## Formas de cargar el application context de spring:

AnnotationConfigApplicationContext, para carga del contexto de aplicación desde código java. Ejemplo:

ApplicationContext context = new AnnotationConfigApplicationContext(

com.springinaction.knights.config.KnightConfig.class);

AnnotationConfigWebApplicationContext, para carga del contexto de una aplicación web desde código java.

ClassPathXmlApplicationContext, carga el contexto de aplicación desde uno o varios ficheros xml colocados en el classpath. Busca dentro del classpath incluso en ficheros jar.

FileSystemXmlApplicationContext, carga el contexto de aplicación desde uno o varios ficheros xml, ubicados en el sistema de ficheros

XmlWebApplicationContext, carga el contexto desde uno o varios ficheros xml contenidos en la aplicación web.

# Opciones para configurar Spring:

Spring es responsable de crear los beans y de coordinar las relaciones entre ellos. La responsabilidad de determinar que objetos son necesarios crear y de cablearlos será del desarrollador.

* configuración explicita con xml
* configuración explicita en java
* descubrimiento implícito de beans y cableado automático.

En un proyecto es posible mezclar los tres tipos de configuraciones.

# Cableado implícito o automático de beans:

Spring resuelve el cableado automático desde dos angulos:

* Components scanning: Spring automáticamente descubre los beans que tiene que crear en el application context.
* Autowiring: Spring resuelve las dependencias entre beans.

Lo recomendable sería recurrir lo mínimo posible a la solución explicita, utilizando lo más posible el Autowiring y el component scanning.

Al añadir la anotación @Component a una clase, spring crea un bean y lo añade al application context. Podemos etiquetar o identificar un bean de dos formas:

@Component(‘NombreBean’)

public class ClaseA ….

también:

@Named (‘NombreBean’)

public class ClaseA …

Si queremos cambiar el paquete donde spring debe buscar para crear los beans podemos:

@ComponentScan (‘paquete’)

@ComponentScan(basePackages={"paquete", "otropaquete"})

Autocableado (@Autowired):

Spring cuando encuentra esta anotación o @Inject (pertenece a Java Dependency Injection), busca en el contexto de aplicación el bean que encaje. Si no encuentra ninguna clase para la creación del bean devuelve exception, a no ser que se índice @Autowired (requered=false).

si encuentra varias clases que cumplen, también devolverá excepción.

# Cableado explícito de beans:

Con librerías de terceros no es posible insertar las anotaciones @Component ni @Autowired, por lo que se hace necesario hacer uso del cableado explícito.

El cableado explíctio de beans puede plantearse con clases java o con xml.

**Cableado explicito con java:**

Con notación @Configuration:

Para crear la JavaConfig class utilizamos la notación @Configuration, con la que spring sabe dónde buscar para crear los beans.

Con notación @bean:

Con la notación bean spring crea el bean y lo incorpora en el application context.

@Bean(name="lonelyHeartsClubBand")

public CompactDisc sgtPeppers() {

return new SgtPeppers();

}

@Bean

public CDPlayer cdPlayer() {

return new CDPlayer(sgtPeppers());

}

El ID de este último bean será cdPlayer, como el nombre del método. Aparentemente CompactDist es proporcionado a través del método sgtPeppers(), pero realmene no es así, spring ya tiene el bean generado en el application context y simplemente propporcional el objeto CompactDist como si se llamase a dicho método.

En este caso:

@Bean

public CDPlayer cdPlayer() {

return new CDPlayer(sgtPeppers());

}

@Bean

public CDPlayer anotherCDPlayer() {

return new CDPlayer(sgtPeppers());

}

Ambos beans del tipo CDPlayer comparten la misma instacncia de CompactDist. Spring en fase de inicialización invocó el método sgtPeppers() y el bean generado de esta forma lo proporciona en las posibles llamadas a éste método que pueda haber.

Otra forma de hacer lo mismo sería:

@Bean

public CDPlayer cdPlayer(CompactDisc compactDisc) {

return new CDPlayer(compactDisc);

}

Aquí, cuando spring llama a cdPlayer para crear el bean, autocablea (autowires) CompactDisc en la configuración del método. El bean CompactDisc, podría haberse descubierto con el component scanner o haberse declarado en XML.

@Bean(destroyMethod="close")

public DataSource dataSource() {

BasicDataSource dataSource = new BasicDataSource();

dataSource.setUrl("jdbc:h2:tcp://dbserver/~/test");

dataSource.setDriverClassName("org.h2.Driver");

dataSource.setUsername("sa");

dataSource.setPassword("password");

dataSource.setInitialSize(20);

dataSource.setMaxActive(30);

return dataSource;

}

Referencia a xml desde la configuración Java (JavaConfig):

@Configuration

@Import(CDPlayerConfig.class)

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

public class SoundSystemConfig {

}

# Entornos y perfiles (profiles)

Estableciendo un determinado profile podemos hacer que spring cree o no determinados beans.

@Configuration

public class DataSourceConfig {

@Bean(destroyMethod="shutdown")

@Profile("dev")

public DataSource embeddedDataSource() {

return new EmbeddedDatabaseBuilder()

.setType(EmbeddedDatabaseType.H2)

.addScript("classpath:schema.sql")

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

.build();

}

@Bean

@Profile("prod")

public DataSource jndiDataSource() {

JndiObjectFactoryBean jndiObjectFactoryBean =

new JndiObjectFactoryBean();

jndiObjectFactoryBean.setJndiName("jdbc/myDS");

jndiObjectFactoryBean.setResourceRef(true);

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

return (DataSource) jndiObjectFactoryBean.getObject();

}

}

Si está establecido el profile “prod”, entonces solo el bean jniDataSource se creará.

La anotación @Profile puede ir a nivel de clase, con lo que afecataría a todas las anotaciones @Bean que esta clase pueda incluir.

Para establecer un profile disponemos de las siguientes opciones:

 As initialization parameters on DispatcherServlet

 As context parameters of a web application

 As JNDI entries

 As environment variables

 As JVM system properties

 Using the @ActiveProfiles annotation on an integration test class

El entorno lo podemos establecer con spring.profile.default y con spring.profile.active. Si se utilizan ambos entonces srping.profile.active tiene prioridad.

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

<web-app version="2.5"

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

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/javaee

http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd">

Listing 3.3 Setting default profiles in a web application’s web.xml file

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/spring/root-context.xml</param-value>

</context-param>

<context-param>

<param-name>spring.profiles.default</param-name>

<param-value>dev</param-value>

</context-param>

<listener>

<listener-class>

org.springframework.web.context.ContextLoaderListener

</listener-class>

</listener>

<servlet>

<servlet-name>appServlet</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<init-param>

<param-name>spring.profiles.default</param-name>

<param-value>dev</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>appServlet</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

</web-app>

# Resolución de ambigüedades en el autocableado

@Component

public class Cake implements Dessert { ... }

@Component

public class Cookies implements Dessert { ... }

@Component

public class IceCream implements Dessert { ... }

Con esto obtenemos excepcion NoUniqueBeanDefinitionException

@Autowired

public void setDessert(Dessert dessert) {

this.dessert = dessert;

}

Para evitar las ambigüedades tenemos dos opciones:

1. Con la notación @Primary indicamos a spring que ante una ambigüedad escoja la creación del bean que contenga esta notación:

@Component

@Primary

public class IceCream implements Dessert { ... }

1. Añadiendo identificadores:

@Autowired

@Qualifier("iceCream")

public void setDessert(Dessert dessert) {

this.dessert = dessert;

}

El bean creado para la clase IceCream tendrá como nombre asignado iceCream.

Otra opción es utilizar la notación @Qualifier cuando se declaran los bean dependientes, esta opción tiene ventajas en la renderización (por ejemplo si cambiamos de nombre de la clase IceCram, tendremos que tener en cuenta que @Qualifier("iceCream") ya no es válido y tendremos que indicar el nuevo nombre.

@Component

@Qualifier("cold")

public class IceCream implements Dessert { ... }

@Autowired

@Qualifier("cold")

public void setDessert(Dessert dessert) {

this.dessert = dessert;

}

también:

@Bean

@Qualifier("cold")

public Dessert iceCream() {

return new IceCream();

}

Java no permite la posibilidad de indicar varios @Qualfier pero tenemos la opción de crear nuestro propio Qualifier:

@Target({ElementType.CONSTRUCTOR, ElementType.FIELD,

ElementType.METHOD, ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Qualifier

public @interface Cold { }

@Target({ElementType.CONSTRUCTOR, ElementType.FIELD,

ElementType.METHOD, ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Qualifier

public @interface Creamy { }

Con lo que ahora es posible sacar a spring de la ambigueadad:

@Component

@Cold

@Creamy

public class IceCream implements Dessert { ... }

# Alcance de los beans

Los bean de spring normalmente son singletons, pero podemos encontrarnos en la necesidad de utilizar varias instancias de una misma clase.

Estos son los scopes de los beans:

 *Singleton*—One instance of the bean is created for the entire application. ES el scope por defecto.

 *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.

Tendremos que concretar el scope con la notación @Scope y la podremos colocar después de la notación @Component o @Bean:

@Component

@Scope(ConfigurableBeanFactory.SCOPE\_PROTOTYPE)

public class Notepad { ... }

@Bean

@Scope(ConfigurableBeanFactory.SCOPE\_PROTOTYPE)

public Notepad notepad() {

return new Notepad();

}

Centrándonos en el caso Sesion-in, que podemos aplicar al típico carrito de la compra en una página web.

@Component

@Scope(

value=WebApplicationContext.SCOPE\_SESSION,

proxyMode=ScopedProxyMode.INTERFACES)

public ShoppingCart cart() { ... }

@Component

public class StoreService {

@Autowired

public void setShoppingCart(ShoppingCart shoppingCart) {

this.shoppingCart = shoppingCart;

}

...

}

# Inyección de valores en tiempo de ejecución:

En ocasiones no será posible la asignación de valores con hard-code, como en este caso:

@Bean

public CompactDisc sgtPeppers() {

return new BlankDisc(

"Sgt. Pepper's Lonely Hearts Club Band",

"The Beatles");

}

y se necesitará hacerlo en tiempo de ejecución, para ello en spring disponemos de :

 Property placeholders

 The Spring Expression Language (SpEL)

# Inyección de valores externos:

package com.soundsystem;

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

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.PropertySource;

import org.springframework.core.env.Environment;

@Configuration

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

public class ExpressiveConfig {

@Autowired

Environment env;

@Bean**t**

public BlankDisc disc() {

return new BlankDisc(

env.getProperty("disc.title"),

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

}

}

el fichero app.properties, podría contener:

disc.title=Sgt. Peppers Lonely Hearts Club Band

disc.artist=The Beatles

El método env.getProperty, está sobreescrito con estas alternativas:

 String getProperty(String key)

 String getProperty(String key, String defaultValue)

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

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

Podemos usar String getProperty(String key, String defaultValue) si queremos dar valores predeterminados, en el caso de que la propiedad no esté includia en el fichero app.propertiers.

La tercera y cuarta posibilidad es para tratar otros tipos diferentes a String, por ejemplo para el tipo Integer:

int connectionCount =

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

Si se utiliza String getProperty(String key), y no se incluye en el fichero app.properties, el valor asignado será null. Si se utiliza env.getRequiredProperty("disc.title"), entonces se lanza excepción IllegalStateException.

también:

public BlankDisc(

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

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

this.title = title;

this.artist = artist;

}

Para resolver ${} y @Value, es necesario registrar el método estático siguiente:

@Bean

public

static PropertySourcesPlaceholderConfigurer placeholderConfigurer() {

return new PropertySourcesPlaceholderConfigurer();

}

# Programación orientada a aspectos.

AOP consiste en definir una funcionalidad en un lugar y declarativamente definir cuándo y dónde se aplica sin necesitar realizar modificaciones sobre la clase a la que se aplica.

El AOP intenta modularizar funcionalidades transversales a través de aspectos. AOP intenta eliminar la duplicidad de código por ejemplo a la hora de validad permisos.

Terminología AOP.

**Adviced** (consejo).

Un advice es una acción que hay que ejecutar en un punto/s para conseguir la implementación del aspecto. Adviced establece el qué y el cuándo de un aspecto.

En AOP de spring se definen los siguientes Advices:

Before: La funcionalidad advice tiene lugar antes de que el método advice sea invocado (@Before).

After: La funcionaliad advice tiene lugar después de que el método advice sea invocado (@After).

After-returning: La funcionalidad advice tiene lugar después de que el método advice haya finalizado satisfactoriamente (@AfterReturning)..

After-throwing: La funcionalidad advice tiene lugar después de que el método advice haya arrojado una excepción (@AfterThrowing).

Around: Antes y después de que el método advice sea invocado (@Around).

**Join point:**

Es un punto en la ejecución de un programa donde podemos colocar un aspecto. Estos puntos pueden ser método que se llame o una excepción lanzada o un campo que se está modificando.

**Pointcuts**

Si los advices establecen el qué y el cuándo, los pointcuts establecen el dónde. Son el conjunto de puntos del código donde hay que ejecutar un advice.

Los pointcuts se especifican por ejemplo usando clases explicitas, nombre de métodos o a través de expresiones regulares que determinan determinadas clases o métodos. Alugnos frameworks AOP pueden asignar pointcuts en tiempo de ejecución.

**Aspects**

Es la fusión del advice y del pointcuts.

El qué más el cuándo más el dónde.

Con el término weaving indicamos el proceso en el que se aplica el aspecto a un objeto creando un proxy. Los aspect se construyen en el objeto en los llamdos join point.

Esta construcción puede tener lugar en diferentes points dentro del ciclo de vida de un objeto:

* Tiempo de compilación.

El aspecto se construye cuando se compila la clase. El compilador aspectJ hace esto mismo.

* Carga de la clase.

Requieren de un claseloader especial como el de AspecJ.

* Tiempo de ejecución. El aspecto se contruye en tiempo de ejecución, así es como trabaja el AOP de spring.

El AOP de spring contempla muchos conceptos de AspectJ. El sporte AOP de spring tiene 4 estilos:

* El AOP clásico de spring.
* Aspectos pure-POJO.
* La anotación @AspectJ.
* Inyección de aspectos AspectJ.

Spring solo soporta los join points en métodos. Otros frameworks AOP como AspectJ o JBoss soportan campos y constructores como join points.

Spring solo soporta un subconjunto de los pointcuts de Aspectj. Spring para no reinventar la rueda, define los aspectos con el lenguaje de expresiones de pointcuts de AspectJ.

# Spring MVC

Pasos de una solicitud (request) en el modelo:

1. Spring MVC utiliza el patrón “front controller servlet” para delegar la responsabilidad de tratamiento de la petición en otros procesos (en spring MVC son los controladores). En spring MVC este front controller se llama DispatcherServlet.
2. Para determinar el controlador, el dispatcherServlet consultará los handler mappers utilizando como parámetro la URL.
3. Una vez es conocido el controlador, el dispatcherServlet envía al controlador la petición.
4. El controlador devuelve el model (los datos de respuesta) al dispatcherServlet. También devuelve el view name, que no se corresponde directamente con un jsp concreto. Es un nombre de vista lógico.
5. El dispatcherServlet, consulta con un mapeo de resolución de vistas (view resolver map), para determinar la implementación de la vista, que puede o no ser un jsp. Normalmente este mapeo lo mantiene la clase que implementa ViewResolver, que deberá estar presente en el applicatión context (con @Bean).
6. Con la vista ya conocida (normalmente un jsp), el dispatcherServlet entrega los datos del modelo (model) a la view.
7. La view renderiza los datos del modelo (respuesta) y se muestran al usuario.

package spittr.config;

import org.springframework.web.servlet.support.

AbstractAnnotationConfigDispatcherServletInitializer;

public class SpittrWebAppInitializer

extends AbstractAnnotationConfigDispatcherServletInitializer {

@Override

protected String[] getServletMappings() {

return new String[] { "/" };

}

@Override

protected Class<?>[] getRootConfigClasses() {

return new Class<?>[] { RootConfig.class };

}

@Override

protected Class<?>[] getServletConfigClasses() {

return new Class<?>[] { WebConfig.class };

}

}

DistpacherServlet es la pieza central de Spring MVC, es responsable del enrutamiento de las solicitudes. En configuraciones xml, el fichero web.xml es responsable de configurar el DistpacherServlet pero en configuraciones por anotaciones, cualquier clase que extiende AbstractAnnotationConfigDispatcherServletInitializer, spring la utilizará para configurar el dispatcherServlet y el spring application Context.

El método getServletMappings, asignará una o más rutas para el mapeo del dispatcherServlet. Con “\” indicamos que será el servlet predeterminado de la aplicación y que manejará todas las solicitudes de la aplicación.

Es posible configurar varios DistpacherServlet, añadiendo otra clase que extienda AbstractAnnotationConfigDispatcherServletInitializer:

public class WebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

@Override

protected Class<?>[] getRootConfigClasses() {

// TODO Auto-generated method stub

return new Class<?>[] { RootConfig.class };

}

@Override

protected Class<?>[] getServletConfigClasses() {

// TODO Auto-generated method stub

return new Class<?>[] { WebConfig.class };

}

@Override

protected String[] getServletMappings() {

// TODO Auto-generated method stub

return new String[] { "/config1/\*" };

}

}

public class WebAppInitializer2 extends AbstractAnnotationConfigDispatcherServletInitializer {

@Override

protected Class<?>[] getRootConfigClasses() {

// TODO Auto-generated method stub

return new Class<?>[] { RootConfig.class };

}

@Override

protected Class<?>[] getServletConfigClasses() {

// TODO Auto-generated method stub

return new Class<?>[] { WebConfig2.class };

}

@Override

protected String[] getServletMappings() {

// TODO Auto-generated method stub

return new String[] { "/config2/\*" };

}

@Override

protected String getServletName() {

// TODO Auto-generated method stub

return "config2";

}

}

Con

<?>[] getServletConfigClasses() {

return new Class<?>[] { WebConfig.class };

estamos indicando pidiendo al DistpacherServlet que carge el application context con los beans definidos en WebConfig.class.

El fichero WebConfig.java, contendrá:

@Configuration

@EnableWebMvc

@ComponentScan("spitter.web")

public class WebConfig extends WebMvcConfigurerAdapter {

@Bean

public ViewResolver viewResolver() {

InternalResourceViewResolver resolver =

new InternalResourceViewResolver();

resolver.setPrefix("/WEB-INF/views/");

resolver.setSuffix(".jsp");

resolver.setExposeContextBeansAsAttributes(true);

return resolver;

}

@Override

public void configureDefaultServletHandling(

DefaultServletHandlerConfigurer configurer) {

configurer.enable();

}

}

Con @ComponentScan("spitter.web"), indicamos el paquete donde deben buscarse los componentes. Estos componentes se identificarán con la anotación @Controller en el caso de tratarse de un controlador.

En el caso de HomeController.java:

@Controller

@RequestMapping("/")

**public** **class** HomeController {

@RequestMapping(method = ***GET***)

**public** String home(Model model) {

**return** "home";

}

}

Con @Controller pedimos a DistpacherServlet que carge este bean en el application context. Podríamos utilizar la anotación @Component y funcionaría, pero resulta más explicativo la primera.

Aquí, el dispatcherServlet determina a través de viewResolver() que la view física le corresponde a “home”, que en este caso sería: /WEB-INF/views/home.jsp.

AbstractAnnotationConfigDispatcherServletInitializer crea tanto DispatcherServlet como ContextLoaderListener. Las clases @Configuration devueltas por getServletConfigClasses(), definirán los beans para el contexto de aplicación del DispatcherServlet, es decir, beans de componentes web como controladores, view resolver y handler mapper. Por otro lado, getRootConfigClasses() devolverá los beans para configurar el contexto de aplicación creado por el ContextLoaderListener. Eston beans son componentes para el nivel de datos y para el nivel medio de la aplicación.

Con

@Override

public void configureDefaultServletHandling(

DefaultServletHandlerConfigurer configurer) {

configurer.enable();

}

, estamos pidiendo al DispatpacherServlet, que no procese las solicitudes de recursos estáticos y que las envíe hacia el servlet predeterminado del contenedor de Servlet.