Wiring Beans

@Component. This simple annotation identifies this class as a component class and serves as a clue to Spring that a bean should be created for the class.

Component scanning isn’t turned on by default, however. You’ll still need to write an explicit configuration to tell Spring to seek out classes annotated with @Component and to create beans from them.

@Configuration

@ComponentScan

public class CDPlayerConfig {

}

With no further configuration, @ComponentScan will default to scanning the same

package as the configuration class.

In xml

<context:component-scan base-package="soundsystem" />

***Naming a component-scanned bean***

@Component("lonelyHeartsClub")

public class SgtPeppers implements CompactDisc {

...

}

@Named("lonelyHeartsClub")

public class SgtPeppers implements CompactDisc {

...

}

***base package for component scanning***

what if you want to scan a different package? Or what if you want to scan multiple base packages?

@Configuration

@ComponentScan(basePackages={"soundsystem", "video"})

public class CDPlayerConfig {}

or

@Configuration

@ComponentScan(basePackageClasses={CDPlayer.class, DVDPlayer.class})

public class CDPlayerConfig {}

**@component Vs @Bean**

<http://stackoverflow.com/questions/10604298/spring-component-versus-bean>

***Annotating beans to be automatically wired***

@Component

public class CDPlayer implements MediaPlayer {

privateCompactDisc cd;

@Autowired

publicCDPlayer(CompactDisc cd) {

this.cd = cd;

}

public void play() {

cd.play();

}

}

The @Autowired annotation’s use isn’t limited to constructors. It can also be used on a

property’s setter method

@Autowired

public void setCompactDisc(CompactDisc cd) {

this.cd = cd;

}

there’s nothing special about setter methods. @Autowired can also be

applied on any method on the class.

@Autowired

public void insertDisc(CompactDisc cd) {

this.cd = cd;

}

@Autowired is a Spring-specific annotation. If it troubles you to be scattering

Spring-specific annotations throughout your code for autowiring, you might consider

using the @Inject annotation instead:

importjavax.inject.Inject;

importjavax.inject.Named;

@Named

public class CDPlayer {

...

@Inject

publicCDPlayer(CompactDisc cd) {

this.cd = cd;

}

...

}

@Inject comes from the Java Dependency Injection specification, the same specification

that gave us @Named. Spring supports the @Inject annotation for autowiring

alongside its own @Autowired.

***Wiring beans with Java***

@Named and @Inject @Bean

importorg.springframework.context.annotation.Configuration;

@Configuration

public class CDPlayerConfig {

}

***Declaring a simple bean***

@Bean

publicCompactDiscsgtPeppers() {

return new SgtPeppers();

}

***Injecting with JavaConfig***

The simplest way to wire up beans in JavaConfig is to refer to the referenced bean’s method.

@Bean

publicCDPlayercdPlayer() {

return new CDPlayer(sgtPeppers());

}

So is it new Objet of Compact Disc will be created ?

It appears that the CompactDisc is provided by calling sgtPeppers, but that’s not exactly true. Because the sgtPeppers() method is annotated with @Bean, Spring will intercept any calls to it and ensure that the bean produced by that method is returned rather than allowing it to be invoked again.

Confusion : what will be name of two beans

@Bean

publicCDPlayercdPlayer() {

return new CDPlayer(sgtPeppers());

}

@Bean

publicCDPlayeranotherCDPlayer() {

return new CDPlayer(sgtPeppers());

}

how referring to a bean by calling its method can be confusing. There’s another way that might be easier to digest:

@Bean

publicCDPlayercdPlayer(CompactDisccompactDisc) {

return new CDPlayer(compactDisc);

}

This approach to referring to other beans is usually the best choice because it doesn’t depend on the CompactDisc bean being declared in the same configuration class.

***Wiring beans with XML***

Xml configulation file and @configuration are same thing.

The simplest possible Spring XML configuration looks like this:

<?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

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

<!-- configuration details go here -->

</beans>

***Initializing a bean with constructor injection***

 The<constructor-arg> element

 Using the c-namespace introduced in Spring 3.0

you could refer to the parameter’s position in the parameter list:

<bean id="cdPlayer" class="soundsystem.CDPlayer"

c:\_0-ref="compactDisc" />

INJECTING CONSTRUCTORS WITH LITERAL VALUES

<bean id="compactDisc"

class="soundsystem.BlankDisc">

<constructor-arg value="Sgt. Pepper's Lonely Hearts Club Band" />

<constructor-arg value="The Beatles" />

</bean>

***Setting properties***

Using the p-namespace, you can wire the compactDisc property like this:

<bean id="cdPlayer"

class="soundsystem.CDPlayer"

p:compactDisc-ref="compactDisc" />

CHOOSING BETWEEN CONSTRUCTOR INJECTION AND PROPERTY INJECTION :

As a general rule, I favor constructor injection for hard dependencies and propertyinjection for any optional dependencies.

<bean id="compactDisc"

class="soundsystem.BlankDisc"

p:title="Sgt. Pepper's Lonely Hearts Club Band"

p:artist="The Beatles">

<property name="tracks">

<list>

<value>Sgt. Pepper's Lonely Hearts Club Band</value>

<value>With a Little Help from My Friends</value>

<value>Lucy in the Sky with Diamonds</value>

<value>Getting Better</value>

<value>Fixing a Hole</value>

<!-- ...other tracks omitted for brevity... -->

</list>

</property>

</bean>

util-namespace offers is the <util:list>element, which creates a list bean.

<util:list id="trackList">

<value>Sgt. Pepper's Lonely Hearts Club Band</value>

<value>With a Little Help from My Friends</value>

<value>Lucy in the Sky with Diamonds</value>

<value>Getting Better</value>

<value>Fixing a Hole</value>

<!-- ...other tracks omitted for brevity... -->

</util:list>

***Importing and mixing configurations***

Fortunately, none of the configuration options available in Spring are mutually exclusive. You’re free to mix component scanning and autowiring with JavaConfigand/or XML configuration.

@Import and @ImportResource

@Configuration

@Import(CDPlayerConfig.class)

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

public class SoundSystemConfig {

}

@Configuration

@Import({CDPlayerConfig.class, CDConfig.class})

public class SoundSystemConfig {

}

***3.0 Advanced wiring***

Environments and profiles

Different beans for DEV and PROD ENV

**@Profile**

In version 3.1, Spring introduced bean profiles. To use profiles, you must gather all the varying bean definitions into one or more profiles and then make sure the proper profile is active when your application is deployed in each environment.

In Java configuration, you can use the @Profile annotation to specify which profile a bean belongs to.

It tells Spring that the beans in this configuration class should be created only if the devprofile is active. If the devprofile isn’t active, then the @Bean methods will be ignored.

@Configuration

public class DataSourceConfig {

@Bean(destroyMethod="shutdown")

@Profile("dev")

publicDataSourceembeddedDataSource() {

return new EmbeddedDatabaseBuilder()

.setType(EmbeddedDatabaseType.H2)

.addScript("classpath:schema.sql")

.addScript("classpath:test-data.sql")

.build();

}

@Bean

@Profile("prod")

publicDataSourcejndiDataSource() {

JndiObjectFactoryBeanjndiObjectFactoryBean =

newJndiObjectFactoryBean();

jndiObjectFactoryBean.setJndiName("jdbc/myDS");

jndiObjectFactoryBean.setResourceRef(true);

jndiObjectFactoryBean.setProxyInterface(javax.sql.DataSource.class);

return (DataSource) jndiObjectFactoryBean.getObject();

}

}

***Activating profiles***

Spring honors two separate properties when determining which profiles are active: spring.profiles.activeand spring.profiles.default. If spring.profiles.activeis set, then its value determines which profiles are active. But if spring.profiles.activeisn’t set, then Spring looks to spring.profiles.default.

There are several ways to set these properties:

As initialization parameters on DispatcherServlet

As context parameters of a web application

As JNDI entries

As environment variables

As JVM system properties

***Conditional beans***

Suppose you want one or more beans to be configured if and only if some library is available in the application’s classpath. Or let’s say you want a bean to be created only if a certain other bean is also declared. Maybe you want a bean to be created if and only if a specific environment variable is set.

For this purpose Spring 4 introduced a new @Conditional annotation that can be applied to @Bean methods.

For example, suppose you have a class named MagicBeanthat you only want Spring to instantiate if a magic environment property has been set. If the environment has no such property, then the MagicBeanshould be ignored.

@Bean

@Conditional(MagicExistsCondition.class)

Public MagicBeanmagicBean() {

return new MagicBean();

}

public class MagicExistsCondition implements Condition {

publicboolean matches(

ConditionContext context, AnnotatedTypeMetadata metadata) {

Environment env = context.getEnvironment();

returnenv.containsProperty("magic");

}

}

***Addressing ambiguity in autowiring***

But autowiring only works when exactly one bean matches the desired result.

When there’s more than one matching bean, the ambiguity prevents Spring from

autowiring the property, constructor argument, or method parameter

NoUniqueBeanDefinitionException:

@Qualifier and @Primary

@Autowired

@Qualifier("iceCream")

public void setDessert(Dessert dessert) {

this.dessert = dessert;

}

@Component

@Primary

public class IceCream implements Dessert { ... }

or

@Bean

@Primary

public Dessert iceCream() {

return new IceCream();

}

***Scoping beans***

Spring defines several scopes under which a bean can be created, including the

following:

*Singleton*—One instance of the bean is created for the entire application.

*Prototype*—One instance of the bean is created every time the bean is injected

into or retrieved from the Spring application context.

*Session*—In a web application, one instance of the bean is created for each session.

*Request*—In a web application, one instance of the bean is created for each

request.

@Component

@Scope(ConfigurableBeanFactory.SCOPE\_PROTOTYPE)

public class Notepad { ... }

@Bean

@Scope(ConfigurableBeanFactory.SCOPE\_PROTOTYPE)

public Notepad notepad() {

return new Notepad();

}

***Runtime value injection***

Property placeholders

The simplest way to resolve external values in Spring is to declare a property source and retrieve the properties via the Spring Environment.

@Configuration

@PropertySource("classpath:/com/soundsystem/app.properties")

public class ExpressiveConfig {

@Autowired

Environment env;

@Bean

public BlankDisc disc() {

return new BlankDisc(

env.getProperty("disc.title"),

env.getProperty("disc.artist"));

}

For example, suppose you’re retrieving a value representing the number of connections to maintain in a connection pool. If you receive a String value from the properties file, then you’ll need to convert it to an Integer before you can use it. But using one of the overloaded getProperty() methods handles that conversion for you:

int connectionCount = env.getProperty("db.connection.count", Integer.class, 30);

T getProperty(String key, Class<T> type, T defaultValue)

RESOLVING PROPERTY PLACEHOLDERS

When relying on component-scanning and autowiring to create and initialize your application components, there’s no configuration file or class where you can specify the placeholders. Instead, you can use the @Value annotation in much the same way as you might use the @Autowired annotation.

public BlankDisc(

@Value("${disc.title}") String title,

@Value("${disc.artist}") String artist) {

this.title = title;

this.artist = artist;

}

@Bean

public

static PropertySourcesPlaceholderConfigurer placeholderConfigurer() {

return new PropertySourcesPlaceholderConfigurer();

}

***Wiring with the Spring Expression Language***

The first thing to know is that SpEL expressions are framed with #{ ... }, much as property placeholders are framed with ${ ... }.

#{T(System).currentTimeMillis()}

Ultimately this expression evaluates to the current time in milliseconds at the moment

when the expression is evaluated. The T() operator evaluates java.lang.System as a

type so that the staticcurrentTimeMillis() method can be invoked

SpEL expressions can also refer to other beans or properties on those beans. For

example, the following expression evaluates to the value of the artist property on a

bean whose ID is sgtPeppers:

#{sgtPeppers.artist}

You can also refer to system properties via the systemProperties object:

#{systemProperties['disc.title']}

***Spring and the Java Persistence API***

*Application-managed*—Entity managers are created when an application directly

requests one from an entity manager factory. With application-managed entity

managers, the application is responsible for opening or closing entity managers

and involving the entity manager in transactions. This type of entity manager is

most appropriate for use in standalone applications that don’t run in a Java EE

container.

 *Container-managed*—Entity managers are created and managed by a Java EE

container. The application doesn’t interact with the entity manager factory at

all. Instead, entity managers are obtained directly through injection or from

JNDI. The container is responsible for configuring the entity manager factories.

This type of entity manager is most appropriate for use by a Java EE container

that wants to maintain some control over JPA configuration beyond what’s specified

in persistence.xml.

LocalEntityManagerFactoryBean produces an application-managed Entity-

ManagerFactory.

 LocalContainerEntityManagerFactoryBean produces a container-managed

EntityManagerFactory

The only real difference between application-managed and container-managed

entity manager factories, as far as Spring is concerned, is how each is configured in the

Spring application context

CONFIGURING APPLICATION-MANAGED JPA

Application-managed entity-manager factories derive most of their configuration

information from a configuration file called persistence.xml. This file must appear in

the META-INF directory in the classpath.

@Bean

public LocalEntityManagerFactoryBean entityManagerFactoryBean() {

LocalEntityManagerFactoryBean emfb

= new LocalEntityManagerFactoryBean();

emfb.setPersistenceUnitName("spitterPU");

return emfb;

}

The value given to the persistenceUnitName property refers to the persistence unit

name as it appears in persistence.xml.

Here’s a typical example of a persistence.xml file as it pertains to

the Spittr application:

<persistence xmlns="http://java.sun.com/xml/ns/persistence"

version="1.0">

<persistence-unit name="spitterPU">

<class>com.habuma.spittr.domain.Spitter</class>

<class>com.habuma.spittr.domain.Spittle</class>

<properties>

<property name="toplink.jdbc.driver"

value="org.hsqldb.jdbcDriver" />

<property name="toplink.jdbc.url" value=

"jdbc:hsqldb:hsql://localhost/spitter/spitter" />

<property name="toplink.jdbc.user"

value="sa" />

<property name="toplink.jdbc.password"

value="" />

</properties>

</persistence-unit>

</persistence>

CONFIGURING CONTAINER-MANAGED JPA

public LocalContainerEntityManagerFactoryBean entityManagerFactory(

DataSource dataSource, JpaVendorAdapter jpaVendorAdapter) {

LocalContainerEntityManagerFactoryBean emfb =

new LocalContainerEntityManagerFactoryBean();

emfb.setDataSource(dataSource);

emfb.setJpaVendorAdapter(jpaVendorAdapter);

emfb.setPackagesToScan("com.habuma.spittr.domain");

return emfb;

}

You can use the jpaVendorAdapter property to provide specifics about the particular

JPA implementation to use

In this case, you’re using Hibernate as a JPA implementation, so you configure it with

a HibernateJpaVendorAdapter:

public LocalContainerEntityManagerFactoryBean entityManagerFactory(

DataSource dataSource, JpaVendorAdapter jpaVendorAdapter) {

LocalContainerEntityManagerFactoryBean emfb =

new LocalContainerEntityManagerFactoryBean();

emfb.setDataSource(dataSource);

emfb.setJpaVendorAdapter(jpaVendorAdapter);

emfb.setPackagesToScan("com.habuma.spittr.domain");

return emfb;

}

***Writing a JPA-based repository***

@Repository

@Transactional

public class JpaSpitterRepository implements SpitterRepository {

@PersistenceUnit

private EntityManagerFactory emf;

public void addSpitter(Spitter spitter) {

emf.createEntityManager().persist(spitter);

}

public Spitter getSpitterById(long id) {

return emf.createEntityManager().find(Spitter.class, id);

}

public void saveSpitter(Spitter spitter) {

emf.createEntityManager().merge(spitter);

}

...

}

the EntityManagerFactory property. It’s annotated with @PersistenceUnit so that Spring can inject the EntityManager- Factory into the repository.

This new version of JpaSpitterRepository is now given an EntityManager directly;

there’s no need for it to create one from an EntityManagerFactory in each of its

methods.

The only gotcha with JpaSpitterRepository as it stands is that each method ends up calling createEntityManager.

The problem is that an EntityManager isn’t thread-safe and generally shouldn’t be injected into a shared singleton bean like your repository

@Repository

@Transactional

public class JpaSpitterRepository implements SpitterRepository {

@PersistenceContext

private EntityManager em;

public void addSpitter(Spitter spitter) {

em.persist(spitter);

}

public Spitter getSpitterById(long id) {

return em.find(Spitter.class, id);

}

@PersistenceContext doesn’t inject an EntityManager—at least,

not exactly. Instead of giving the repository a real EntityManager, it gives a proxy to a

real EntityManager. That real EntityManager either is one associated with the current

transaction or, if one doesn’t exist, creates a new one

It’s important to understand that @PersistenceUnit and @PersistenceContext

aren’t Spring annotations; they’re provided by the JPA specification

JpaSpitterRepository is annotated with @Repository

and @Transactional. @Transactional indicates that the persistence methods in this

repository are involved in a transactional context

Without a template to handle exception translation, you need to annotate your repository with @Repository so PersistenceExceptionTranslationPostProcessor knows that this is a bean for which

exceptions should be translated into one of Spring’s unified data-access exceptions

***Automatic JPA repositories with Spring Data***

@Configuration

@EnableJpaRepositories(basePackages="com.habuma.spittr.db")

public class JpaConfiguration {

...

}

public interface SpitterRepository extends JpaRepository<Spitter, Long> {

}

@EnableJpaRepositories

The <jpa:repositories>\@EnableJpa-Repositories element holds all the magic of Spring Data JPA. Much like the <context:component-scan> element, <jpa:repositories> is given a basepackage

to scan. But where <context:component-scan> scans a package (and its subpackages)

for classes that are annotated with @Component, <jpa:repositories> scans

its base package for any interfaces that extend Spring Data JPA’s Repository interface.

When it finds any interface extending Repository, it automatically (at application

startup time) generates an implementation of that interface.

It also inherits 18 methods for performing

common persistence operations, such as saving a Spitter, deleting a Spitter, and

finding a Spitter by its ID.

***Defining query methods***

public interface SpitterRepository

extends JpaRepository<Spitter, Long> {

Spitter findByUsername(String username);

}

When creating the repository implementation, Spring Data will examine any methods

in the repository interface, parse the method name, and attempt to understand

the method’s purpose in the context of the persisted object. In essence, Spring Data

defines a sort of miniature domain-specific language (DSL) where persistence details

are expressed in repository method signatures

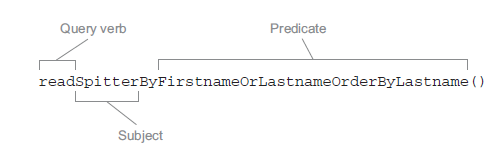
Spring Data allows for four verbs in the method name: *get*, *read*, *find*, and *count*. The

*get*, *read*, and *find* verbs are synonymous; all three result in repository methods that

query for data and return objects.

The subject of a repository method is optional. Its primary purpose is to allow you

some flexibility in how you name the method.



The predicate is the most interesting part of the method name. It specifies the

properties that will constrain the result set

some example

List<Spitter> readByFirstnameOrLastnameOrderByLastnameAsc(

String first, String last);

List<Pet> findPetsByBreedIn(List<String> breed)

 int countProductsByDiscontinuedTrue()

 List<Order> findByShippingDateBetween(Date start, Date end)

***Declaring custom queries***

In situations where the desired data can’t be adequately expressed in the method

name, you can use the @Query annotation to provide Spring Data with the query that

should be performed. For the findAllGmailSpitters() method, you might use

@Query like this:

@Query("select s from Spitter s where s.email like '%gmail.com'")

List<Spitter> findAllGmailSpitters();

***Mixing in custom functionality***

public class SpitterRepositoryImpl implements SpitterSweeper {

@PersistenceContext

private EntityManager em;

public int eliteSweep() {

String update =

"UPDATE Spitter spitter " +

"SET spitter.status = 'Elite' " +

"WHERE spitter.status = 'Newbie' " +

"AND spitter.id IN (" +

"SELECT s FROM Spitter s WHERE (" +

" SELECT COUNT(spittles) FROM s.spittles spittles) > 10000" +

")";

return em.createQuery(update).executeUpdate();

}

}

# Q [Difference between @Transactional and @TransactionAttribute](https://stackoverflow.com/questions/21453856/difference-between-transactional-and-transactionattribute)

**@TransactionAttribute** is for EJB3 beans.

**@Transactional** is for POJOs (for example Seam, Spring/Hibernate).

<https://stackoverflow.com/questions/21453856/difference-between-transactional-and-transactionattribute>

<https://blog.frankel.ch/transaction-management-ejb3-vs-spring/>

**12**

***Persisting documents with MongoDB***