may not be available to tools that may generate documentation from a Spring container.

Java-based bean configuration is the trend, not so much xml

Xml probably not going away any time soon because of all the legacy code though java-based configuration is the current standard

You want to be very specific about what beans to involve because projects can be very big

Will use prepare and perform for release plugin goals mostly

Other goals are for when there are problems or need to do something special on it

<build>

<plugins>

<plugin>

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

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

<executions>

<execution>

<goals>

<goal>repackage</goal>

</goals>

</execution>

</executions>

<configuration>

<skip>true</skip> ... THIS IS FOR MAKING SURE NOT ALL THE DEPENDENCIES ARE INCLUDED IN THE PACKAGE FOR JUST THE DATA MODULE

</configuration>

</plugin>

</plugins>

</build>

Actuator endpoints let you monitor and interact with your application. Spring Boot includes a number of built-in endpoints and lets you add your own. For example, the health endpoint provides basic application health information.

Constructors is better than setters or POJOs for dependency injection

DI with interfaces is preferred over concrete classes cuz

It allows the runtime to decide which implementation to inject

Follows IS principle of SOLID

Makes code more testable

Example: easy to swap out different databases because using very specific interfaces that follow IS principle and don't have dependencies (ME) that are not needed, not there, etc.

depeneency injection of properites is not preferred, but if you do, don't make the properties private because there will be problems in testing down the road

IoC ... method to inject dependencies at runtime ... dependencies are not predetermined

ME: DI is what "physically" happens between classes and IoC is conceptual to how your runtime eivornment/container ... so nothing inside the class can change it in theory anyway

IoC refers to the runtime environment (or library) that injects depenencies

spring' s container is often called an IoC container because a container in the context is what is going to be determining what gets injected as a dependency into your classes

With IoC the classes in a runtime environment are not in control ... not asking for anything ... control is being done by the framework

Open Closed Principle

As applications evolve, changes are required. Changes are required when a new functionality is added or an existing functionality is updated in the application. Often in both situations, you need to modify the existing code, and that carries the risk of breaking the application's functionality. For good application design and the code writing part, you should avoid change in the existing code when requirements change. Instead, you should extend the existing functionality by adding new code to meet the new requirements. You can achieve this by following the Open Closed Principle.

The Open Closed Principle represents the "O" of the five SOLID software engineering principles to write well-designed code that are more readable, maintainable, and easier to upgrade and modify. Bertrand Meyer coined the term Open Closed Principle, which first appeared in his book Object-Oriented Software Construction, release in 1988. This was about eight years before the initial release of Java.

What annotation do you use to specify a bean name that is to be injected? @Qualifier

What two interfaces can you use to tap into the bean lifecycle? InitializingBean and DisposableBean

@Autowired is required for all kinds of injectors except for constructors

... as of Spring 4.2 The actual words in the annotations @Component, @Service, @Controller don't mean anything; any of the annotations will work because they make it a spring-managed component; the specific word is to show intention of how to be utilized

Is it common to put JPA entities in their own module? Yes

Not just code that's executing, spring is using spring context to manage components (spring beans) , Those spring bean component objects are going to be constructed and become included in the context

This means we can go in and ask the spring context for a reference to that object

So if we have a controller that has been created and is in the context, we can ask the context for a reference to that controller object that is being managed by the spring framework

There is instantiation of objects but with spring it's all behind the scenes. The spring framework is managing the construction of new objects

There is not as much need for new keyword to instantiate objects with dependency injection but there are use cases for doing it

Best practices

use final properties for injected components (ex injecting a data source)

When practical, code to an interface ... might be practical to code acc. To a spec. Type but usually use interface

5. The IoC container

5.1 Introduction to the Spring IoC container and beans

This chapter covers the Spring Framework implementation of the Inversion of Control (IoC) [1]principle. IoC is also known as dependency injection (DI). It is a process whereby objects define their dependencies, that is, the other objects they work with, only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method. The container then injects those dependencies when it creates the bean. This process is fundamentally the inverse, hence the name Inversion of Control (IoC), of the bean itself controlling the instantiation or location of its dependencies by using direct construction of classes, or a mechanism such as the Service Locator pattern.

The org.springframework.beans and org.springframework.context packages are the basis for Spring Framework's IoC container. The BeanFactory interface provides an advanced configuration mechanism capable of managing any type of object. ApplicationContext is a sub-interface of BeanFactory. It adds easier integration with Spring's AOP features; message resource handling (for use in internationalization), event publication; and application-layer specific contexts such as the WebApplicationContext for use in web applications.

In short, the BeanFactory provides the configuration framework and basic functionality, and the ApplicationContext adds more enterprise-specific functionality. The ApplicationContext is a complete superset of the BeanFactory, and is used exclusively in this chapter in descriptions of Spring's IoC container. For more information on using the BeanFactory instead of the ApplicationContext, refer to Section 5.15, "The BeanFactory".

In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. Otherwise, a bean is simply one of many objects in your application. Beans, and the dependencies among them, are reflected in the configuration metadata used by a container.

5.2 Container overview

The interface org.springframework.context.ApplicationContext represents the Spring IoC container and is responsible for instantiating, configuring, and assembling the aforementioned beans. The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata. The configuration metadata is represented in XML, Java annotations, or Java code. It allows you to express the objects that compose your application and the rich interdependencies between such objects.

Several implementations of the ApplicationContext interface are supplied out-of-the-box with Spring. In standalone applications it is common to create an instance of ClassPathXmlApplicationContext or FileSystemXmlApplicationContext. While XML has been the traditional format for defining configuration metadata you can instruct the container to use Java annotations or code as the metadata format by providing a small amount of XML configuration to declaratively enable support for these additional metadata formats.

In most application scenarios, explicit user code is not required to instantiate one or more instances of a Spring IoC container. For example, in a web application scenario, a simple eight (or so) lines of boilerplate J2EE web descriptor XML in the web.xml file of the application will typically suffice (see Section 5.14.4, "Convenient ApplicationContext instantiation for web applications"). If you are using the SpringSource Tool Suite Eclipse-powered development environment or Spring Roo this boilerplate configuration can be easily created with few mouse clicks or keystrokes.

The following diagram is a high-level view of how Spring works. Your application classes are combined with configuration metadata so that after the ApplicationContext is created and initialized, you have a fully configured and executable system or application.

SpringBoot is a configuration wrapper for the whole spring framework. It encompasses SpringMVC but it is NOT a web application such as SpringMVC

Example of XML-based configuration src/main/java/com/stazman/generator\_app/GeneratorAppApplication.java

@@ -2,8 +2,10 @@

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication

;import org.springframework.context.annotation.ImportResource;

@SpringBootApplication

@ImportResource("classpath:chuck-config.xml")

public class GeneratorAppApplication {

public static void main(String[] args) {

....

src/main/java/com/stazman/generator\_app/config/ChuckConfiguration.java

@@ -1,15 +1,15 @@

package com.stazman.generator\_app.config;

import guru.springframework.norris.chuck.ChuckNorrisQuotes;

public ChuckNorrisQuotes chuckNorrisQuotes(){ return new ChuckNorrisQuotes(); }}}

7 src/main/resources/chuck-config.xml

@@ -0,0 +1,7 @@

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

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

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean name="chuckNorrisQuotes"

class="guru.springframework.norris.chuck.ChuckNorrisQuotes"/>

</beans>

Example of Java-based configuration package com.stazman.generator\_app.config;

import guru.springframework.norris.chuck.ChuckNorrisQuotes;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

public class ChuckConfiguration {

@Bean

public ChuckNorrisQuotes

chuckNorrisQuotes(){ return new

ChuckNorrisQuotes();

}

}

Steps in coding with Spring beans 1. Enable component scanning in Spring config

instead of listing out all the beans, we can remove

all of that stuff and simply have one entry here.

We'll simply say, context component scan,

and you give it the base package that you want it to scan.

And so Spring will actually go through and scan all classes

in this package and all subpackages and it'll identify

the components that have that annotation on it and it'll

automatically register them with the Spring container.

That happens in the background for you automatically.

2. Add @Component annotations in Java classes

and we simply give the actual bean ID that we wanted to use as a parameter (eg., @Component("id-goes-here"))

3. Retrieve the bean from the Spring container

What is ApplicationContext, most essentially? ApplicationContext is an interface for providing configuration information to an application. There are multiple classes provided by springframework that implements this interface and helps us use configuration information in applications. ApplicationContext provides standard bean factory lifecycle capabilities.

What is a SpringContext? Spring contexts are also called Spring IoC containers, which are responsible for instantiating, configuring, and assembling beans by reading configuration metadata from XML, Java annotations, and/or Java code in the configuration files.

What does using Spring-context scanning and using annotations let you do? Not have to list individual beans in the configuration file (eg., applicationContext.xml

How do you make a bean default? @Component doesn't need to have the bean ID as a parameter; not including the id as a parameter makes it default

What is Spring AutoWiring? For dependency injection,

Spring can automatically wire up your objects together,

so basically what'll happen is that Spring

will look for a class that matches a given property.

And it'll actually match by type,

so the type could be either the class or the interface.

Once Spring finds a match,

then it'll automatically inject it.

Hence it's called autowired.

What are the three Autowiring injection types? 1. Constructor injection

2. Setter injection

3. Field injection

What is the development process for constructor injection? 1. Define the dependency interface and class

2. Create a constructor in our class for injections

3. Configure the dependency injection using

@Autowired

What does @Autowired do specifically, with an example? the parameter being passed in FortuneService,

we want Spring to automatically wire up this component,

so instead of doing it long-hand, using XML configs,

we're making use of this Autowired annotation.

So again, what happens in the background

is that Spring will say hey,

I need to satisfy this dependency.

So Spring will again, it'll scan all the components there,

it'll find the component that implements

this FortuneService interface,

which in our case is HappyFortuneService.

And it'll take that bean, actually inject it,

or Autowire it here into this TennisCoach.

So that's how they handle the dependency injection

using that Autowired annotation.

Do you need to list out all the beans in the applicationContext.xml file if you use annotations? If not, what is the one thing you must have? No. The one thing that you must have is like this:

<context:component-scan base-package="com.writesourcer" />

What is Auto Component Scanning? Using component scan is one method of asking Spring to detect Spring-managed components (or beans). Spring needs the information to locate and register all the Spring components with the application context when the application starts. Spring can auto scan, detect, and instantiate components from pre-defined project packages.

What does this code mean?

<context:component-scan base-package="com.fully-qualified-name" /> It's putting the context:component in the bean configuration file. It means enabling the auto scanning feature in Spring. The base-package indicates where your components are stored, Spring will scan this folder and find beans (annotated with the @Component annotation) and register it in the Spring container.

Why/when is it possible not to use the @Autowired annotation for constructor injection? This is a new feature of Spring 4.3.

Here is the snippet from the Spring Docs.

Section 1.9.2: Autowired

As of Spring Framework 4.3, an @Autowired annotation on such a constructor is no longer necessary if the target bean only defines one constructor to begin with. However, if several constructors are available, at least one must be annotated to teach the container which one to use.

I personally prefer to use the @Autowired annotation because it makes the code more readable. But as mentioned, the @Autowired is not required for this scenario.

What is the developer process for using setter injection? 1. Create setter methods in our class for injections

2. Configure the dependency injection with the Autowired annotation

Does Spring support interface injection? No

Does Spring restrict your use of either Setter or Constructor Injection in a single configuration file? No

When should you use setter injection instead of constructor injection 1. When the number of dependencies is great and they don't need to all be run at once

2. When you need readability

3. if there is a circular dependency between two object A and B.If Object A and B are dependent each other i.e A is depends ob B and vice-versa. Spring throws ObjectCurrentlyInCreationException while creating objects of A and B bcz A object cannot be created until B is created and vice-versa. So spring can resolve circular dependencies through setter-injection. Objects constructed before setter methods invoked.

Read more:https://javarevisited.blogspot.com/2012/11/difference-between-setter-injection-vs-constructor-injection-spring-framework.html#ixzz6sEOnij3z

When should you use constructor injection instead of setter injection? When an object must be created with all of its dependencies at once

A drawback of setter Injection is security. By using setter injection, you can accidentally override certain dependency which is not possible with constructor injection because every time you call the constructor, a new object is gets created

one of the drawbacks of setter injection is that it does not ensures dependency Injection. You can not guarantee that certain dependency is injected or not, which means you may have an object with incomplete dependency.

Can you use any method with a class by using the @Autowired annotation? YES!

Does using Spring make it less and less necessary to name codes exactly according to related codes? YES!

How is field injection accomplished? Through Java Reflection

Can you use field injection with private variables? Yes

What is the development process for field injection? 1. Configure the dependency injection

using the Autowired annotation

2. Place this annotation directly above the field

Is there any need for setter methods with field injection? No ... so they could be commented out or removed

Is field injection recommended? No

Annotation Type Autowired

@Target (value ={CONSTRUCTOR ,METHOD ,PARAMETER ,FIELD ,ANNOTATION\_TYPE })

@Retention (value =RUNTIME )

@Documented

public @interface Autowired

Marks a constructor, field, setter method, or config method as to be autowired by Spring's dependency injection facilities. This is an alternative to the JSR-330 Inject annotation, adding required-vs-optional semantics.

Autowired Constructors

Only one constructor of any given bean class may declare this annotation with the required() attribute set to true, indicating the constructor to autowire when used as a Spring bean. Furthermore, if the required attribute is set to true, only a single constructor may be annotated with @Autowired. If multiple non-required constructors declare the annotation, they will be considered as candidates for autowiring. The constructor with the greatest number of dependencies that can be satisfied by matching beans in the Spring container will be chosen. If none of the candidates can be satisfied, then a primary/default constructor (if present) will be used. Similarly, if a class declares multiple constructors but none of them is annotated with @Autowired, then a primary/default constructor (if present) will be used. If a class only declares a single constructor to begin with, it will always be used, even if not annotated. An annotated constructor does not have to be public.

Autowired Fields

Fields are injected right after construction of a bean, before any config methods are invoked. Such a config field does not have to be public.

Autowired Methods

Config methods may have an arbitrary name and any number of arguments; each of those arguments will be autowired with a matching bean in the Spring container. Bean property setter methods are effectively just a special case of such a general config method. Such config methods do not have to be public.

Autowired Parameters

Although @Autowired can technically be declared on individual method or constructor parameters since Spring Framework 5.0, most parts of the framework ignore such declarations. The only part of the core Spring Framework that actively supports autowired parameters is the JUnit Jupiter support in the spring-test module (see the TestContext framework reference documentation for details).

Multiple Arguments and 'required' Semantics

In the case of a multi-arg constructor or method, the required() attribute is applicable to all arguments. Individual parameters may be declared as Java-8 style Optional or, as of Spring Framework 5.0, also as @Nullable or a not-null parameter type in Kotlin, overriding the base 'required' semantics.

Autowiring Arrays, Collections, and Maps

In case of an array, Collection, or Map dependency type, the container autowires all beans matching the declared value type. For such purposes, the map keys must be declared as type String which will be resolved to the corresponding bean names. Such a container-provided collection will be ordered, taking into account Ordered and @Order values of the target components, otherwise following their registration order in the container. Alternatively, a single matching target bean may also be a generally typed Collection or Map itself, getting injected as such.

Not supported in BeanPostProcessor or BeanFactoryPostProcessor

Note that actual injection is performed through a BeanPostProcessor which in turn means that you cannot use @Autowired to inject references into BeanPostProcessor or BeanFactoryPostProcessor types. Please consult the javadoc for the AutowiredAnnotationBeanPostProcessor class (which, by default, checks for the presence of this annotation).

Since:

2.5

Author:

Juergen Hoeller, Mark Fisher, Sam Brannen

Can you use the @Qualifier annotation for all three kinds of injection? Yes

What is Java Reflection? Java Reflection is the process of analyzing and modifying all the capabilities of a class at runtime. Reflection API in Java is used to manipulate class and its members which include fields, methods, constructor, etc. at runtime. ... reflect package provides many classes to implement reflection java.

Why is field injection not recommended? Injection types

There are three options for how dependencies can be injected into a bean:

Through a constructor

Through setters or other methods

Through reflection, directly into fields

You are using option 3. That is what is happening when you use @Autowired directly on your field.

Injection guidelines

A general guideline, which is recommended by Spring (see the sections on Constructor-based DI or Setter-based DI) is the following:

For mandatory dependencies or when aiming for immutability, use constructor injection

For optional or changeable dependencies, use setter injection

Avoid field injection in most cases

Field injection drawbacks

The reasons why field injection is frowned upon are as follows:

You cannot create immutable objects, as you can with constructor injection

Your classes have tight coupling with your DI container and cannot be used outside of it

Your classes cannot be instantiated (for example in unit tests) without reflection. You need the DI container to instantiate them, which makes your tests more like integration tests

Your real dependencies are hidden from the outside and are not reflected in your interface (either constructors or methods)

It is really easy to have like ten dependencies. If you were using constructor injection, you would have a constructor with ten arguments, which would signal that something is fishy. But you can add injected fields using field injection indefinitely. Having too many dependencies is a red flag that the class usually does more than one thing, and that it may violate the Single Responsibility Principle.

Conclusion

Depending on your needs, you should primarily use constructor injection or some mix of constructor and setter injection. Field injection has many drawbacks and should be avoided. The only advantage of field injection is that it is more convenient to write, which does not outweigh all the cons.

Annotations - Default Bean Names - The Special Case

Annotations - Default Bean Names ... and the Special Case

In general, when using Annotations, for the default bean name, Spring uses the following rule.

If the annotation's value doesn't indicate a bean name, an appropriate name will be built based on the short name of the class (with the first letter lower-cased).

For example:

HappyFortuneService --> happyFortuneService

---

However, for the special case of when BOTH the first and second characters of the class name are upper case, then the name is NOT converted.

For the case of RESTFortuneService

RESTFortuneService --> RESTFortuneService

No conversion since the first two characters are upper case.

Behind the scenes, Spring uses the Java Beans Introspector to generate the default bean name. Here's a screenshot of the documentation for the key method.

Also, here's a link to the documentation.

- https://docs.oracle.com/javase/8/docs/api/java/beans/Introspector.html#decapitalize(java.lang.String)

---

As always, you can specify a name for your bean.

@Component("foo")

public class RESTFortuneService .... {

}

Then you can access it using the name of "foo".

How do you use @Qualifier with Constructors, specifically? @Qualifier is a nice feature, but it is tricky when used with Constructors.

The syntax is much different from other examples and not exactly intuitive.

You have to place the @Qualifier annotation inside of the constructor arguments.

Here's an example from our classroom example. I updated it to make use of constructor injection, with @Autowired and @Qualifier.

package com.luv2code.springdemo;

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

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

@Component

public class TennisCoach implements Coach {

private FortuneService fortuneService;

// define a default constructor

public TennisCoach() {

System.out.println(">> TennisCoach: inside default constructor");

}

@Autowired

public TennisCoach (@Qualifier("randomFortuneService") FortuneService theFortuneService) {

System.out.println(">> TennisCoach: inside constructor using @autowired and @qualifier");

fortuneService = theFortuneService;

}

How to inject properties file using Java annotations This solution will show you how inject values from a properties file using annotatons. The values will no longer be hard coded in the Java code.

1. Create a properties file to hold your properties. It will be a name value pair.

New text file: src/sport.properties

foo.email=myeasycoach@luv2code.com

foo.team=Silly Java Coders

Note the location of the properties file is very important. It must be stored in src/sport.properties

2. Load the properties file in the XML config file.

File: applicationContext.xml

Add the following lines:

<context:property-placeholder location="classpath:sport.properties"/>

This should appear just after the <context:component-scan .../> line

3. Inject the properties values into your Swim Coach: SwimCoach.java

@Value("${foo.email}")

private String email;

@Value("${foo.team}")

private String team;

What three things does a bean's scope indicate? 1. how long the bean lives

2. how many instances are created

3. how the bean is shared

What is the default scope for a bean? Singleton

What are three characteristics of a bean with singleton scope? 1. It's a single instance of the bean.

2. It's cached in memory.

3. All requests for the bean will return a shared reference to the same bean.

Which scope tells Spring to make new instances of a bean? Prototype

What are the ways to set the scope of a bean? 1. In the first bean tag in a pair of tags in an XML file

2. As a @Scope annotation under the @Component bean annotation that's above the class declaration, with the kind of scope as a parameter to @Scope

Characteristics of method signatures for @PostConstruct and @PreDestroy are the same as those for the xml-config-file-only lifecycle methods init-method and destroy-method. Access modifier

The method can have any access modifier (public, protected, private)

Return type

The method can have any return type. However, "void' is most commonly used. If you give a return type just note that you will not be able to capture the return value. As a result, "void" is commonly used.

Method name

The method can have any method name.

Arguments

The method can not accept any arguments. The method should be no-arg.

HEADS UP - FOR JAVA 9 USERS - @PostConstruct and @PreDestroy If you are using Java 9 or higher, then you will encounter an error when using @PostConstruct and @PreDestroy in your code.

These are the steps to resolve it. Come back to the lecture if you hit the error.

Error

Eclipse is unable to import @PostConstruct or @PreDestroy

This happens because of Java 9 and higher.

When using Java 9 and higher, javax.annotation has been removed from its default classpath. That's why we Eclipse can't find it.

---

Solution

1. Download the javax.annotation-api-1.3.2.jar from

https://search.maven.org/remotecontent?filepath=javax/annotation/javax.annotation-api/1.3.2/javax.annotation-api-1.3.2.jar

2. Copy the JAR file to the lib folder of your project

---

Use the following steps to add it to your Java Build Path.

3. Right-click your project, select Properties

4. On left-hand side, click Java Build Path

5. In top-center of dialog, click Libraries

6. Click Classpath and then Click Add JARs ...

7. Navigate to the JAR file <your-project>/lib/javax.annotation-api-1.3.2.jar

8. Click OK then click Apply and Close

Eclipse will perform a rebuild of your project and it will resolve the related build errors.

Special Note about Destroy Lifecycle and Prototype Scope

Here is a subtle point you need to be aware of with "prototype" scoped beans.

For "prototype" scoped beans, Spring does not call the @PreDestroy method. Gasp!

I didn't know this either until I dug through the Spring reference manual researching a student's question.

Here is the answer from the Spring reference manual. Section 1.5.2

https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-scopes-prototype

---

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

Thus, although initialization lifecycle callback methods are called on all objects regardless of scope, in the case of prototypes, configured destruction lifecycle callbacks are not called. The client code must clean up prototype-scoped objects and release expensive resources that the prototype bean(s) are holding.

To get the Spring container to release resources held by prototype-scoped beans, try using a custom bean post-processor, which holds a reference to beans that need to be cleaned up.

---

This also applies to XML configuration.

---

QUESTION: How can I create code to call the destroy method on prototype scope beans

ANSWER:

You can destroy prototype beans but custom coding is required. This examples shows how to destroy prototype scoped beans.

1. Create a custom bean processor. This bean processor will keep track of prototype scoped beans. During shutdown it will call the destroy() method on the prototype scoped beans.

2. The prototype scoped beans MUST implement the DisposableBean interface. This interface defines a "destroy()" method. This method should be used instead of the @PreDestroy annotation.

3. In this app, BeanProcessorDemoApp.java is the main program. TennisCoach.java is the prototype scoped bean. TennisCoach implements the DisposableBean interface and provides the destroy() method. The custom bean processing is handled in the MyCustomBeanProcessor class.

See source code here for details: destroy-protoscope-bean-with-custom-processor.zip

What are the three ways to configure a Spring container? 1. Full XML Config (without annotations ... beans listed in the xml file)

2. XML Component Scan (with annotations)

3. Java Configuration Class (Java source code ... No XML!)

What is the development process for making Java source code configuration? 1. Create a Java class and we annotate it as @Configuration

2. Add component scanning support

with @ComponentScan (this is an optional step)

3. Read the Java configuration class in our main application

4. Retrieve the bean from the Spring container

What is the development process for defining beans in a Java configuration file (no xml)? 1. Define methods to expose the bean

2. Inject the bean dependencies

3. Read the Spring configuration class

in our main program

4. Retrieve the bean from the Spring container.

What is the equivalent to the "bean id" when defining a bean in a Java source code configuration? The method name under the @Bean annotation

Where are all the beans defined in a Java source code configuration? The Java configuration file

Is there component scanning when the beans are defined in a Java source code configuration file? No

FAQ: How @Bean works behind the scenes

Question:

During All Java Configuration, how does the @Bean annotation work in the background?

Answer

This is an advanced concept. But I'll walk through the code line-by-line.

For this code:

@Bean

public Coach swimCoach() {

SwimCoach mySwimCoach = new SwimCoach();

return mySwimCoach;

}

At a high-level, Spring creates a bean component manually. By default the scope is singleton. So any request for a "swimCoach" bean, will get the same instance of the bean since singleton is the default scope.

However, let's break it down line-by-line

@Bean

The @Bean annotation tells Spring that we are creating a bean component manually. We didn't specify a scope so the default scope is singleton.

public Coach swimCoach(){

This specifies that the bean will bean id of "swimCoach". The method name determines the bean id. The return type is the Coach interface. This is useful for dependency injection. This can help Spring find any dependencies that implement the Coach interface.

The @Bean annotation will intercept any requests for "swimCoach" bean. Since we didn't specify a scope, the bean scope is singleton. As a result, it will give the same instance of the bean for any requests.

SwimCoach mySwimCoach = new SwimCoach();

This code will create a new instance of the SwimCoach.

return mySwimCoach;

This code returns an instance of the swimCoach.

----

Now let's step back and look at the method in it's entirety.

@Bean

public Coach swimCoach() {

SwimCoach mySwimCoach = new SwimCoach();

return mySwimCoach;

}

It is important to note that this method has the @Bean annotation. The annotation will intercept ALL calls to the method "swimCoach()". Since no scope is specified the @Bean annotation uses singleton scope. Behind the scenes, during the @Bean interception, it will check in memory of the Spring container (applicationContext) and see if this given bean has already been created.

If this is the first time the bean has been created then it will execute the method as normal. It will also register the bean in the application context. So that is knows that the bean has already been created before. Effectively setting a flag.

The next time this method is called, the @Bean annotation will check in memory of the Spring container (applicationContext) and see if this given bean has already been created. Since the bean has already been created (previous paragraph) then it will immediately return the instance from memory. It will not execute the code inside of the method. Hence this is a singleton bean.

The code for

SwimCoach mySwimCoach = new SwimCoach();

return mySwimCoach;

is not executed for subsequent requests to the method public Coach swimCoach() . This code is only executed once during the initial bean creation since it is singleton scope.

That explains how @Bean annotation works for the swimCoach example.

====

Now let's take it one step further.

Here's your other question

>> Please explain in detail whats happening behind the scene for this statement.

return new SwimCoach(sadFortuneService())

The code for this question is slightly different. It is injecting a dependency.

In this example, we are creating a SwimCoach and injecting the sadFortuneService().

// define bean for our sad fortune service

@Bean

public FortuneService sadFortuneService() {

return new SadFortuneService();

}

// define bean for our swim coach AND inject dependency

@Bean

public Coach swimCoach() {

SwimCoach mySwimCoach = new SwimCoach(sadFortuneService());

return mySwimCoach;

}

Using the same information presented earlier

The code

// define bean for our sad fortune service

@Bean

public FortuneService sadFortuneService() {

return new SadFortuneService();

}

In the code above, we define a bean for the sad fortune service. Since the bean scope is not specified, it defaults to singleton.

Any calls for sadFortuneService, the @Bean annotation intercepts the call and checks to see if an instance has been created. First time through, no instance is created so the code executes as desired. For subsequent calls, the singleton has been created so @Bean will immediately return with the singleton instance.

Now to the main code based on your question.

return new SwimCoach(sadFortuneService())

This code creates an instance of SwimCoach. Note the call to the method sadFortuneService(). We are calling the annotated method above. The @Bean will intercept and return a singleton instance of sadFortuneService. The sadFortuneService is then injected into the swim coach instance.

FAQ: What is a real-time use case for @Bean?

Here is a real-time use case of using @Bean: You can use @Bean to make an existing third-party class available to your Spring framework application context.

For example, I was recently working on a global real-time project using Amazon Web Services. The project made use of the Amazon Simple Storage Service (AWS S3). This is remote service that provides object storage in the cloud. You can think of AWS S3 at a high-level as a remote file server for storing files (pdfs, pngs etc).

Our Spring application needed to integrate with AWS S3 and store pdf documents. Amazon provides an AWS SDK for integrating with AWS S3. Their API provides a class, S3Client. This is a regular Java class that provides a client interface to the AWS S3 service. We needed to share the S3Client object in various services in our Spring application. However, the S3Client does not have the @Component annotation. The S3Client does not use Spring.

Since the S3Client is part of the AWS framework, we can't modify the source code for the S3Client directly. We can't simply add the @Component annotation to the S3Client source code. As a result, we need an alternative solution.

But no problem, by using the @Bean annotation, I can wrap this third-party class, S3Client, as a Spring bean. And then once it is wrapped using @Bean, it is as a singleton object and available in our Spring framework application context. I can now easily share this bean in my app using dependency injection and @Autowired. So think of the @Bean annotation was a wrapper / adapter for third-party classes. You want to make the third-party classes available to your Spring framework application context.

Here's a real-time example

Here is a snippet from our @Configuration class. We create an instance of the S3Client and wrap it as a Spring bean. The default scope is singleton. It is now available in our application context and we can inject it to other parts of our Spring application using @Autowired.

@Bean

public S3Client remoteClient() {

// Create an S3 client to connect to AWS S3

S3Client s3Client = S3Client.builder().region(Region.of(region))

.credentialsProvider(StaticCredentialsProvider.create(awsCreds)).build();

return s3Client;

}

---

In the code below, this is a Spring service that uses the S3Client. The service @Service annotation is a subclass of @Component. This code uses @Autowired to inject the bean named "remoteClient". This bean was created in the configuration code above using @Bean.

Once the bean is injected, then our method can use this to interact with the Amazon S3 service. In this real-time project, we were processing insurance claims. We store the PDF invoices in the cloud using the AWS S3 service.

@Service

public class InsuranceClaimsServiceImpl implements ClaimsService {

@Autowired

private S3Client remoteClient;

...

public void processClaim(Claim theClaim) {

// read claim data

FileData fileData = theClaim.getFileData("payerInvoice");

String fileName = theClaim.getSubmittedFileName();

// get the input stream and file size

InputStream fileInputStream = fileData.getInputStream();

long contentLength = fileData.getSize();

//

// store claim data in AWS S3

//

// Create a put request for the object

PutObjectRequest putObjectRequest = PutObjectRequest.builder()

.bucket(bucketName)

.key(subDirectory + "/" + fileName)

.acl(ObjectCannedACL.BUCKET\_OWNER\_FULL\_CONTROL).build();

// perform the putObject operation to AWS S3 ... using our autowired bean

remoteClient.putObject(putObjectRequest, RequestBody.fromInputStream(fileInputStream, contentLength))

}

}

As you can see, I was able to wrap a third-party class as a Spring bean. The AWS S3Client object was not originally annotated with @Component. The S3Client is not aware of Spring. But I could manually wrap it using @Bean. By doing this, the object is now available in our Spring application context. We can now share/reuse this bean in other areas of our Spring app by using dependency injection and @Autowired.

For other services in our application, if they need access to the S3client (singleton) then they can simply inject it using @Autowired. No need for each service to create a new instance of the S3Client every time. This keeps the application efficient in terms of memory and performance.

---

In summary: You can use @Bean to make an existing third-party class available to your Spring framework application context.

What are five benefits of Spring MVC? 1. It's Spring's way of building

web apps using Java.

2. You can leverage a set of reusable UI

components, and these are available in the form of Spring JSP custom tags.

3. You can manage your application state for web requests for session tracking, or application tracking

4. You can process your form data, so you can also perform validation on the form data, conversion, etc.

5. It has a very flexible configuration for the view layers, so you're not limited to only using JSP; you can use other view layers like Thymeleaf, Velocity, or FreeMarker.

What are the three main components of Spring MVC? 1. A collection of web pages to lay out your UI components

2. A collection of Spring beans (controllers, services, etc.)

3. Spring configuration (XML, Annotations, or pure Java)

How does the flow of Spring MVC go? Everything starts off with that first

incoming request and it encounters something

called a front controller.

So the front controller is known as

the dispatcher servelet.

This front controller's actually part

of the Spring framework.

It's already developed by the Spring development team

so you don't have to create this.

This is part of the Spring jar files

that you download,

so it's given to you for free.

So out of the box you have his front controller.

What this front controller will do

is it will actually delegate the request

to some other objects or items

here in our system.

As a developer, you will create

the model, the view, and the controller MVC.

So the model objects are in orange.

The model objects contain data.

The view templates,

that are in dark green,

that's your actual JSP page,

or your view page to actually

render the data.

And then C,

the controller classes that's in yellow,

that's your actual business logic

or your processing logic.

Coming up next, I'll actually talk about

each one of these components in detail.

Let's start with controller.

So when the front controller has a request,

it delegates the request to the controller.

That's in yellow.

The controller is the code that you will actually create.

Basically in this controller,

this contains your business logic.

So this is where you'll handle the request

where you'll maybe read some form data

then you'll take this data

and store it or retrieve it.

You may store it into a database or

retrieve information from a web service.

Basically, once you have your data

and you're using it,

then you can take that data

and place it into the model.

So the model is just a container for your data

and then you pass it back to the appropriate

view template.

So again, your controller,

code that you create,

contains your business logic,

and it handles the web request.

So let's talk about the model.

As I mentioned earlier,

the model contains your data.

So when your controller goes off

and performs an operation to retrieve data

from a backend system, like a database or web service,

or a Spring Bean,

you can take that data and place it into the model.

So the model again is your container,

like your suitcase or your luggage,

for shipping data between various parts

of your Spring MVC application.

So that model data will actually get passed

over to the view template

and they can actually handle that

for displaying the data.

That's kinda where we're at right now.

So the view template.

The most common view template that you'll use

is JSP and JSTL.

Spring MVC is very flexible.

There's many different view template types.

I'll talk about that in a second.

But for now let's just think about JSP, JSTL.

This model data comes over to your view template

and then your JSP page can read that model data

and display it.

So, say for example,

you have a list of students,

or list of products,

then your JSP page can create a table

to display that product list

or that student list.

Or, say for example, somebody is signing

up for an airline flight,

or is signing up for a class,

then your view template,

or your page can give them a confirmation,

hey, you're registered for the class,

here's your confirmation number.

So that's the idea of the view template.

It's basically a JSP page

that will provide data to the user.

Now as I mentioned,

Spring MVC is very flexible on the view templates.

There's actually other view templates

that are supported,

so if you don't wanna use JSP,

you can make use of some other templates

like Thymeleaf, Groovy, Velocity, Freemarker,

the list goes on.

You can plug in all different types of view templates.

If you'd like to get details on this,

go to this link I have on the screen,

luv2code.com/spring-mvc-views.

This'll redirect you to the official Spring documentation

and you can get more information

on the various other templates

that are out there and available.

Alright, good.

This is a quick behind the scenes view

of Spring MVC.

You learned about the model, the view, the controller

and a conceptual theoretical level.

In the following videos we're gonna

start getting our hands dirty

and setting up our environment

and then starting to write some code.

So I'll see you there.

In this video,

I'm gonna give you a behind the scenes tour

of Spring MVC.

What are the components of a Spring MVC application?

Basically it's a collection of web pages

to layout your UI components.

It's also a collection of Spring beans,

for controlling, handling services,

and so on.

And then finally you have your Spring configuration.

You can choose XML, Annotations, or pure Java.

And those are kinda the main components

of a Spring MVC application.

Now, how does Spring MVC work behind the scenes?

In a previous video I showed you this

little diagram here of the process model.

What I wanna do is actually take this

little process model and kind of break it down,

and kinda dig into it a little bit,

and kinda go step-by-step.

Do a little deep dive on how the

whole process flow works.

Everything starts off with that first

incoming request and it encounters something

called a front controller.

So the front controller is known as

the dispatcher servelet.

This front controller's actually part

of the Spring framework.

It's already developed by the Spring development team

so you don't have to create this.

This is part of the Spring jar files

that you download,

so it's given to you for free.

So out of the box you have his front controller.

What this front controller will do

is it will actually delegate the request

to some other objects or items

here in our system.

As a developer, you will create

the model, the view, and the controller MVC.

So the model objects are in orange.

The model objects contain data.

The view templates,

that are in dark green,

that's your actual JSP page,

or your view page to actually

render the data.

And then C,

the controller classes that's in yellow,

that's your actual business logic

or your processing logic.

Coming up next, I'll actually talk about

each one of these components in detail.

Let's start with controller.

So when the front controller has a request,

it delegates the request to the controller.

That's in yellow.

The controller is the code that you will actually create.

Basically in this controller,

this contains your business logic.

So this is where you'll handle the request

where you'll maybe read some form data

then you'll take this data

and store it or retrieve it.

You may store it into a database or

retrieve information from a web service.

Basically, once you have your data

and you're using it,

then you can take that data

and place it into the model.

So the model is just a container for your data

and then you pass it back to the appropriate

view template.

So again, your controller,

code that you create,

contains your business logic,

and it handles the web request.

So let's talk about the model.

As I mentioned earlier,

the model contains your data.

So when your controller goes off

and performs an operation to retrieve data

from a backend system, like a database or web service,

or a Spring Bean,

you can take that data and place it into the model.

So the model again is your container,

like your suitcase or your luggage,

for shipping data between various parts

of your Spring MVC application.

So that model data will actually get passed

over to the view template

and they can actually handle that

for displaying the data.

That's kinda where we're at right now.

So the view template.

The most common view template that you'll use

is JSP and JSTL.

Spring MVC is very flexible.

There's many different view template types.

I'll talk about that in a second.

But for now let's just think about JSP, JSTL.

This model data comes over to your view template

and then your JSP page can read that model data

and display it.

So, say for example,

you have a list of students,

or list of products,

then your JSP page can create a table

to display that product list

or that student list.

Or, say for example, somebody is signing

up for an airline flight,

or is signing up for a class,

then your view template,

or your page can give them a confirmation,

hey, you're registered for the class,

here's your confirmation number.

So that's the idea of the view template.

It's basically a JSP page

that will provide data to the user.

Now as I mentioned,

Spring MVC is very flexible on the view templates.

There's actually other view templates

that are supported,

so if you don't wanna use JSP,

you can make use of some other templates

like Thymeleaf, Groovy, Velocity, Freemarker,

the list goes on.

You can plug in all different types of view templates.

Is @Autowired optional for Constructor Injection? Yes, This is a new feature of Spring 4.3.

Here is the snippet from the Spring Docs.

Section 1.9.2: Autowired

As of Spring Framework 4.3, an @Autowired annotation on such a constructor is no longer necessary if the target bean only defines one constructor to begin with. However, if several constructors are available, at least one must be annotated to teach the container which one to use.

I personally prefer to use the @Autowired annotation because it makes the code more readable. But as mentioned, the @Autowired is not required for this scenario.

What two annotations do you use to inject a properties in Spring? 1. @PropertySource("classpath:foo.properties"); in the Java config file, below the @Configuration annotation

2. @Value("${foo.property1}) in the relevant model, on top of each field declaration

how to configure Spring MVC.

So, here's the big picture of Spring MVC.

Remember, we have a Front Controller.

They dispatch something to a Controller class,

and also make use of View Templates,

and I'll show you how to get this set up

and how to get it configured in Spring MVC,

and I must admit, up front,

this is probably the hardest part

of getting started with Spring MVC,

is getting the configuration right,

so I'll have a step-by-step process,

and I'll show you how to get all of this working.

So, here's the first part of doing the

Spring MVC Configuration Process.

What we'll have to do is make some modifications

to our web.xml file.

So, the first thing is,

configuring a Spring MVC dispatcher servlet, and then

setting up a URL mapping to this dispatcher servlet.

And then, part two of the configuration process

is that we'll add some entries into a

Spring configuration file,

so here, we'll add support for a component scanning.

We'll also add support for

conversion, formatting, and validation.

And then, finally, we'll configure

the Spring MVC view resolver, and don't worry.

In the following slides,

I'll go through each of these, step-by-step.

Alright, let's start off with step one,

Configure the Spring MVC DispatcherServlet.

So, in our web.xml file, we need to add an entry

for the Spring DispatcherServlet, or the Front Controller,

so you put in a servlet reference,

you give the name and class of the servlet,

and, again, the DispatcherServlet is part of

the core Spring framework,

so you get it for free, in the Spring .jar files.

There's nothing that you have to create.

Once you have the servlet reference,

then you set up initial parameter,

so you basically tell it where your

Spring context configuration file is located.

In this example, we'll make use of

/WEB-INF/spring-mvc-demo-servlet.xml.

You can give any name you want, here,

as long as you reference it accordingly.

So, the next thing we need to do is set up

the URL mappings for the Spring MVC DispatcherServlet.

So, basically, what we want to do is

tell the system, "Hey, for any URL pattern coming in,

I'd like for you to pass it off to the DispatcherServlet."

So, in this case, our URL pattern's going to be slash,

meaning all web requests, coming in,

should be handled by the DispatcherServlet.

If you want to have a different pattern, here,

you can simply say slash FUBAR, slash STAR, or whatever,

to have it work with a different pattern,

but, here, I'm going to keep it simple and make use of

the slash pattern for handling all the requests.

Now, one thing that's important, here, is to know here

that servlet name matches with the servlet reference

that you just set up, higher up in this file,

or at the top, that was set up in step one,

so those two items have to match up.

Okay, moving to the next step,

Add support for component scanning, so in our file,

this is our Spring file, spring-mvc-demo-servlet.xml,

we simply add context:component-scan.

And, again, this works just like we've learned so far,

that it'll basically scan this package

for any special Spring beans, and make them available,

so any @component items out there,

it'll make them available,

and put them into the Spring context.

Alright, step four, adding support for conversion,

formatting, and validation.

So, when you make use of spring-mvc,

it could perform conversions of form data.

It can also format form data, for you,

and you can also perform form validation,

so in order to get the support,

you make use of this mvc:annotation-driven.

And then the final step here,

is Configuring the Spring MVC View Resolver, so,

really, a fancy name for, "How do we display the pages?"

Or, "Where are the pages located?"

So, at the bottom, here, we make use of

this internal resource, ViewResolver, and then

we give a prefix, and then a suffix.

And you're probably wondering,

"What's up with the prefix and suffix?"

(laughs) Well, when your app provides a "view" name,

Spring MVC will automatically prepend the prefix

and append the suffix.

Alright, so it's basically telling it

where to look for files,

to actually render the view for your application.

So, let's take a look at an example, here.

So, say, for example, we returned a view name of

show-student-list.

Automatically, Spring will prepend /WEB-INF/view,

because that's the prefix,

and then we'll have the actual view name,

and they'll append the suffic, .jsp.

So, that's the actual path of the view page

that will actually show.

Alright, so a little funny, a little tricky,

but once you understand this configuration file,

and you understand that Spring will automatically

do this work for you, then it'll work out just fine.

The MVC, you have the front controller,

you have a controller object, and a view template.

What we're gonna do here is actually

set up a request mapping for a given path,

and then we'll have a home controller

that's gonna handle the request,

then it's gonna forward it over to a view template,

which we'll call main-menu.jsp.

And it's very simple, very straightforward,

but I just wanna make sure we get

all the mechanics down for building this out.

All right, so again, I love this

step-by-step process, right?

The first thing we're gonna do is

create the controller class,

then we'll define the controller method.

Next we'll add request mappings

to the controller method,

and then we'll return the view name,

and then finally we'll develop the view page.

Woo, seems like a lot of stuff, right?

But don't worry, we'll go through all of this step-by-step,

and we'll do it all from scratch.

All right, so let's go ahead and start with step one,

creating the controller class.

Basically, you define a class,

and then you annotate it with @Controller.

Now, @Controller basically says that

this is a Spring MVC controller,

and also, one thing that's really cool is that

@Controller inherits from @Component,

so it's really just a specialized component

that supports web MVC.

The nice thing about this is that,

when Spring does its component scanning,

then it'll also pick up @Controllers

because they inherit or extend from @Component.

So this is the basic process here

of defining a controller class.

All right, so let's go ahead and take a look at

step two, defining a controller method.

So, here I have a method, public String showMyPage.

Now, this is a very simple method.

The actual method name is flexible.

You can give any method name that you want.

Here I called it showMyPage,

but I could've easily called it fubar.

The return type here is String.

I'm simply gonna return the actual view name

that I want them to show.

But again, this method here is actually very flexible.

So, in Spring, when you define your controller method,

you can actually pass in any number of parameters,

like request parameters, session objects,

and so on, or model objects.

And you can also return different variations here.

What does Spring Data REST do? So Spring Data REST will basically scan our project

for a JpaRepository

Expose REST APIs for each entity type

for our JpaRepository.

How do Spring Data REST endpoints work? So by default, Spring Data REST will create endpoints

based on the entity type.

It'll make use of the simple, pluralized form.

So it'll basically take the first character

of the entity type and make it lowercase

and then just add an S to the entity.

How to make a Read-only, etc. controller in Spring Data Rest? Option one is, simply don't use Spring Data REST.

So in this case,

we could manually create our own REST controller,

and manually define methods for access using @GetMapping.

But we'd loose the Spring Data support for paging,

sorting, and so on.

Those are some really nice features.

I don't want to loose those.

And the other option is to use Spring Data REST,

but configure it to disable certain HTTP methods,

like POST, DELETE, et cetera.

And this is the one that we'll choose.

Example of @RequestParam binding annotation @RequestMapping("/processNewGroup")

public String processNewGroup(

@RequestParam("groupModerator") String theName,

Model model) {

//This line is no longer needed, with @RequestParam

// String theName = req.getParameter("groupModerator");

theName = theName.toUpperCase();

String result = "Welcome, " + theName + "!";

model.addAttribute("message", result);

return "processNewGroup";

}

What is request mapping at the controller level? You can actually define a request mapping

at the controller level, so it basically

serves as like the parent mapping for the controller.

All of the request mappings on the methods

are actually relative to the controller's path.

And this is very similar to setting up

a folder directory structure where you have

a parent directory, and you have subdirectories

underneath it.

all of the individual method mappings are relative to

the actual controller mapping.

Benefits of request mapping at the controller level? 1. It's a very nice way

for you to group your request mappings together,

2. It's a great technique for resolving any problems or conflicts that you may have with other request mappings.

What is a good example of how request mapping at the controller level can resolve conflicts that may occur with more than one request mapping? When more than one controller have the same name for a form

Example for request mapping at the controller level @Controller

@RequestMapping("/writersGroupMain")

public class WritersGroupsController {

@RequestMapping("/showNewGroupForm")

private String showNewGroupForm() {

return "showNewGroupForm";

}

How can you explain the refactoring to allow request mapping at the controller level? The key here is relative path of showing the form and then submitting to relative path.

ME: The parent @RequestMapping at the controller level matches the added parent directory in the url to the form

More, from the tutorial:

FAQ:

Question: Can you please clarify how /hello is getting appended to the jsp file action for "processForm"?

Answer

You can use "processForm" because it is a relative path to the controller "/hello" request mapping. Here is how it works.

1. When you wish to view the form, the HTML link points to "hello/showForm". This calls the controller and it displays the form.

2. At this point the browser URL/path is: http://localhost:8080/spring-mvc-demo/hello

3. The HTML form uses "processForm" for the form action. Notice that it does not have a forward slash, as a result, this will be relative to the current browser URL. Since the current browser URL is

http://localhost:8080/spring-mvc-demo/hello

Then the actual form URL submission will send it to

http://localhost:8080/spring-mvc-demo/hello/processForm

The part in bold with map to the controller with top-level request mapping "/hello" and then map to request mapping in that class "/processForm"

Where does Spring MVC form configuration tag go? At the beginning of the JSP page

What is an example of a Spring MVC form configuration tag? <%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>

What are most common Spring MVC form tags? form:formIt is a container tag that contains all other form tags.form:inputThis tag is used to generate the text field.form:radiobuttonThis tag is used to generate the radio buttons.form:checkboxThis tag is used to generate the checkboxes.form:passwordThis tag is used to generate the password input field.form:selectThis tag is used to generate the drop-down list.form:textareaThis tag is used to generate the multi-line text field.form:hiddenThis tag is used to generate the hidden input field.

The Spring MVC form:form tag? A container tag. It is a parent tag that contains all the other tags of the tag library. This tag generates an HTML form tag and exposes a binding path to the inner tags for binding.

What is the syntax for the Spring MVC form:form tag Syntax

<form:form action="nextFormPath" modelAttribute=?abc?>

What is a model attribute for a form? It is basically a bean that's used to hold the form data,

and this will give us support for data binding, so we'll actually bind form data to our bean.

And remember, theModel is an object that we can use to

kind of pass data around between controllers and views.

creating an empty object with new keyword

that I'll pass to the form

for the form to make use of with data binding (so a null constructor parameter is acceptable if a parameter is required)

So I say form:form, I give the action,

and then the modelAttribute= student.

Again, that's the same attribute name

that I used on the previous slide

for writing the code for showing this form,

and then I go ahead and just list out

my two text fields here.

First name, I give form:input,

that will generate a text field.

I'll set the path to firstName.

Last name, form:input, path= lastName,

and finally, a submit button.

Now, you're probably wondering,

what's up with the path= firstName

and the path= lastName, kind of how does that map out?

Well, this actually binds this form field

to a property on the bean.

When the form is first loaded,

what Spring MVC will do behind the scenes is that

they'll actually use this to populate the form field.

When they have path= firstName,

Spring MVC will call student.getFirstName,

and they use that model attribute from up top to

retrieve that data.

If it's null, then that form field will simply be empty.

So in our case of creating a new student from scratch,

then it'll be empty,

but you can easily prepopulate that

accordingly if you'd like.

We'll do a similar thing here for last name,

form:input path= lastName.

Behind the scenes, Spring MVC will call student.getLastName.

Now, when the form is submitted,

Spring will actually call the setter methods.

All right, so they're real simple.

When they submit, they call setter methods.

So here for first name, form:input path= firstName,

Spring will call student.setFirstName

and only use whatever data the user

entered there in the form field.

A similar thing for last name, student.setLastName.

Again, Spring will call the setter methods

when they actually submit the form.

All right, so the form data's been submitted.

Now, how do we actually read the data in our controller?

Well, in our controller,

we can simply make use of a new Spring annotation called

@ModelAttribute, and we give the name of the attribute.

So again, we're using student, in all lower case.

They'll actually bind that object

to this parameter being passed in theStudent.

Some work behind the scenes,

Spring will actually take that model attribute

and bind it to this variable here, theStudent,

and we can use it in our controller.

he key here is that Spring will actually

take all that form data,

bind it automatically to your object,

and then pass it into your controller

so you can make use of it.

So there's no need for you to do the manual

request.get parameter for each one of the fields.

Spring will handle all of that work for you,

and that's the real benefit

of using the Spring MVC form tags,

is they have actual data binding,

What is the best choice when you want to customize which HTTP methods are available in a Spring Data REST configuration? Make a custom Spring Data Rest configuration class

Here are the specifics of the choice:

So I'd like to disable the POST method,

allow the GET methods, disable the PUT method,

and also disable the DELETE method.

So how can we do this?

Well there's some possible solutions:

Option one is, simply don't use Spring Data REST.

So in this case,

we could manually create our own REST controller,

and manually define methods for access using @GetMapping.

But we'd loose the Spring Data support for paging,

sorting, and so on.

Those are some really nice features.

I don't want to loose those.

And the other option is to use Spring Data REST,

but configure it to disable certain HTTP methods,

like POST, DELETE, et cetera.

And this is the one that we'll choose.

Description and example of such a class:

So we're going to disable the HTTP methods

for POST, PUT, and DELETE.

So we'll create this special configuration class.

We'll have this MyDataRestConfig

that implements RepositoryRestConfigurer.

And this interface is from the Spring Data REST API.

And we'll simply override this method

or implement this method,

for ConfigureRepositoryRestConfiguration.

We'll simply set up a list of HTTP methods

to disable for PUT, POST, DELETE.

And then in the configuration,

we'll say that it's for this given domain type,

Product.class.

So this is for the product repo.

And we'll set up the configuration

for single item and also collection.

And then, this API makes use of the fancy

Java Lambdas annotation.

So we have a parens for metadata, httpMethods,

the arrow symbol, that's Java Lambdas.

And we specify the method here,

we get httpMethods.disable,

and we pass in that array that we created

a little earlier, theunsupportedActions.

And this block of code

will essentially disable the methods

for POST, PUT, and DELETE for our productRestRepository.

How to add path to style page in a JSP page? href="${pageContext.request.contextPath}/resources/css/style.css"

What is pageContext.request.contextPath in href="${pageContext.request.contextPath}/resources/css/style.css"? the proper app name

In this video, we're gonna learn

about Get Mapping and Post Mapping.

So, these are some new annotations

that were recently added to Spring.

And we're gonna see how we can apply these Get Mappings

and Post Mappings to our Spring code.

Now, when you send data over to a Spring MVC Controller,

you normally have an HTML form.

You send over a request.

The controller will process it and send you back a response.

Now, these are the most commonly used HTTP methods.

You can send over a GET request and also a POST request.

And they have other methods out there,

but we'll see the differences between GET and POST

since they're the most commonly used.

So, to send over data with the GET method,

you set up your form

and the key item here is you say method=GET.

So, in that case, when the form data is submitted,

it's actually appended to the end of the URL

as name/value pairs.

So, you'll have a URL with a question mark

and then you'll have field1 value1 and field2 value2

and so on.

And you may have seen this on the web.

If you go to different websites,

you'll see this big long URL

at the top of your browser location bar.

They're actually sending data back and forth

using the GET method.

Now, to handle a form submission,

as we've seen already in our Spring MVC code,

we simply set up an @RequestMapping

and you give the actual path.

Now, with this simple version that we have here,

it actually handles all of the HTTP methods.

So, this mapping will handle GET requests,

POST requests, and so on.

And, depending on your application structure,

this may be fine, or you may want to be able to constrain

or limit the actual methods that a given mapping supports.

So, here's a scenario

on how we can constrain the request mapping.

In this case, we want this mapping

to only handle GET requests.

So, here we say @RequestMapping

give path equals /processForm

comma method=RequestMethod.GET.

So again, with this little item here,

this mapping will only handle GET requests.

Any other requests being sent to this method

will get rejected.

Now, this new annotation actually provides a shortcut

so you don't have to write out all of that long text

like we saw before.

So, here you simply make use

of this new annotation called @GetMapping.

Again, it will only handle GET requests

and you simply give a path that you want to map it to

and that's basically it.

So, it's just a shorthanded way of making use

of this Get Map.

Now, let's take a look at sending data with the POST Method.

So here we set up our form, action equals processForm.

And then, method equals POST.

So, in this case, the form data is passed

in the body of the HTTP request message.

So, here's a diagram.

So you have your form sending the data across

and the request message going across.

You have your request headers and then you have the body.

That's where the actual form data resides

when you send it over using a POST method.

Now, we can also constrain a Request Mapping for POST.

So here, again, I simply say @RequestMapping.

I give the path, comma, method=RequestMethod.POST.

So again, this will only handle POST methods.

Any other request type coming across will get rejected.

And here's the new annotation shortcut.

So instead of typing out all that long information,

you simply use @PostMapping and you simply give a path.

And they just make it very simple

and easy to set up this configuration.

Alrighty, so I showed you how to send data over

using GET and also POST.

You may wonder, well, which one should I use?

Well, I like to say that GET is good for debugging

because you can see everything on the URL.

You can also bookmark and email the URL.

However, with GET requests,

there's limitations on data lengths.

So, depending on the browser,

they may truncate some of the data.

So, I say in general,

you're safe with using about 1000 characters.

Anything beyond 1000 characters length,

you may want to look at using the POST method.

The POST method, you can't bookmark or email the URL,

but the really nice thing about POST method

is that there's no limitations on data length.

So, if you have a very large form

or very large piece of data you need to send across,

you can make use of POST.

Also, another benefit here is that the POST method

can also send binary data.

So, if you need to do like a file attachment

or a file upload,

you can make use of the POST method for that.

So, those are kind of the pros and cons of each one

and you know, play around with it

and see which one works best for your application scenario.

So anyways, just as a recap,

we showed the new annotations

for @GetMapping and @PostMapping.

Why do we have to create Service layer what has the same functions as DAO layer? Is it necessary to create all this layers?

Answer

Agreed, there are a lot of layers. However this is the architecture that you will see on real world, complex Spring projects.

In our example, it is fairly simple. We simply delegate the calls to the DAO. So I agree, you could remove the service layer in this simple example and have controller call dao directly.

However, we added the service layer to leverage the Service Layer design pattern. On a much more complex project, we could use the service layer to integrate multiple data sources (daos) and perform transaction management between the two. So, for a simple project that we have here ... this probably overkill. However, I wanted to show you design patterns that you will encounter on real projects.

Here are the benefits / use cases for use the service layer on a much larger project. In the video, I discuss benefits / use cases for the service layer

Video: Refactor - Add a Service Layer

====

And here is a another scenario where you would like to perform transaction management at the service layer.

You can use @Transactional at the service layer if you want DAO methods to run in the same transaction.

Say for example we have

BankDAO

- deposit(...)

- withdraw(...)

If we are transferring funds, we want that to run in the same transaction. By making use of @Transactional at service layer, then we can have this transactional support and both methods will run in the same transaction. This would call deposit() and withdraw(). If either of those methods failed then we'd want to roll the transaction back.

However, if we had @Transactional at DAO level instead of service level, then the methods deposit() and withdraw() would run in separate transactions. If one of them failed, then we would not be able to rollback the other method ... because it is in a separate transaction.

So that's one real-time project use case for applying @Transactional at the Service layer.

===

Of course, in your personal project, there is no strict requirement to use layers. In fact, there is no requirement to use DAO. You could add all of your code to one controller class. But from an architectural point of view, that would result in a poor design.