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 ángulos:

* 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.

Spring web flow

Spring web flow está basado en Srping MVC, por lo que las solicitudes de flujo pasan en primer lugar por el DistpacherServlet.

Spring web flow solo se puede configurar mediante xml, no con anotaciones java.

**Pasos en Spring web flow:**

* Carga y ejecución de los flujos, el ejecutor.

Se crea un ejecutor de flujos con:

<flow:flow-executor id=”flowExecutor”>

* Cargar y localizar los flujos. Registro de flujos.

Tipo 1. Con path base y patrón de fichero:

Ejemplo de registro de flujo:

<flow:flow-registry id = “flowRegistry” base-path=”WEB-INF/flows”>

<flow:flow-location-patter value = “/\*\*/\*-flow.xml”/>

</flow:flow-registry>

identificación de las partes:

/WEB-INF/flows/order/order-flow.xml

/WEB-INF/flows/ -> ruta base del registro de flujo

order -> flow id

/order-flow.xml -> xml con la definición del flujo

Tipo 2. Con identificación del fichero de flujo de forma explícita.

Ejemplo:

<flow:flow-registry id =”flowRegistry”>

<flow:flow-locattion path=”/WEB-INF/flows/springpizza.xml”/>

</flow:flow-registry>

En este caso el identificador del flujo se obtiene del nombre base del fichero identificador de flujo, es decir, el id sería “srpingpizza”

Tipo 3. Con identificación explicita del fichero de flujo y con identificador explícito del flujo.

Ejemplo:

<flow:flow-registry id =”flowRegistry”>

<flow:flow-locattion id= “pizza” path=”/WEB-INF/flows/springpizza.xml”/>

</flow:flow-registry>

* El conductor de la ruta del flujo.

El DistpacherServlet, se apoya en el FlowHandlerMapping para que determine las clases que recibirán las solicitudes del flujo.

El FlowHandlerMapping se configura en el contexto de aplicación de la siguiente manera:

<bean class = “org.springframework.webflow.mvc.servlet.FlowHandlerMapping”>

property name=”flowRegistry” ref=”flowRegistry” />

</bean>

* Dar respuestas a las solicitudes del flujo.

El FlowHandlerAdapter, recibe las solicitues del flujo, las procesa y les da respuesta. Es como un controlador en el más puro mvc.

Se incorpora como bean en el contexto de spring:

<bean class = “org.springframework.webflow.mvc.servlet.FlowHandleAdapter”>

<property name =”flowExecutor” ref =”flowExecutor” />

</bean>

**Componentes de un flujo:**

Estados, transiciones y datos del flujo.

Estados:

Los estados son puntos del flujo donde se toma una decisión o se muestra una página al usuario.

Tipos de estado:

* Estado de vista:

En este estado se muestra información al usuario para que tome un papel activo en el flujo.

Definición de un estado de vista:

<view-state id=”welcome”/>

Arriba se identifica el estado de vista y también se determina la vista lógica que sería “welcome”.

<view estate id=”welcome” view =”greeting” />

Arriba se identifica el estado de la vista y también se termina explícitamente el nombre lógico de la vista, en este caso “greeting”.

<view-state id=”takePayment” model =”flowScope.paymentDetails” />

Arriba el formulario asociado a la vista “takaPayment” está asociado al objeto flowScope.paymentDetails.

* Estado de acción:

Es este estado es cuando se realiza el trabajo, en los estados de acción se suele invocar algún método (de un bean controlado por spring), que realiza la acción y a continuación, dependiendo del resultado se realiza un transición u otra.

<action-state id=”saveOrder”>

<evaluate expresión=”pizzaFlowActions.saveOrder(order)” />

<transaction to=”thankYou”” />

</action-state>

* Estado de decision:

En este estado se evalua una expression booleana para determinar una transición u otra.

<decision-state id="checkDeliveryArea">

<if test="pizzaFlowActions.checkDeliveryArea(customer.zipCode)"

then="addCustomer"

else="deliveryWarning" />

</decision-state>

* Estado de subflujo

Permite ejecutar un flujo dentro de otro.

Declaración de un subflujo:

<subflow-state id="order" subflow="pizza/order">

<input name="order" value="order"/>

<transition on="orderCreated" to="payment" />

</subflow-state>

El elemento input se utiliza para proporcionar el objeto del pedido como entrada al subflujo.

* Estados finales.

Cuando el flujo alcanza el estado <end-state>, finaliza. Lo que sucede a continuación depende de:

Si el flujo finaliza en un subflujo, el Id del <end-state> se utilizará como evento para activar la transición desde <subflow-state>.

Si <end-state> tiene un atributo view configurado, entonces se presentará la vista.

Si el flujo final no es un subflujo y no se especifica vista, entonces el flujo termina. El navegador accederá a la URL base del flujo y se inicia una nueva instancia de este.

Transiciones:

Las transiciones conectan los estados. Un estado puede tener varias transiciones a excepción de los estados finales.

<transition on="phoneEntered" to="lookupCustomer"/>

Transicion a lookupCustomer cuando se produce el evento phoneEntered. En un estado vista, el evento suele ser una acción del usuario. En un estado de acción, el evento es el resultado de evaluar una expresión.

<transition on-exception= "com.springinaction.pizza.service.CustomerNotFoundException"

to="registrationForm" />

Si varios estados tiene transiciones comunes, podemos definir una transacción global:

<global-transitions>

<transition on="cancel" to="endState" />

</global-transitions>

Flujo de datos.

En el paso de un estado a otro, se van recogiendo datos que pueden ser utilizado en cualquier momento o al finalizar el flujo.

Declaración de variables.

Con la etiqueta var podemos declarar variables visibles en cualquier estado del flujo.

<var name="customer" class="com.springinaction.pizza.domain.Customer"/>

En un estado de acción o entrada de vista podemos declarar una variable con ámbito de vista (view-scope):

<evaluate result="viewScope.toppingsList"

expression="T(com.springinaction.pizza.domain.Topping).asList()" />

también:

<set name="flowScope.pizza"

value="new com.springinaction.pizza.domain.Pizza()" />

Tipos de scope:

Conversation: Created when a top-level flow starts, and destroyed when the top-level flow ends. Shared by a top-level flow and all of its subflows.

Flow: Created when a flow starts, and destroyed when the flow ends. Only visible in the

flow it was created by.

Request: Created when a request is made into a flow, and destroyed when the flow returns.

Flash: Created when a flow starts, and destroyed when the flow ends. It’s also cleared

out after a view state renders.

View: Created when a view state is entered, and destroyed when the state exits. Visible

only in the view state.

Spring WebSockets

[https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html#websocket](https://docs.spring.io/spring/docs/5.0.3.RELEASE/spring-framework-reference/web.html%23websocket)

A partir de HTML5, los navegadores son capaces de soportar websockets.

El protocolo webSockets definido en la RFC 6455, establece un estándar para la comunicación full-duplex entre el cliente y el servidor web.

El framework spring, da soporte a este protocolo con una API compatible con “Java WebSocket API” que implemente a su vez el estándar (JSR 356).

WebSocket clients and servers can negotiate the use of a higher-level, messaging protocol (e.g. STOMP), via the "Sec-WebSocket-Protocol" header on the HTTP handshake request, or in the absence of that they need to come up with their own conventions.

Los proxys pueden interceptar las conexiones WebSockets, para ello se han creado protocolos como SockJs, que se basan en HTTP para emular WebSocket.

Establecimiento del websockets.

Lado servidor.

El proyecto que maneje websockets en el servidor, tendrá que importar las dependencias spring-boot-starter-web y spring-boot-starter-webSocket.

Deberemos crear una clase que extienda a AbstractWebSocketMessageBrokerConfigurer y con las anotaciones @Configuration y @EnableWebSocketMessageBroker.

@Configuration

@EnableWebSocketMessageBroker

public class WebSocketConfiguration extends AbstractWebSocketMessageBrokerConfigurer{

@Override

public void registerStompEndpoints(StompEndpointRegistry registry) {

registry.addEndpoint("/socket")

.setAllowedOrigins("\*")

.withSockJS();

}

@Override

public void configureMessageBroker(MessageBrokerRegistry registry) {

registry.setApplicationDestinationPrefixes("/app")

.enableSimpleBroker("/chat");

}

}

En el primer método con addEndpoint(“/socket”), definimos que los clientes que se quieran conectar tendrán que hacerlo mediante una URL <http://xxx/socket>

Con setAllowdOrigins(“\*”), se permitirá separar cliente y servidor como dos proyectos diferentes. La URL del cliente podrá ser distinta de la URL del servidor o backend.

Lado cliente

El cliente creará una instancia SockJS indicando la URL definida para el establecimiento del canal.

import SockJS from 'sockjs-client';

. . .

**let ws = new SockJS(“**[**http://xxx/socket**](http://xxx/socket)**”);**

Mensajería hacia el servidor:

Definición del canal en servidor

En el segundo método, setApplicationDestinationPrefixes (“/app”), definimos un prefijo que utilizarán los clientes para enviar a través del socket.

Envio de mensajes del cliente

Ejemplo de envío de mensajes con angular:

import SockJS from 'sockjs-client';

…

let ws = new SockJS(“<http://xxx/socket>”);

this.stompClient = Stomp.over(ws);

…

**this.stompClient.send("/app/send/message" , {}, message);**

Recepción de los mensajes en el servidor

Desde el controlador del servidor, utilizando la anotación @MessageMapping de spring recogemos los mensajes enviados desde el cliente.

@Controller

public class WebSocketController {

private final SimpMessagingTemplate template;

@Autowired

WebSocketController(SimpMessagingTemplate template){

this.template = template;

}

**@MessageMapping("/send/message")**

**public void onReceivedMesage(String message)**{

this.template.convertAndSend("/chat", new SimpleDateFormat("HH:mm:ss").format(new Date())+"- "+message);

}

}

Subscripción y envío de la mensajería hacia el cliente

Lado servidor

El servidor envía mensajes sobre el canal definido en enableSimpleBlorker, en este claso (“\chat”).

@Controller

public class WebSocketController {

private final SimpMessagingTemplate template;

@Autowired

WebSocketController(SimpMessagingTemplate template){

this.template = template;

}

@MessageMapping("/send/message")

public void onReceivedMesage(String message){

**this.template.convertAndSend("/chat",** new SimpleDateFormat("HH:mm:ss").format(new Date())+"- "+message);

}

}

En el ejemplo de arriba, los mensajes que se reciben se enviarán a los clientes que se hayan suscrito a “/chat”.

Lado cliente

Los clientes se suscriben a “/chat” de la siguiente manera:

this.stompClient.connect({}, function(frame) {

that.stompClient.subscribe("/chat", (message) => {

if(message.body) {

$(".chat").append("<div class='message'>"+message.body+"</div>");

console.log("mensaje : " + message.body);

}

});

});

Al mismo tiempo que nos subscribimos a “/chat”, definimos la actividad con los mensajes recibidos sobre este canal.

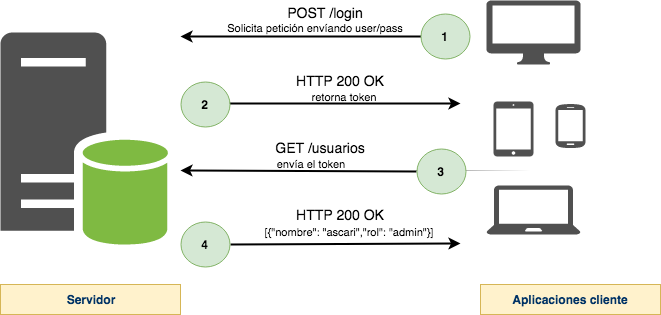
Spring security

Seguridad con token en spring boot (JWT)

Los servicios RESTful, encajan perfectamente con la seguridad basada en tokens.

JWT significa **J**son **W**eb **T**oken.

Al dar el cliente las credenciales de usuario y password, el servidor devolverá el token si las credenciales son correctas. El cliente para acceder a zonas restringidas de la web, necesitará incluir en la cabecera de la petición el token. A su vez, cada petición será comprobada por el servidor para retornar con los recursos solicitados por el cliente:



El cliente envía sus credenciales (usuario y password) al servidor.

Si las credenciales son válidas, el servidor devuelve al cliente un token de acceso.

El cliente solicita un recurso protegido. En la petición, se envía el token de acceso.

El servidor valida el token y en caso de ser válido, devuelve el recurso solicitado.

JWT consta de 3 partes separadas por un punto ( . )

**Header**

**Payload**

**Signature**

Cada una de estas partes se codifica en base64 de tal forma que el token generado tiene una apariencia como esta,

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9

.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9

.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ

**Header**

El header consta de dos partes, el tipo de token y el algoritmo de hash.

{

"alg": "HS256",

"typ": "JWT"

}

Si al JSON anterior lo codificamos en base64 tendremos nuestra primer parte del JWT.

**Payload**

El payload contiene datos como: iss (issuer), exp (expiration time) y sub (subject)

iss es quien emitió el token

exp contiene la fecha de expiración del token

sub indica el usuario del token

Además podemos indicar otros campos como el nombre, roles, etc.

{

"sub": "1234567890",

"name": "John Doe",

"admin": true,

"exp": "1425390142"

}

Al codificar el json anterior en base 64, obtenemos la segunda parte de nuestro JWT. Para más información recomiendo leer el siguiente enlace

**Signature**

To create the signature part you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.

For example if you want to use the HMAC SHA256 algorithm, the signature will be created in the following way:

HMACSHA256(

base64UrlEncode(header) + "." +

base64UrlEncode(payload),

secret)

The signature is used to verify that the sender of the JWT is who it says it is and to ensure that the message wasn't changed along the way.

