

Microservicios con Spring

Victor Herrero Cazurro

Contenidos

Spring Web	1
Ambitos	2
Recursos JNDI del Servidor	2
Filtros	3
Spring MVC	4
Introduccion	4
Arquitectura	5
DispatcherServlet	5
ContextLoaderListener	6
Namespace MVC	7
ResourceHandler (Acceso a recursos directamente)	8
Default Servlet Handler	8
ViewController (Asignar URL a View)	9
HandlerMapping	9
BeanNameUrlHandlerMapping	10
SimpleUrlHandlerMapping	10
ControllerClassNameHandlerMapping	11
DefaultAnnotationHandlerMapping	11
RequestMappingHandlerMapping	11
Controller	12
@Controller	12
Activación de @Controller	13
@PathVariable	13
@RequestParam	14
@RequestBody	14
@ResponseBody	15
@ModelAttribute	15
@SessionAttributes	16
@InitBinder	16
@ExceptionHandler	17
@ControllerAdvice	17
ViewResolver	17
InternalResourceViewResolver	18
XmlViewResolver	18
ResourceBundleViewResolver	19
View	19
AbstractExcelView	20
AbstractPdfView	21

JasperReportsPdfView	21
MappingJackson2JsonView	22
Formularios	23
Etiquetas	24
Paths Absolutos	26
Inicialización	27
Validaciones	27
Mensajes personalizados	28
Anotaciones JSR-303	28
Validaciones Custom	29
Internacionalización - i18n	29
Interceptor	30
LocaleChangeInterceptor	32
ThemeChangeInterceptor	33
Thymeleaf	34
HttpMessageConverters	35
Pila por defecto de HttpMessageConverters	36
Personalización de la Pila de HttpMessageConverters	37
Microservicios	37
Ventajas	37
Desventajas	38
Ejemplo: Aplicación de puntos por viajes	38
Ejemplo: Orden de compra	38
Ejemplo: Agencia de viajes	39
Rest	39
Personalizar el Mapping de la entidad	40
Estado de la petición	40
Localización del recurso	40
Cliente se servicios con RestTemplate	41
Spring WebService	43
Servicio Standalone	43
Cliente Servicio	44
Mapping de entidades	44
Despliegue en contenedor web	45
Spring Security	45
Arquitectura	45
Dependencias con Maven	46
Filtro de seguridad	46
Contexto de Seguridad	47
AuthenticationManagerBuilder	48
Protección de recursos	48

Login	49
Logout	49
CSRF	49
UserDetailsService	50
Encriptación	50
Remember Me	52
Seguridad en la capa transporte - HTTPS	52
Sesiones concurrentes	53
SessionFixation	54
Librería de etiquetas	54
Expresiones SpEL	55
Seguridad de métodos	55
Spring Test	57
Mocks	57
MVC Mocks	58
Spring Boot	60
Introducción	60
Instalación de Spring Boot CLI	61
Creación e implementación de una aplicación	62
Uso de plantillas	66
Thymeleaf	66
JSP	67
Recursos estáticos	67
Webjars	67
Recolección de métricas	68
Endpoint Custom	69
Uso de Java con start.spring.io	70
Starters	75
Soporte a propiedades	75
Configuración del Servidor	77
Configuración del Logger	77
Configuración del Datasource	77
Custom Properties	79
Profiles	79
JPA	80
Errores	82
Seguridad de las aplicaciones	82
Soporte Mensajería JMS	83
Consumidores	84
Productores	85
Soporte Mensajería AMQP	85

Receiver	86
Producer	87
Spring Cloud	87
Arquitectura de Microservicios	87
Orquestacion vs Coreografia	89
Servidor de Configuracion	89
Seguridad	91
Clientes del Servidor de Configuracion	91
Actualizar en caliente las configuraciones	93
Servidor de Registro y Descubrimiento	94
Registrar Microservicio	95
Localizacion de Microservicio registrado en Eureka con Ribbon	96
Uso de Ribbon sin Eureka	97
Simplificación de Clientes de Microservicios con Feign	97
Acceso a un servicio seguro	98
Uso de Eureka	99
Servidor de Enrutado	99
Seguridad	101
Circuit Breaker	102
Monitorizacion: Hystrix Dashboard	103
Monitorizacion: Turbine	103
Configuracion Distribuida en Bus de Mensajeria	104
Servidor	104
Cliente	105
OAuth2	106
Caracteristicas	106
Que se quiere conseguir	106
Actores	106
Caracteristicas	106
Grants	107
Implementacion Servidor OAuth con Spring	110
Implementacion Aplicacion Cliente OAuth con Spring	115
Implementacion Servidor de Recursos OAuth con Spring	116

Spring Web

Framework que permite incluir el contexto de Spring en una aplicación web.

Se basa en la definición de un **Listener** de contexto web, que cree el contexto de Spring.

Para configuraciones tradicionales con **web.xml**, se define el **Listener** y la ubicación de todos los ficheros que formen el contexto de Spring.

```
<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    classpath:xfiles-config.xml,
    /WEB-INF/security.xml
  </param-value>
</context-param>
```

Para configuraciones basadas en JavaConfig, el Api proporciona una interface **WebApplicationInitializer**, que permite la sustitución del **web.xml**

```
public class AppInitializer implements WebApplicationInitializer {
    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {
        WebApplicationContext context = getContext();
        servletContext.addListener(new ContextLoaderListener(context));
    }

    private AnnotationConfigWebApplicationContext getContext() {
        AnnotationConfigWebApplicationContext context = new
        AnnotationConfigWebApplicationContext();
        context.setConfigLocation("com.cursospring");
        return context;
    }
}
```

En este caso el contexto de Spring también se basa en JavaConfig.

Una vez definido el contexto de Spring y creado en la creación del contexto Web, cualquier componente Web JEE, podrá acceder a dicho contexto a través del contexto Web, de la siguiente manera

```
ApplicationContext context = WebApplicationContextUtils.getWebApplicationContext  
(getServletContext());
```

Ambitos

A parte de los ambitos conocidos **singleton** y **prototype**, se introducen otros 2 scope

- **request** → Los Bean se crearán por cada nueva petición a la aplicación que lo precise.
- **session** → Los Bean se crearán por cada nueva sesión de usuario que lo precise.

Para asociarlos a un Bean se puede emplear la anotación **@Scope**

Recursos JNDI del Servidor

Es habitual que el Servidor de Aplicaciones donde se despliega la aplicación web, provea a esta de recursos empleando el api JNDI, desde Spring se puede incluir dichos recursos en el contexto de spring con la siguiente configuracion xml

```
<beans:bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">  
    <beans:property name="jndiName" value="java:comp/env/jdbc/MyLocalDB"/>  
</beans:bean>
```

O de forma mas sencilla empleando el espacio de nombres jee

```
<jee:jndi-lookup expected-type="javax.sql.DataSource" id="dataSource" jndi-name=  
"jdbc/MyLocalDB"/>
```

o JavaConfig

```
@Bean  
public DataSource dataSource(@Value("${db.jndi}" String jndiName) {  
    JndiDataSourceLookup lookup = new JndiDataSourceLookup();  
    lookup.setResourceRef(true);  
    return lookup.getDataSource(jndiName);  
}
```

NOTE La clase especializada en DataSource, **JndiDataSourceLookup**, se obtiene con **spring-jdbc**

NOTE Donde db.jndi, valdra algo como **java:comp/env/jdbc/MyDB**

Para publicar el recurso JNDI se ha de consultar la documentación del servidor en concreto, dado que cada uno lo hace de una forma distinta.

Por ejemplo para un Tomcat 8, se haría definiendo en `<TOMCAT_HOME>/conf/context.xml`

```
<Context>
  <Resource name="jdbc/MyDB" auth="Container" type="javax.sql.DataSource"
    maxTotal="100" maxIdle="30" maxWaitMillis="10000"
    username="admin" password="admin" driverClassName=
"org.apache.derby.jdbc.ClientDriver"
    url="jdbc:derby://localhost:1527/jndi"/>
</Context>
```

Mapeando dicho recurso en el fichero `/WEB-INF/web.xml`

```
<resource-ref>
  <description>DB Connection</description>
  <res-ref-name>jdbc/MyDB</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

En configuraciones JavaConfig, se puede añadir al **@Bean** la anotación **@Resource** del api de servlets

```
@Configuration
public class Configuracion {
    @Bean
    @Resource(name="jdbc/MyDB")
    public DataSource dataSourceLookup() {
        final JndiDataSourceLookup dsLookup = new JndiDataSourceLookup();
        dsLookup.setResourceRef(true);
        DataSource dataSource = dsLookup.getDataSource("java:comp/env/jdbc/MyDB");
        return dataSource;
    }
}
```

Filtros

Como se ha comentado Spring Web, pretende extender el contenedor web estandar con los Bean del contexto de Spring, se ha visto como a traves del **ContextLoaderListener**, se crea el contexto y se permite el acceso a dicho Contexto desde componente dentro del contenedor web.

Cuando se trabaja con Spring MVC, se realiza algo parecido, ya que se asocia un contexto de Spring a un Servlet.

Tambien es posible extender los Filtros Web, Spring proporciona una implementación de Filtro Web **DelegatingFilterProxy** que permite delegar las intercepciones que realice dicho Filtro Web sobre Beans de Spring que implementen la interface **Filter**, en concreto delega la petición, sobre un

Bean de Spring, que se llame como el Filtro Web **DelegatingFilterProxy**.

Con XML

```
<filter>
  <filter-name>elNombreDeMiBeanEnSpring</filter-name>
  <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>
<filter-mapping>
  <filter-name>elNombreDeMiBeanEnSpring</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Con Java Config

```
public class AppInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {
        // ...
        servletContext.addFilter("elNombreDeMiBeanEnSpring",
            new DelegatingFilterProxy("elNombreDeMiBeanEnSpring"))
            .addMappingForUrlPatterns(null, false, "/*");
        // ...
    }
}
```

En los ejemplos anteriores, deberá existir en el contexto de Spring un Bean llamado **elNombreDeMiBeanEnSpring**, que implemente la interface **Filter** del API de Servlets.

Spring MVC

Introduccion

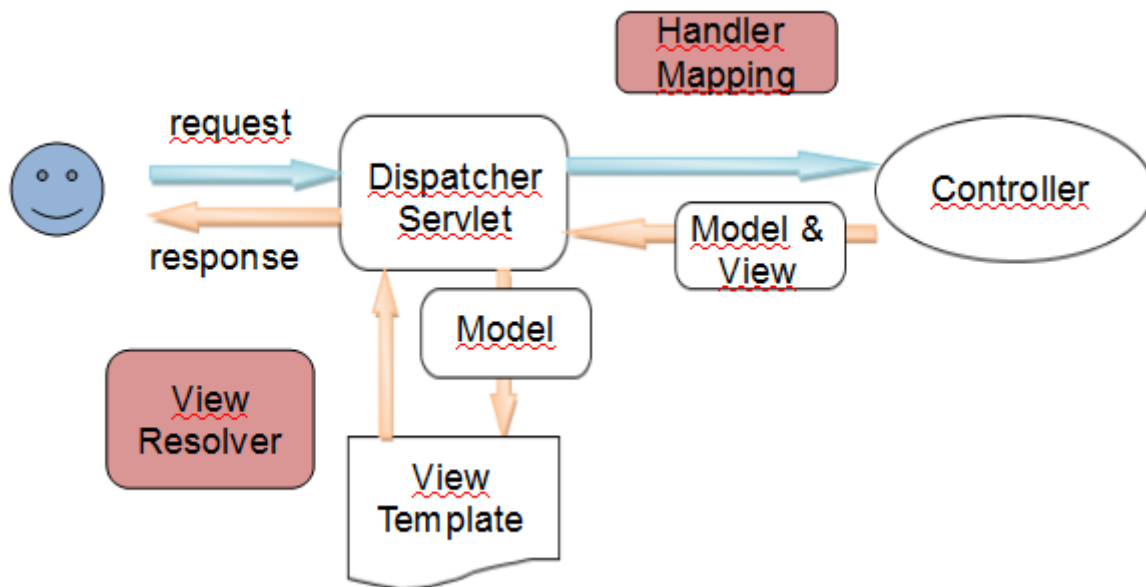
Spring MVC, como su nombre indica es un framework que implementa Modelo-Vista-Controlador, esto quiere decir que proporcionará componentes especializados en cada una de esas tareas.

Para incorporar las librerías con Maven, se añade al pom.xml

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
  <version>4.2.3.RELEASE</version>
</dependency>
```

Arquitectura

Spring MVC, como la mayoría de frameworks MVC, se basa en el patrón **FrontController**, en este caso el componente que realiza esta tarea es **DispatcherServlet**.



DispatcherServlet

El **DispatcherServlet**, realiza las siguientes tareas.

- Consulta con los **HandlerMapping**, que **Controller** ha de resolver la petición.
- Una vez el **HandlerMapping** le retorna que **Controller** ha de invocar, lo invoca para que resuelva la petición.
- Recoge los datos del **Model** que le envía el **Controller** como respuesta y el identificador de la **View** (o la propia **View** dependerá de la implementación del **Controller**) que se empleará para mostrar dichos datos.
- Consulta a la cadena de **ViewResolver** cual es la **View** a emplear, basandose en el identificador que le ha retornado el **Controller**.
- Procesa la **View** y el resultado lo retorna como resultado de la petición.

La configuración del **DispatcherServlet** se puede realizar siguiendo dos formatos

- Con ficheros XML. Para ello se han de declarar el servlet en el **web.xml**

```

<servlet>
  <servlet-name>miApp</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/miApp-servlet.xml</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>miApp</servlet-name>
  <url-pattern>/expedientesx/*</url-pattern>
</servlet-mapping>

```

NOTE

De no incluir el parametro de configuracion **contextConfigLocation** para el servlet, sera importante el nombre del servlet, ya que por defecto este buscara en el directorio WEB-INF, el xml de Spring con el nombre **<servlet-name>-servlet.xml** en este caso **miApp-servlet.xml**

Se puede incluir más de un fichero de configuracion de contexto, separandolos con comas.

- Con clases anotadas al estilo **JavaConfig**. Para ello el API proporciona una interface que se ha de implementar **WebApplicationInitializer** y allí se ha de registrar el servlet.

```

public class AppInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {
        WebApplicationContext context = getContext();
        ServletRegistration.Dynamic dispatcher = servletContext.addServlet(
            "DispatcherServlet", new DispatcherServlet(context));
        dispatcher.setLoadOnStartup(1);
        dispatcher.addMapping("/expedientesx/*");
    }

    private AnnotationConfigWebApplicationContext getContext() {
        AnnotationConfigWebApplicationContext context = new
        AnnotationConfigWebApplicationContext();
        context.setConfigLocation("expedientesx.cfg");
        return context;
    }
}

```

ContextLoaderListener

Adicionalmente, se puede definir otro contexto de Spring global a la aplicación, para ello se ha de

declarar el listener **ContextLoaderListener**, que al igual que el **DispatcherServlet** puede ser declarado de dos formas.

- Con XML

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    classpath:aplicacion.xml,
    /WEB-INF/seguridad.xml
  </param-value>
</context-param>
<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-
class>
</listener>
```

NOTE

Se puede incluir más de un fichero de configuracion de contexto, separandolos con comas.

- Con JavaConfig

```
public class AppInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {

        WebApplicationContext context = getContext();
        servletContext.addListener(new ContextLoaderListener(context));
    }

    private AnnotationConfigWebApplicationContext getContext() {
        AnnotationConfigWebApplicationContext context = new
AnnotationConfigWebApplicationContext();
        context.setConfigLocation("expedientesx.cfg");
        return context;
    }
}
```

NOTE

La clase **AnnotationConfigWebApplicationContext** es una clase capaz de descubrir y considerar los Beans declarados en clases anotadas con **@Configuration**

Namespace MVC

Se incluye el siguiente namespace con algunas etiquetas nuevas, que favorecen la configuracion del contexto

```
xmlns:mvc="http://www.springframework.org/schema/mvc"
```

ResourceHandler (Acceso a recursos directamente)

No todas las peticiones que se realizan a la aplicación necesitarán que se ejecute un **Controller**, algunas de ellas harán referencia a imagenes, hojas de estilo, ... Se puede añadir con XML o JavaConfig

Con XML

```
<mvc:resources mapping="/resources/**" location="/resources/" />
```

Donde **mapping** hace referencia al patrón de URL de la petición y **location** al directorio donde encontrar los recursos.

NOTE

La forma de abordar esta explicación, es retomar la arquitectura y el patrón **FrontController**, y la no necesidad de un **Controller** para ofrecer un recurso estatico, los **Controller** son necesarios para los recursos dinamicos, para los estaticos introducen demasida complejidad de forma inecesaria.

Con JavaConfig, se ha de hacer extender la clase **@Configuration** de **WebMvcConfigurerAdapter** y sobrescribir el método **addResourceHandlers** con lo siguiente.

```
@Override
public void addResourceHandlers(final ResourceHandlerRegistry registry) {
    registry.addResourceHandler("/resources/**").addResourceLocations("/resources/");
}
```

Donde **ResourceHandler** hace referencia al patrón de URL de la petición y **ResourceLocation** al directorio donde encontrar los recursos.

NOTE

La forma de abordar esta explicación, es retomar la arquitectura y el patrón **FrontController**, y la no necesidad de un **Controller** para ofrecer un recurso estatico, los **Controller** son necesarios para los recursos dinamicos, para los estaticos introducen demasida complejidad de forma inecesaria.

Default Servlet Handler

Cuando los recursos estaticos, estan situados en la carpeta **webapp**, se pueden sustituir las configuraciones anteriores por

```
<mvc:default-servlet-handler/>
```

```

@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {
    @Override
    public void configureDefaultServletHandling(DefaultServletHandlerConfigurer
configurer) {
        configurer.enable();
    }
}

```

ViewController (Asignar URL a View)

En ocasiones se necesita acceder a una **View** directamente, sin pasar por un controlador, para ello Spring MVC ofrece los **ViewControllers**. Se puede añadir con XML o JavaConfig

Con XML

```

<mvc:view-controller path="/" view-name="welcome" />

```

Con JavaConfig, de nuevo se ha de hacer extender la clase **@Configuration** de la clase **WebMvcConfigurerAdapter**, en este caso implementando el método

```

@Override
public void addViewControllers(ViewControllersRegistry registry) {
    registry.addViewController("/").setViewName("index");
}

```

En este caso **ViewController** representa el path que le llega al **DispatcherServlet** y **ViewName** el nombre de la **View** que deberá ser resuelto por un **ViewResolver**.

NOTE Los **ViewController** se resuelven posteriormente a los **Controller** anotados con **RequestMapping**, por lo que si se emplean mappings con path similares en ambos escenarios, nunca se llegará a los **ViewController**, para conseguirlo se ha de configurar la precedencia del **ViewControllerRegistry** a un valor inferior al del **RequestMappingHandlerMapping**.

NOTE Los **ViewController** no pueden acceder a elementos del Modelo definidos con **@ModelAttribute**, ya que estos son interpretados por el **RequestMappingHandlerMapping**, que no participa en el proceso de resolución de los **ViewController**

HandlerMapping

Es el primero de los componentes necesarios dentro del flujo de Spring MVC, siendo el encargado

de encontrar el controlador capaz de procesar la petición recibida.

Este componente extrae de la URL un Path, que coteja con las entradas configuradas dependiendo de la implementación empleada.

Para activar los HandlerMapping unicamente hay que declararlos en el contexto de Spring como Beans.

NOTE

Dado que se pueden configurar varios **HandlerMapping**, para establecer en que orden se han de emplear, existe la propiedad **Order**

El API proporciona las siguientes implementaciones

- **BeanNameUrlHandlerMapping** → Usa el nombre del Bean **Controller** como mapeo `<bean name="/inicio.htm" ... >`, debe comenzar por `/`.
- **SimpleUrlHandlerMapping** → Mapea mediante propiedades `<prop key="/verClientes.htm">beanControlador</prop>`
- **ControllerClassNameHandlerMapping** → Usa el nombre de la clase asumiendo que termina en **Controller** y sustituyéndola la palabra **Controller** por **.htm**
- **DefaultAnnotationHandlerMapping** → Emplea la propiedad `path` de la anotación `@RequestMapping`

NOTE

Las implementaciones por defecto en Spring MVC 3 son **BeanNameUrlHandlerMapping** y **DefaultAnnotationHandlerMapping**

BeanNameUrlHandlerMapping

Al emplear esta configuración, cuando lleguen peticiones con path `/helloWoorld.html`, el **Controller** que lo procesará será de tipo **EjemploAbstractController**

```
<bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping" />
<bean name="/helloWorld.html" class="org.ejemplos.springmvc.HelloWorldController" />
```

SimpleUrlHandlerMapping

```
<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
  <property name="mappings">
    <props>
      <prop key="/helloWorld.htm">helloWorldController</prop>
    </props>
  </property>
</bean>
<bean name="helloWorldController" class="org.ejemplos.springmvc.HelloWorldController" />
```

ControllerClassNameHandlerMapping

```
public class HelloWorldController extend AbstractController{
    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest request,
        HttpServletResponse response)
        throws Exception {
        return new ModelAndView("otro");
    }
}
```

DefaultAnnotationHandlerMapping

```
@RequestMapping("helloWorld")
public class HelloWorldController extend AbstractController{
    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest request,
        HttpServletResponse response)
        throws Exception {
        return new ModelAndView("otro");
    }
}
```

RequestMappingHandlerMapping

Esta implementacion permite interpretar las anotaciones **@RequestMapping** en los controladores, haciendo coincidir la url, con el atributo **path** de dichas anotaciones.

```
@RequestMapping("helloWorld")
public class HelloWorldController extend AbstractController{
    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest request,
        HttpServletResponse response)
        throws Exception {
        return new ModelAndView("otro");
    }
}
```

El espacio de nombres **mvc**, ofrece una etiqueta que simplifica la configuracion

```
<mvc:annotation-driven/>
```

Tambien se ofrece una anotacion **@EnableWebMvc** a añadir a la clase **@Configuration** para la configuracion con JavaConfig, esta anotación, define por convencion una pila de **HandlerMapping**, ya que en realidad lo que hace es cargar la clase **WebMvcConfigurationSupport** como calse

@Configuration, en esta clase se describen los **HandlerMapping** cargados.

```
@Configuration
@EnableWebMvc
public class ContextoGlobal {
}
```

NOTE

En la ultima version de Spring no es necesario añadirlo, la unica diferencia al añadirlo, es que se consideran menos path validos para cada **@RequestMapping** definido, con ella solo **/helloWorld** y sin ella **/helloWorld**, **/helloWorld.** * y **/helloWorld/**

Controller

El siguiente de los componentes en el que delega el **DispatcherServlet**, será el encargado de ejecutar la logica de negocio.

Spring proporciona las siguientes implementaciones

- **AbstractController**
- **ParametrizableViewController**
- **AbstractCommandController**
- **SimpleFormController**
- **AbstractWizardFormController**
- **@Controller**

@Controller

Anotacion de Clase, que permite indicar que una clase contiene funcionalidades de Controlador.

```
@Controller
public class HelloWorldController {
    @RequestMapping("helloWorld")
    public String helloWorld(){
        return "exito";
    }
}
```

La firma de los métodos de la clase anotada es flexible, puede retornar

- String
- View
- ModelAndView
- Objeto (Anotado con **@ResponseBody**)

Y puede recibir como parámetro

- **Model** → Datos a emplear en la **View**.
- Parametros anotados con **@PathVariable** → Dato que llega en el path de la Url.
- Parametros anotados con **@RequestParam** → Dato que llega en los parametros de la Url.
- **HttpServletRequest**
- **HttpServletResponse**
- **HttpSession**

Activación de @Controller

Para activar esta anotación, habra que indicarle al contexto de Spring a partir de que paquete puede buscarla. Se puede hacer con XML y con JavaConfig

Con XML, se emplea la etiqueta **ComponentScan**

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

NOTE

Esta etiqueta activa el descubrimiento de las clases anotadas con @Component, @Repository, @Controller y @Service

Con JavaConfig, se emplea la anotacion **@ComponentScan**

```
@Configuration
@EnableWebMvc
@ComponentScan(basePackages={ "controllers" })
public class ContextoGlobal {

}
```

NOTE

Esta anotacion activa el descubrimiento de las clases anotadas con @Component, @Repository, @Controller y @Service

@PathVariable

Anotacion que permite obtener información de la url que provoca la ejecucion del controlador.

```
@RequestMapping(path="/saludar/{nombre}")
public ModelAndView saludar(@PathVariable("nombre") String nombre){
}
```

Para el anterior ejemplo, dada la siguiente url <http://...../saludar/Victor>, el valor del parametro **nombre**, será **Victor**

NOTE

Se pueden definir expresiones regulares para alimentar a los `@PathVariable`, siguiendo la firma `{varName:regex}`, por ejemplo

```
@RequestMapping("/spring-web/{symbolicName:[a-z]-  
{version:\\d\\.\\d\\.\\d}{extension:\\.[a-z]}") public void handle(@PathVariable String  
version, @PathVariable String extension) { // ... }
```

@RequestParam

Anotacion que permite obtener información de los parametros de la url que provoca la ejecucion del controlador.

```
@RequestMapping(path="/saludar")  
public ModelAndView saludar(@RequestParam("nombre") String nombre){  
}
```

Para el anterior ejemplo, dada la siguiente url <http://...../saludar?nombre=Victor>, el valor del parametro **nombre**, será **Victor**

@RequestBody

Permite tranformar el contenido del **body** de peticiones **POST** o **PUT** a un objeto java, tipicamente una representación en JSON.

```
@RequestMapping(path="/alta", method=RequestMethod.POST)  
public String getDescription(@RequestBody UserStats stats){  
    return "resultado";  
}  
  
public class UserStats{  
    private String firstName;  
    private String lastName;  
}
```

En el ejemplo anterior, se convertirán a objeto, contenidos del **body** de la petición como por ejemplo

```
{ "firstName" : "Elmer", "lastName" : "Fudd" }
```

Para transformaciones a JSON, se emplea la siguiente libreria de **Jackson**

```
<dependency>
  <groupId>org.codehaus.jackson</groupId>
  <artifactId>jackson-mapper-asl</artifactId>
  <version>1.4.2</version>
</dependency>
```

@ResponseBody

Análogo al anterior, pero para generar un resultado.

Se aplica sobre métodos que retornan un objeto de información.

```
// controller
@ResponseBody
@RequestMapping("/description")
public Description getDescription(@RequestBody UserStats stats){
    return new Description(stats.getFirstName() + " " + stats.getLastName() + " hates
wacky wabbits");
}

public class UserStats{
    private String firstName;
    private String lastName;
    // + getters, setters
}

public class Description{
    private String description;
    // + getters, setters, constructor
}
```

Precisa dar de alta el API de marshall en el classpath.

```
<dependency>
  <groupId>org.codehaus.jackson</groupId>
  <artifactId>jackson-mapper-asl</artifactId>
  <version>1.4.2</version>
</dependency>
```

Es muy empleado en servicios REST.

@ModelAttribute

Se pueden añadir Beans al objeto **Model** de un controlador en el ambito **request** con la anotación **@ModelAttribute**.

```
@Controller
public class MyController {

    @ModelAttribute("persona")
    public Persona addPersonaToModel() {
        return new Persona("Victor");
    }
}
```

@SessionAttributes

También se puede asociar a la **session**, para ello se emplea la anotación **@SessionAttributes("nombreDelBeanDelModeloAAlmacenarEnLosAtributosDeLaSession")**, incluyéndola como anotación de clase en la clase **Controller** que declare el bean del modelo con **@ModelAttribute**.

```
@Controller
@SessionAttributes("persona")
public class MyController {

    @ModelAttribute("persona")
    public Persona addPersonaToModel() {
        return new Persona("Victor");
    }
}
```

Los objetos en Model, pueden ser inyectados directamente en los métodos del controlador con **@ModelAttribute**

```
@RequestMapping("/saludar")
public String saludar (@ModelAttribute("persona") Persona persona, Model model) {
    return "exito";
}
```

@InitBinder

Permite redefinir:

- CustomFormatter → Permite definir transformaciones de tipos, se basa en la interface **Formatter**
- Validators → Validadores nuevos a aplicar a los Bean del Modelo, se basa en **Validator**
- CustomEditor → Parseos a aplicar a campos de los formularios, se basan en **PropertyEditor**

```
@InitBinder
public void customizeBinding(WebDataBinder binder) {

}
```

@ExceptionHandler

Permiten definir vistas a emplear cuando se producen excepciones en los métodos de control

```
@ExceptionHandler(CustomException.class)
public ModelAndView handleCustomException(CustomException ex) {

    ModelAndView model = new ModelAndView("error");
    model.addObject("ex", ex);
    return model;
}
```

@ControllerAdvice

Permiten definir en una clase independiente configuraciones de **@ExceptionHandler**, **@InitBinder** y **@ModelAttribute** que afectaran a los controladores que se desee.

```
@ControllerAdvice(basePackages="com.viewnext.holamundo.javaconfig.controllers")
public class GlobalConfig {
    @ModelAttribute
    public void initGlobal(Model model) {
        model.addAttribute("persona", new Persona());
    }
}
```

ViewResolver

El último componente a definir del flujo es el **ViewResolver**, este componente se encarga de resolver que **View** se ha emplear a partir del objeto **View** retornado por el **Controller**.

Pueden existir distintos **Bean** definidos de tipo **ViewResolver**, pudiendose ordenar con la propiedad **Order**.

NOTE

Es importante que de emplear el **InternalResourceViewResolver**, este sea el ultimo (Valor mas alto).

Se proporcionan varias implementaciones, alguna de ellas

- **InternalResourceViewResolver** → Es el más habitual, permite interpretar el **String** devuelto por el **Controller**, como parte de la url de un recurso, componiendo la URL con un prefijo y un

sufijo.

- **BeanNameViewResolver** → Busca un **Bean** declarado de tipo **View** cuyo **Id** sea igual al **String** retornado por el **Controller**.
- **ContentNegotiatingViewResolver** → Delega en otros **ViewResolver** dependiendo del **ContentType**.
- **FreeMarkerViewResolver** → Similar al **InternalResourceViewResolver**, pero el recurso buscado debe ser una plantilla Freemarker.
- **JasperReportsViewResolver** → Similar al **InternalResourceViewResolver**, pero el recurso buscado debe ser una plantilla JasperReport.
- **ResourceBundleViewResolver** → Busca la implementacion de la View en un fichero de properties.
- **TilesViewResolver** → Busca una plantillas de **Tiles** con nombre igual al **String** retornado por el **Controller**
- **VelocityViewResolver** → Similar al **InternalResourceViewResolver**, pero el recurso buscado debe ser una plantilla Velocity.
- **XmlViewResolver** → Similar a **BeanNameViewResolver**, salvo porque los **Bean** de las **View** han de ser declaradas en el fichero **/WEB-INF/views.xml**
- **XsltViewResolver** → Similar al **InternalResourceViewResolver**, pero el recurso buscado debe ser una plantilla XSLT.

InternalResourceViewResolver

Se ha de definir el Bean

```
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <propertyname="prefix" value="/WEB-INF/views/" />
    <propertyname="suffix" value=".jsp" />
</bean>
```

XmlViewResolver

Se ha de definir el Bean

```
<bean class="org.springframework.web.servlet.view.XmlViewResolver">
    <property name="location" value="/WEB-INF/views.xml" />
    <property name="order" value="0" />
</bean>
```

Y en el fichero **/WEB-INF/views.xml**

```
<bean id="pdf/listado" class="com.aplicacion.presentacion.vistas.ListadoPdfView"/>
<bean id="excel/listado" class="com.aplicacion.presentacion.vistas.ListadoExcelView"/>
<bean id="json/listado" class=
"org.springframework.web.servlet.view.json.MappingJacksonJsonView"/>
```

ResourceBundleViewResolver

Se ha de definir el Bean

```
<bean id="viewResolver" class=
"org.springframework.web.servlet.view.ResourceBundleViewResolver">
    <property name="basename" value="views" />
</bean>
```

Y en el fichero **views.properties** que estará en la raíz del classpath.

```
listado.(class)=org.springframework.web.servlet.view.jasperreports.JasperReportsPdfView
listado.url=/WEB-INF/jasperTemplates/reporteAfines.jasper
listado.reportDataKey=listadoKey
```

Donde **url** y **reportDataKey**, son propiedades del objeto **JasperReportsPdfView**, y **listado** el **String** que retorna el **Controller**

View

Son los componentes que renderizaran la respuesta a la petición procesada por Spring MVC.

Existen diversas implementaciones dependiendo de la tecnología encargada de renderizar.

- AbstractExcelView
- AbstractAtomFeedView
- AbstractRssFeedView
- MappingJackson2JsonView
- MappingJackson2XmlView
- AbstractPdfView
- AbstractJasperReportView
- AbstractPdfStamperView
- AbstractTemplateView
- InternalResourceView
- JstlView → Es la que se emplea habitualmente para los JSP, exige la librería JSTL.

- TilesView
- XsltView

AbstractExcelView

El API de Spring proporciona una clase abstracta que esta destinada a hacer de puente entre el API capaz de generar un Excel y Spring, pero no genera el Excel, para ello hay que incluir una libreria como **POI**

```
<dependency>
  <groupId>org.apache.poi</groupId>
  <artifactId>poi</artifactId>
  <version>3.10.1</version>
</dependency>
```

Algunas de las clases que proporciona **POI** son

- HSSFWorkbook
- HSSFSheet
- HSSFRow
- HSSFCell

```
public class PoiExcelView extends AbstractExcelView {
    @Override
    protected void buildExcelDocument(Map<String, Object> model, HSSFWorkbook
workbook, HttpServletRequest request, HttpServletResponse response) throws Exception {
        // model es el objeto Model que viene del Controller
        List<Book> listBooks = (List<Book>) model.get("listBooks");
        // Crear una nueva hoja excel
        HSSFSheet sheet = workbook.createSheet("Java Books");
        sheet.setDefaultColumnWidth(30);
        HSSFRow header = sheet.createRow(0);
        header.createCell(0).setCellValue("Book Title");
        header.createCell(1).setCellValue("Author");
        int rowCount = 1;
        for (Book aBook : listBooks) {
            HSSFRow aRow = sheet.createRow(rowCount++);
            aRow.createCell(0).setCellValue(aBook.getTitle());
            aRow.createCell(1).setCellValue(aBook.getAuthor());
        }
        response.setHeader("Content-disposition", "attachment; filename=books.xls");
    }
}
```

AbstractPdfView

De forma analoga al anterior, para los PDF, se tiene la libreria **Lowagie**

```
<dependency>
  <groupId>com.lowagie</groupId>
  <artifactId>itext</artifactId>
  <version>4.2.1</version>
</dependency>
```

Algunas de las clases que proporciona **Lowagie** son

- Document
- PdfWriter
- Paragraph
- Table

```
public class ITextPdfView extends AbstractPdfView {
    @Override
    protected void buildPdfDocument(Map<String, Object> model, Document doc, PdfWriter
writer, HttpServletRequest request, HttpServletResponse response) throws Exception {
        // model es el objeto Model que viene del Controller
        List<Book> listBooks = (List<Book>) model.get("listBooks");
        doc.add(new Paragraph("Recommended books for Spring framework"));
        Table table = new Table(2);
        table.addCell("Book Title");
        table.addCell("Author");
        for (Book aBook : listBooks) {
            table.addCell(aBook.getTitle());
            table.addCell(aBook.getAuthor());
        }
        doc.add(table);
    }
}
```

JasperReportsPdfView

En este caso Spring proporciona una clase concreta, que es capaz de procesar las platillas de **JasperReports**, lo unico que necesita es la libreria de **JaspertReport**, la plantilla compilada **jasper** y un objeto **JRBeanCollectionDataSource** que contenga la información a representar en la plantilla.

NOTE La plantilla sin compilar será un fichero **jrxml**, que es un xml editable.

```
<dependency>
  <groupId>jasperreports</groupId>
  <artifactId>jasperreports</artifactId>
  <version>3.5.3</version>
</dependency>
```

NOTE

A tener en cuenta que la version de la libreria de JasperReport debe coincidir con la del programa iReport empleando para generar la plantilla.

```
<bean id="reporteAfines" class=
"org.springframework.web.servlet.view.jasperreports.JasperReportsPdfView">
  <property name="url" value="/WEB-INF/jasperTemplates/reportes.jasper"/>
  <property name="reportDataKey" value="listadoKey"></property>
</bean>
```

NOTE

reportDataKey indica la clave dentro del objeto **Model** que referencia al objeto **JRBeanCollectionDataSource**

```
@Controller
public class AfinesReportController {
    @RequestMapping("/reporte")
    public String generarReporteAfines(Model model){
        JRBeanCollectionDataSource jrbean = new JRBeanCollectionDataSource(listado,
false);
        model.addAttribute("listadoKey", jrbean);
        return "reporteAfines";
    }
}
```

MappingJackson2JsonView

```
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.4.1</version>
</dependency>
```

NOTE

Es para versiones de Spring posteriores a 4, para la 3 se emplea otro API y la clase **MappingJacksonJsonView**

Los Bean a convertir a JSON, han de tener propiedades.

Formularios

Para trabajar con formularios Spring proporciona una librería de etiquetas

```
<%@ taglib prefix="sf" uri="http://www.springframework.org/tags/form"%>
```

Tag	Descripción
checkbox	Renders an HTML 'input' tag with type 'checkbox'.
checkboxes	Renders multiple HTML 'input' tags with type 'checkbox'.
errors	Renders field errors in an HTML 'span' tag.
form	Renders an HTML 'form' tag and exposes a binding path to inner tags for binding.
hidden	Renders an HTML 'input' tag with type 'hidden' using the bound value.
input	Renders an HTML 'input' tag with type 'text' using the bound value.
label	Renders a form field label in an HTML 'label' tag.
option	Renders a single HTML 'option'. Sets 'selected' as appropriate based on bound value.
options	Renders a list of HTML 'option' tags. Sets 'selected' as appropriate based on bound value.
password	Renders an HTML 'input' tag with type 'password' using the bound value.
radiobutton	Renders an HTML 'input' tag with type 'radio'.
select	Renders an HTML 'select' element. Supports databinding to the selected option.

Un ejemplo de definición de formulario podría ser

```
<form:form action="altaUsuario" modelAttribute="persona">
  <table>
    <tr>
      <td>Nombre:</td>
      <td><form:input path="nombre" /></td>
    </tr>
    <tr>
      <td>Apellidos:</td>
      <td><form:input path="apellidos" /></td>
    </tr>
    <tr>
      <td>Sexo:</td>
      <td><form:select path="sexo" items="${listadoSexos}" />
    </tr>
    <tr>
      <td colspan="2">
        <input type="submit" value="Guardar info" />
      </td>
    </tr>
  </table>
</form:form>
```

NOTE

No es necesario definir el action si se emplea la misma url para cargar el formulario y para recibirlo, basta con cambiar unicamente el METHOD HTTP No hay diferencia entre **commandName** y **modelAttribute**

En el ejemplo anterior, se han definido a nivel del formulario.

- **action** → Indica la Url del Controlador.
- **modelAttribute** → Indica la clave con la que se envía el objeto que se representa en el formulario. (de forma analoga se puede emplear **commandName**)

Para recuperar en el controlador el objeto enviado, se emplea la anotación **@ModelAttribute**

El objeto que se representa en el formulario ha de existir al representar el formulario. Es típico para los formularios definir dos controladores uno GET y otro POST.

- El GET inicializara el objeto.
- El POST tratara el envío del formulario.

```
@RequestMapping(value="altaPersona", method=RequestMethod.GET)
public String inicializacionFormularioAltaPersonas(Model model){
    Persona p = new Persona(null, "", "", null, "Hombre", null);
    model.addAttribute("persona", p);
    model.addAttribute("listadoSexos", new String[]{"Hombre","Mujer"});
    return "formularioAltaPersona";
}

@RequestMapping(value="altaPersona", method=RequestMethod.POST)
public String procesarFormularioAltaPersonas(
    @ModelAttribute("persona") Persona p, Model model){
    servicio.altaPersona(p);
    model.addAttribute("estado", "OK");
    model.addAttribute("persona", p);
    model.addAttribute("listadoSexos", new String[] {"Hombre","Mujer"});
    return "formularioAltaPersona";
}
```

Etiquetas

Spring proporciona dos librerías de etiquetas

- Formularios
- **<form:form></form:form>** → Crea una etiqueta HTML form.
- **<form:errors></form:errors>** → Permite la visualización de los errores asociados a los campos del **ModelAttribute**
- **<form:checkboxes items="" path=""/>** →

```
<form:checkbox path=""/>
```

```
<form:hidden path=""/>
```

```
<form:input path=""/>
```

```
<form:label path=""></form:label>
```

```
<form:textarea path=""/>
```

```
<form:password path=""/>
```

```
<form:radiobutton path=""/>
```

```
<form:radiobuttons path=""/>
```

```
<form:select path=""></form:select>
```

```
<form:option value=""></form:option>
```

```
<form:options/>
```

```
<form:button></form:button>
```

- Core

```
<spring:argument></spring:argument>
```

```
<spring:bind path=""></spring:bind>
```

```
<spring:escapeBody></spring:escapeBody>
```

```
<spring:eval expression=""></spring:eval>
```

```
<spring:hasBindErrors name=""></spring:hasBindErrors>
```

```
<spring:htmlEscape defaultHtmlEscape=""></spring:htmlEscape>
```

```
<spring:message></spring:message>
```

```
<spring:nestedPath path=""></spring:nestedPath>
```

```
<spring:param name=""></spring:param>
```

```
<spring:theme></spring:theme>
```

```
<spring:transform value=""></spring:transform>
```

```
<spring:url value=""></spring:url>
```

Paths Absolutos

En ocasiones, se requiere acceder a un controlador desde distintas JSP, las cuales estan a distinto nivel en el path, por ejemplo desde **/gestion/persona** y desde **/administracion**, se quiere acceder a **/buscar**, teniendo en cuenta que la propiedad **action** representa un path relativo, no serviria en mismo formulario, salvo que se pongan path absolutos, para los cual, se necesita obtener la url de la aplicación, hay varias alternativas

- Expresiones EL

```
<form action="${pageContext.request.contextPath}/buscar" method="GET" />
```

- Libreria de etiquetas JSTL core

```
<form action="<c:url value="/buscar" />" method="GET" />
```

Inicialización

Otra opción para inicializar los objetos necesarios para el formulario, sería crear un método anotado con **@ModelAttribute**, indicando la clave del objeto del Modelo que disparará la ejecución de este método.

```
@ModelAttribute("persona")
public Persona initPersona(){
    return new Persona();
}
```

Validaciones

Spring MVC soporta validaciones de JSR-303.

Para aplicarlas se necesita una implementación como **hibernate-validator**, para añadirla con Maven.

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-validator</artifactId>
  <version>5.1.3.Final</version>
</dependency>
```

Para activar la validación entre **View** y **Controller**, se añade a los parámetros de los métodos del **Controller**, la anotación **@Valid**.

```
@RequestMapping(method = RequestMethod.POST)
public Persona altaPersona(@Valid @RequestBody Persona persona) {}
```

Si además se quiere conocer el estado de la validación para ejecutar la lógica del controlador, se puede indicar en los parámetros que se recibe un objeto **Errors**, que tiene un método **hasErrors()** que indica si hay errores de validación.

```
public String altaPersona(@Valid @ModelAttribute("persona") Persona p,
    Errors errors, Model model){

    if (errors.hasErrors()) {
        return "error";
    } else {
        return "ok";
    }
}
```

Y en la clase del **Model**, las anotaciones correspondientes de JSR-303


```
public class Persona {
    @NotEmpty(message="Hay que rellenar el campo nombre")
    private String nombre;
    @NotEmpty
    private String apellido;
    private int edad;
}
```

Mensajes personalizados

Como se ve en el anterior ejemplo, se ha personalizado el mensaje para la validación **@NotEmpty** del campo **nombre**

Se puede definir el mensaje en un properties, teniendo en cuenta que el property tendra la siguiente firma

```
<validador>.<entidad>.<caracteristica>
```

Por ejemplo para la validación anterior de **nombre**

```
notempty.persona.nombre = Hay que rellenar el campo nombre
```

Tambien se puede referenciar a una propiedad cualquiera, pudiendo ser cualquier clave.

```
@NotEmpty(message="{notempty.persona.nombre}")
private String nombre;
```

Anotaciones JSR-303

Las anotaciones están definidas en el paquete **javax.validation.constraints**.

- **@Max**
- **@Min**
- **@NotNull**
- **@Null**
- **@Future**
- **@Past**
- **@Size**
- **@Pattern**

Validaciones Custom

Se pueden definir validadores nuevos e incluirlos en la validación automatizada, para ello hay que implementar la interface **org.springframework.validation.Validator**

```
public class PersonaValidator implements Validator {
    @Override
    public boolean supports(Class<?> clazz) {
        return Persona.class.equals(clazz);
    }
    @Override
    public void validate(Object obj, Errors e) {
        Persona persona = (Persona) obj;
        e.rejectValue("nombre", "formulario.persona.error.nombre");
    }
}
```

NOTE

El metodo de supports, indica que clases se soportan para esta validación, si retornase true, aceptaria todas, no es lo habitual ya que tendrá al menos una característica concreta que será la validada.

Una vez definido el validador, para añadirlo al flujo de validación de un **Controller**, se ha de añadir una instancia de ese validador al **Binder** del **Controller**, creando un método en el **Controller**, anotado con **@InitBinder**

```
@InitBinder
protected void initBinder(final WebDataBinder binder) {
    binder.addValidators(new PersonaValidator());
}
```

Los errores asociados a estas validaciones pueden ser visualizados en la **View** empleando la etiqueta **<form:errors/>**

```
<form:errors path="*" />
```

NOTE

La propiedad path, es el camino que hay que seguir en el objeto de **Model** para acceder a la propiedad validada.

Internacionalización - i18n

Para poder aplicar la internacionalización, hay que trabajar con ficheros properties manejados como **Bundles**, esto en Spring se consigue definiendo un **Bean** con id **messageSource** de tipo **AbstractMessageSource**

```
<bean id="messageSource" class=
"org.springframework.context.support.ReloadableResourceBundleMessageSource">
    <property name="basename" value="/WEB-INF/messages/messages" />
</bean>
```

Una vez definido el Bean deberán existir tantos ficheros como idiomas soportados con la firma

```
/WEB-INF/messages/messages_<COD-PAIS>_<COD-DIALECTO>.properties
```

Como por ejemplo

```
/WEB-INF/messages/messages_es.properties
/WEB-INF/messages/messages_es_es.properties
/WEB-INF/messages/messages_en.properties
```

Para acceder a estos mensajes desde las **View** existe una libreria de etiquetas

```
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring"%>
```

Que proporciona la etiqueta

```
<spring:message code="<clave en el properties>"/>
```

Tambien es posible emplear JSTL

```
<dependency>
    <groupId>jstl</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
</dependency>
```

```
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
```

```
<fmt:message key="<clave en el properties>"/>
```

Interceptor

Permiten interceptar las peticiones al **DispatcherServlet**.

Son clases que extienden de **HandlerInterceptorAdapter**, que permite actuar sobre la petición con

tres métodos.

- **preHandle()** → Se invoca antes que se ejecute la petición, retorna un booleano, si es **True** continua la ejecución normalmente, si es **False** la para.
- **postHandle()** → Se invoca después de que se ejecute la petición, permite manipular el objeto **ModelAndView** antes de pasárselo a la **View**.
- **afterCompletion()** → Called after the complete request has finished. Seldom use, can't find any use case.

Los **Interceptor** pueden ser asociados

- A cada **HandlerMapping** en particular, con la propiedad **interceptors**.

```
<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
  <property name="mappings">
    <props>
      <prop key="/index.html">indexController</prop>
    </props>
  </property>
  <property name="interceptors">
    <list>
      <ref bean="auditoriaInterceptor" />
    </list>
  </property>
</bean>

<bean id="auditoriaInterceptor" class=
"com.ejemplo.mvc.interceptor.AuditoriaInterceptor" />

<bean id="indexController" class="com.ejemplo.mvc.interceptor.IndexController" />
```

- O de forma general a todos

Con XML, se emplearía la etiqueta del namespace **mvc**

```
<mvc:interceptors>
  <bean class="com.ejemplo.mvc.interceptor.AuditoriaInterceptor" />
</mvc:interceptors>
```

Con **JavaConfig**, sobrescribiendo el método **addInterceptors** obtenido por la herencia de **WebMvcConfigurerAdapter**

```

@EnableWebMvc
@Configuration
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new LocaleInterceptor());
    }

}

```

Se proporcionan las siguientes implementaciones

- **ConversionServiceExposingInterceptor** → Situa el **ConversionService** en la **request**.
- **LocaleChangeInterceptor** → Permite interpretar el parámetro **locale** de la petición para cambiar el **Locale** de la aplicación.
- **ResourceUrlProviderExposingInterceptor** → Situa el **ResourceUrlProvider** en la **request**.
- **ThemeChangeInterceptor** → Permite interpretar el parámetro **theme** de la petición para cambiar el **Tema** (conjunto de estilos) de la aplicación.
- **UriTemplateVariablesHandlerInterceptor** → Se encarga de resolver las variables del Path y ponerlas en la **request**.
- **UserRoleAuthorizationInterceptor** → Comprueba la autorizacion del usuario actual, validando sus roles.

LocaleChangeInterceptor

Se declara el **Interceptor**.

```

<mvc:interceptors>
    <bean id="localeChangeInterceptor" class=
"org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
        <property name="paramName" value="language" />
    </bean>
</mvc:interceptors>

```

Para cambiar el **Locale** basta con acceder a la URL

```
http://.....?language=es
```

NOTE Por defecto el parametro que representa el codigo idiomático es **locale**

Se puede configurar como se almacena la referencia al **Locale**, para ello basta con definir un Bean llamado **localeResolver** de tipo

- Para almacenamiento en una **Cookie**

```
<bean id="localeResolver" class=
"org.springframework.web.servlet.i18n.CookieLocaleResolver">
    <property name="defaultLocale" value="es" />
    <property name="cookieName" value="myAppLocaleCookie"></property>
    <property name="cookieMaxAge" value="3600"></property>
</bean>
```

- Para almacenamiento en la **Session**

```
<bean id="localeResolver" class=
"org.springframework.web.servlet.i18n.SessionLocaleResolver" />
```

- El por defecto, busca en la cabecera **accept-language**

```
<bean id="localeResolver" class=
"org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver"/>
```

ThemeChangeInterceptor

Se declara el **Interceptor**.

```
<mvc:interceptors>
    <bean id="themeChangeInterceptor" class=
"org.springframework.web.servlet.theme.ThemeChangeInterceptor">
        <property name="paramName" value="theme" />
    </bean>
</mvc:interceptors>
```

Para cambiar el **Tema** basta con acceder a la URL

```
http://.....?theme=aqua
```

Tambien se ha de declarar un Bean que indique el nombre del fichero **properties** que almacenará el nombre de los ficheros de estilos a emplear en cada **Tema**, este Bean se ha de llamar **themeSource**

```
<bean id="themeSource" class=
"org.springframework.ui.context.support.ResourceBundleThemeSource">
    <property name="basenamePrefix" value="theme-" />
</bean>
```

Se puede configurar como se almacena la referencia al **Tema**, para ello basta con definir un Bean llamado **themeResolver** de tipo

- Para almacenamiento en una **Cookie**

```
<bean id="themeResolver" class=
"org.springframework.web.servlet.theme.CookieThemeResolver">
    <property name="defaultThemeName" value="default" />
</bean>
```

- Para almacenamiento en la **Session**

```
<bean id="themeResolver" class=
"org.springframework.web.servlet.i18n.SessionThemeResolver" >
    <property name="defaultThemeName" value="default" />
</bean>
```

Para poder aplicar alguna de las hojas de estilos definidas en el tema, se puede emplear la etiqueta **spring:theme**

```
<link rel="stylesheet" href="<spring:theme code='css'/>" type="text/css" />

<spring:theme code="welcome.message" />
```

Thymeleaf

Motor de plantillas.

Define el espacio de nombres **th** que proporciona atributos para instrumentalizar las etiquetas **xhtml**.

```
<html xmlns:th="http://www.thymeleaf.org"></html>
```

Para emplearlo, se han de añadir los siguientes Bean a la configuración

```

<bean id="templateResolver" class=
"org.thymeleaf.templateresolver.ServletContextTemplateResolver">
    <property name="prefix" value="/WEB-INF/templates/" />
    <property name="suffix" value=".html" />
    <property name="templateMode" value="HTML5" />
</bean>

<bean id="templateEngine" class="org.thymeleaf.spring4.SpringTemplateEngine">
    <property name="templateResolver" ref="templateResolver" />
</bean>

<bean class="org.thymeleaf.spring4.view.ThymeleafViewResolver">
    <property name="templateEngine" ref="templateEngine" />
</bean>

```

Con esto se considera que cualquier fichero con extension **.html** que se encuentre en la carpeta **/WEB-INF/templates/** a la que se haga referencia por el nombre del fichero como plantilla para una **View**, se resolverá con **Thymeleaf**.

Se pueden emplear dos tipos de expresiones dentro de los **HTML**, **\${}** y **#...**

En Spring Boot se genera una cache para las platillas, la cual se puede deshabilitar para desarrollo

```

spring:
  thymeleaf:
    cache: false

```

HttpMessageConverters

Son los encargados de realizar el Marshall y el Unmarshall de tipologías complejas a formatos de representación como json o xml.

El contexto de Spring los emplea cuando los métodos de los controladores emplean

- **@ResponseBody** → Indica que se debe transformar un objeto retornado por el método de controlador a un formato de representación marcado por la cabecera **Accept** y retornarlo en el cuerpo de la respuesta.
- **@RequestBody** → Indica que se debe leer el cuerpo de la petición como un objeto cuyo tipo de representación viene marcado por la cabecera **ContentType**.

El uso de los converters se activa en los xml con

```
<mvc:annotation-driven/>
```

y con java config con

Pila por defecto de `HttpMessageConverters`

Por defecto al activar Spring MVC, se carga la siguiente pila de converters.

- **ByteArrayHttpMessageConverter** → convierte los arrays de bytes
- **StringHttpMessageConverter** → convierte las cadenas de caracteres
- **ResourceHttpMessageConverter** → convierte a objetos `org.springframework.core.io.Resource` desde y hacia cualquier **Stream**.
- **SourceHttpMessageConverter** → convierte a **javax.xml.transform.Source**
- **FormHttpMessageConverter** → convierte datos de formulario (`application/x-www-form-urlencoded`) desde y hacia un **MultiValueMap<String, String>**.
- **Jaxb2RootElementHttpMessageConverter** → convierte objetos Java desde y hacia XML, con media type **text/xml** o **application/xml** (solo si la librería de JAXB2 está presente en el classpath).

```
<dependency>
  <groupId>com.fasterxml.jackson.dataformat</groupId>
  <artifactId>jackson-dataformat-xml</artifactId>
  <version>2.5.3</version>
</dependency>
```

- **MappingJackson2HttpMessageConverter** → convierte objetos Java desde y hacia JSON (solo si la librería de Jackson2 está presente en el classpath).

```
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.5.3</version>
</dependency>
```

- **MappingJacksonHttpMessageConverter** → convierte objetos Java desde y hacia JSON (sólo si la librería de Jackson está presente en el classpath).
- **AtomFeedHttpMessageConverter** → convierte objetos Java del tipo **Feed** que proporciona la librería Rome desde y hacia feeds Atom, media type **application/atom+xml** (solo si la librería Roma está presente en el classpath).
- **RssChannelHttpMessageConverter** → convierte objetos Java del tipo **Channel** que proporciona la librería Rome desde y hacia feeds RSS (sólo si la librería Roma está presente en el classpath).

Personalizacion de la Pila de HttpMessageConverters

La Pila generada por defecto se puede modificar, para ello en XML se hace

```
<mvc:annotation-driven>
  <mvc:message-converters>
    <bean class=
"org.springframework.http.converter.json.MappingJackson2HttpMessageConverter"/>
  </mvc:message-converters>
</mvc:annotation-driven>
```

Y con Javaconfig

```
public class WebConfig extends WebMvcConfigurerAdapter {
    @Override
    public void configureMessageConverters(List<HttpMessageConverter<?>> converters) {
        messageConverters.add(new MappingJackson2HttpMessageConverter());
        super.configureMessageConverters(converters);
    }
}
```

La clase **RestTemplate** tambien emplea los **HttpMessageConverter** para realizar los marshall, pudiendo establecer la pila de la siguiente manera

```
RestTemplate restTemplate = new RestTemplate();
restTemplate.setMessageConverters(getMessageConverters());
```

Microservicios

Segun [Martin Fowler](#) y [James Lewis](#), los microservicios son pequeños servicios autonomos que se comunican con APIs ligeros, tipicamente con APIs REST.

El concepto de autonomo, indica que el microservicio encierra toda la lógica necesaria para cubrir una funcionalidad completa, desde el API que expone que puede hacer hasta el acceso a la base de datos.

La arquitectura de microservicios, se forma con pequeñas aplicaciones independientes que se ejecutan y despliegan independientemente, comunicandose con la parte cliente o si es necesario entre ellas sincronamente a través de APIs o asincronamente a través de servicios de mensajería.

Ventajas

- Desarrollos pequeños.
- Desarrollo rapido, sin bloqueos.
- Despliegue rapido, ya que no dependerá de otros desarrollos.

NOTE

Las tres características anteriores, cuadran con **Continuous Delivery** (entrega continua).

- Escalado mas granular, permitiendo aprovechar de forma mas eficiente los recursos.
- Aprovechamiento de los puntos fuertes de cada lenguaje de programación, ya que permite el uso de distintas tecnologías para la implementacion de cada microservicio.
- Mejor respuesta a fallos, que falle un modulo, no hará que fallen otros.

Desventajas

- Administracion más compleja, exigen mayor esfuerzo en el despliegue, control del versionado, copatibilidad entre versiones, actualización, monitorización, ...
- Rendimiento afectado por latencia de red.

Ejemplo: Aplicación de puntos por viajes.

Tres servicios independientes

- Consulta de puntos acumulados.
- Consulta de ofertas para consumir los puntos.
- Consulta de viajes planificados que aportan puntos.

Es este caso dichas funcionalidades pueden implementarse como microservicios independientes, dado que ninguno necesita de los demas.

Ejemplo: Orden de compra.

Un servicio que debe comunicarse con multiples servicios.

- Servicio principal de compra que da de alta en el sistema la orden de compra y que debe poner en marcha el mecanismo para que al comprador se le cobre la compra, se le envíe y pueda realizar una traza del estado de su compra.
- Servicio de envio, que gestiona con la empresa de paqueteria que se realice el envio.
- Servicio de actualizacion de stock, que resta del Stock lo que el cliente a comprado.
- Servicio de notificacion al cliente, que indica al cliente que su pedido esta en marcha.

La comunicacion entre estos servicios se podria implementar asincronamente, permitiendo con ello que puedan aparecer otros servicios nuevos, que no afectan al resto, como

- Servicio de puntos, que actualiza los puntos acumulados por el cliente al realizar la compra.

Cualquiera de los servicios de segundo nivel, podrian hacer uso de otros servicios, como por ejemplo el servicio de **actualizacion de stock**, podria hacer uso de

- Servicio de gestion de stock, que revisa si los niveles minimos de stock del producto se han

alcanzado y en ese caso lanza la orden de reabastecimiento.

Incluso del servicio de notificación ya empleado antes para notificar al proveedor de ese pedido nuevo para reabastecer el stock.

Ejemplo: Agencia de viajes.

- Clientes → Servicio que gestiona los datos de los clientes.
- Reservas → Da de alta una reserva de vuelos para un determinado cliente y después consulta el servicio de Vuelos, para validar y actualizar los datos de los vuelos reservados (invocación a un servicio, desde otro servicio, se conoce como orquestación), finalmente a través de JMS envía un evento para que se notifique al cliente el estado de la reserva, con el servicio de notificación (se conoce como coreografía).
- Visados → Permite comprobar los requisitos de Visado para una determinada reserva
- Vuelos → Buscador de vuelos para poder realizar una reserva para un Cliente
- Notificación → Permite enviar un mail al Cliente para confirmarle la Reserva.

NOTE

La arquitectura que permite un mayor desacople entre microservicios es la **Coreografía**, pero no se puede aplicar en todas las ocasiones, dado que en muchos casos se precisa una respuesta inmediata a la consulta, empleando el mismo medio por el que se realizó dicha consulta, esto nos lleva al uso de la **Orquestación**.

Rest

Los servicios REST son servicios basados en recursos, montados sobre HTTP, donde se da significado al Method HTTP.

La palabra REST viene de

- **Representación:** Permite representar los recursos en múltiples formatos, aunque el más habitual es JSON.
- **Estado:** Se centra en el estado del recurso y no en las operaciones que se pueden realizar con él.
- **Transferencia:** Transfiere los recursos al cliente.

Los significados que se dan a los Method HTTP son:

- **POST:** Permite crear un nuevo recurso.
- **GET:** Permite leer/obtener un recurso existente.
- **PUT o PATCH:** Permiten actualizar un recurso existente.
- **DELETE:** Permite borrar un recurso.

Spring MVC, ofrece una anotación **@RestController**, que aúna las anotaciones **@Controller** y **@ResponseBody**, esta última empleada para representar la respuesta directamente con los objetos retornados por los métodos de controlador.

```

@RestController
@RequestMapping(path="/personas")
public class ServicioRestPersonaControlador {

    @RequestMapping(path="/{id}", method= RequestMethod.GET, produces=MediaType
.APPLICATION_JSON_VALUE)
    public Persona getPersona(@PathVariable("id") int id){
        return new Persona(1, "victor", "herrero", 37, "M", 1.85);
    }
}

```

De esta representación se encargan los **HttpMessageConverter**.

Personalizar el Mapping de la entidad

En transformaciones a XML o JSON, de querer personalizar el Mapping de la entidad retornada, se puede hacer empleando las anotaciones de JAXB, como son **@XmlRootElement**, **@XmlElement** o **@XmlAttribute**.

Estado de la petición

Cuando se habla de servicios REST, es importante ofrecer el estado de la petición al cliente, para ello se emplea el código de estado de HTTP.

Para incluir este código en las respuestas, se puede encapsular las entidades retornadas con **ResponseEntity**, el cual es capaz de representar también el código de estado con las constantes de **HttpStatus**

```

@RequestMapping(value="/{id}", method=RequestMethod.GET)
public ResponseEntity<Spittle> spittleById(@PathVariable long id) {
    Spittle spittle = spittleRepository.findOne(id);
    HttpStatus status = spittle != null ? HttpStatus.OK : HttpStatus.NOT_FOUND;
    return new ResponseEntity<Spittle>(spittle, status);
}

```

Localización del recurso

En la creación del recurso, petición POST, se ha de retornar en la cabecera **location** de la respuesta la Url para acceder al recurso que se acaba de generar, siendo estas cabeceras retornadas gracias de nuevo al objeto **ResponseEntity**

```
HttpHeaders headers = new HttpHeaders();
URI locationUri = URI.create("http://localhost:8080/spittr/spittles/" + spittle.getId());
headers.setLocation(locationUri);
ResponseEntity<Spittle> responseEntity = new ResponseEntity<Spittle>(spittle, headers,
HttpStatus.CREATED)
```

Cliente se servicios con RestTemplate

Las operaciones que se pueden realizar con RestTemplate son

- **Delete** → Realiza una petición DELETE HTTP en un recurso en una URL especificada

```
public void deleteSpittle(long id) {
    RestTemplate rest = new RestTemplate();
    rest.delete(URI.create("http://localhost:8080/spittr-api/spittles/" + id));
}
```

- **Exchange** → Ejecuta un método HTTP especificado contra una URL, devolviendo un ResponseEntity que contiene un objeto mapeado del cuerpo de respuesta
- **Execute** → Ejecuta un método HTTP especificado contra una URL, devolviendo un objeto mapeado en el cuerpo de la respuesta.
- **GetForEntity** → Envía una solicitud HTTP GET, devolviendo un ResponseEntity que contiene un objeto mapeado del cuerpo de respuesta

```
public Spittle fetchSpittle(long id) {
    RestTemplate rest = new RestTemplate();
    ResponseEntity<Spittle> response = rest.getForEntity(
"http://localhost:8080/spittr-api/spittles/{id}", Spittle.class, id);
    if(response.getStatusCode() == HttpStatus.NOT_MODIFIED) {
        throw new NotModifiedException();
    }
    return response.getBody();
}
```

- **GetForObject** → Envía una solicitud HTTP GET, devolviendo un objeto asignado desde un cuerpo de respuesta

```
public Spittle[] fetchFacebookProfile(String id) {
    Map<String, String> urlVariables = new HashMap<String, String>();
    urlVariables.put("id", id);
    RestTemplate rest = new RestTemplate();
    return rest.getForObject("http://graph.facebook.com/{spitter}", Profile.class,
urlVariables);
}
```

- **HeadForHeaders** → Envía una solicitud HTTP HEAD, devolviendo los encabezados HTTP para los URL de recursos
- **OptionsForAllow** → Envía una solicitud HTTP OPTIONS, devolviendo el encabezado Allow URL especificada
- **PostForEntity** → Envía datos en el cuerpo de una URL, devolviendo una ResponseEntity que contiene un objeto en el cuerpo de respuesta

```
RestTemplate rest = new RestTemplate();
ResponseEntity<Spitter> response = rest.postForEntity(
"http://localhost:8080/spittr-api/spitters", spitter, Spitter.class);
Spitter spitter = response.getBody();
URI url = response.getHeaders().getLocation();
}
```

- **PostForLocation** → POSTA datos en una URL, devolviendo la URL del recurso recién creado

```
public String postSpitter(Spitter spitter) {
    RestTemplate rest = new RestTemplate();
    return rest.postForLocation("http://localhost:8080/spittr-api/spitters", spitter)
.toString();
}
```

- **PostForObject** → POSTA datos en una URL, devolviendo un objeto mapeado de la respuesta cuerpo

```
public Spitter postSpitterForObject(Spitter spitter) {
    RestTemplate rest = new RestTemplate();
    return rest.postForObject("http://localhost:8080/spittr-api/spitters", spitter,
Spitter.class);
}
```

- **Put** → PUT pone los datos del recurso en la URL especificada

```
public void updateSpittle(Spittle spittle) throws SpitterException {
    RestTemplate rest = new RestTemplate();
    String url = "http://localhost:8080/spittr-api/spittles/" + spittle.getId();
    rest.put(URI.create(url), spittle);
}
```

Spring WebService

Framework que permite publicar Beans de Spring como Servicios Web SOAP.

La más nueva especificación en JEE para los Servicios Web SOAP, es JAX-WS, proporciona anotaciones como `@WebService` y `@WebMethod`.

Servicio Standalone

La JRE a partir de 1.6, proporciona una implementación de JAX-WS, la cual se puede emplear con Spring para publicar un servicio, para ello, se debe crear un Bean de Spring.

```
<bean id="ServicioEndpoint" class="com.curso.spring.Servicio">
</bean>
```

Anotando la clase con `@WebService` y los métodos expuestos con `@WebMethod`

```
@WebService(serviceName="servicio")
public class Servicio {
    @WebMethod
    public Persona metodo(Persona persona){
        //Implementacion del servicio
    }
}
```

Y declarar un bean de Spring de tipo **SimpleJaxWsServiceExporter** que buscará todos aquellos Bean de Spring anotados con las etiquetas del estándar JAX-WS (`@WebService`) y los publicará como EndPoint, por defecto en <http://localhost:8080>, aunque a través del objeto **SimpleJaxWsServiceExporter** se tiene control sobre las características del servicio.

```
<bean class="org.springframework.remoting.jaxws.SimpleJaxWsServiceExporter">
    <property name="baseAddress" value="http://localhost:8080/" />
</bean>
```

Basta con cargar el contexto de Spring para que se instancie el **SimpleJaxWsServiceExporter**, y el servicio quedará publicado.


```
public static void main(String[] args) {
    ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(
        "ApplicationContext.xml");
}
```

Cliente Servicio

Para consumir el servicio el propio API proporciona una implementación de un Proxy **JaxWsPortProxyFactoryBean**, el cual absorbe toda la problemática de conectar al servicio, proporcionando acceso a las funcionalidades del servicio a través de la interface que cumple el servicio.

```
<bean id="servicio"
    class="org.springframework.remoting.jaxws.JaxWsPortProxyFactoryBean">
    <property name="serviceInterface" value="com.curso.spring.IServicio" />
    <property name="wsdlDocumentUrl" value="http://localhost:8080/servicio?WSDL" />
    <property name="namespaceUri" value="http://spring.curso.com/" />
    <property name="serviceName" value="servicio" />
    <property name="portName" value="ServicioPort" />
</bean>
```

Hay que indicar:

- La interface del servicio.
- El WSDL del servicio.
- El namespace XML empleado en el servicio
- El nombre del servicio.
- El nombre del port donde se publica.

Este Bean se puede inyectar a cualquier otro Bean de Spring como un **com.curso.spring.IServicio**

```
<bean class="com.curso.spring.Cliente">
    <property name="servicio" ref="servicio"/>
</bean>
```

Mapping de entidades

Para la creación del esquema que define los datos a intercambiar con el servicio, se emplea JAXB, por lo que se podría personalizar dicho mapeo con las anotaciones de JAXB, como **@XmlRootElement** o **@XmlElement**

```

@XmlRootElement
public class Persona {
    private int id;
    private String nombre;
    private Date fecha;

    @XmlElement(name="identificador")
    public int getId() {
        return id;
    }

    ....
}

```

Despliegue en contenedor web

Para el despliegue en un contenedor web, se precisa de una nueva dependencia, se puede encontrar mas información [aquí](#)

Spring Security

La seguridad en Spring:

- Basada en otorgar el acceso.
- Jerárquica y perimetral. Aplica niveles.
- Transportable.

NOTE

Perimetral = estas dentro o no. Jerárquica = se pueden aplicar niveles de acceso a los contenidos.

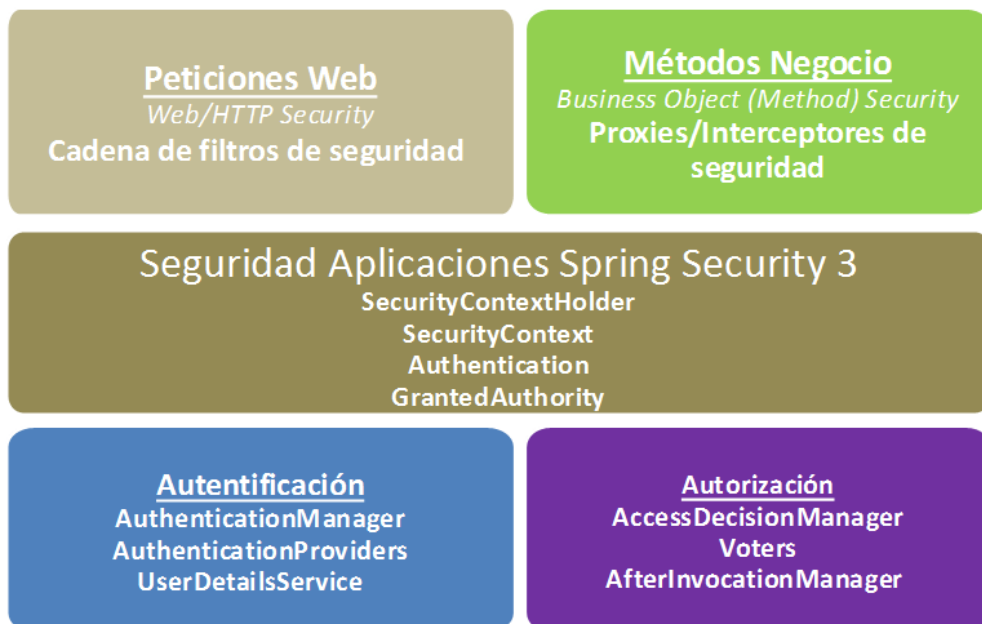
La seguridad en JEE:

- Basada en restricciones.
- Perimetral.
- Difícil migrar.

NOTE

Restricciones = Todo es accesible hasta que se restringe el acceso. No es estandar dentro de los contenedores JEE, no es facil migrar.

Arquitectura



NOTE

Acceder a la app de fbi.war sin seguridad, se ve que accede hasta el fondo, a todas las funcionalidades sin restriccion.

localhost:8081/fbi → "mostrar expedientes" → "clasificar" → "desclasificar" → "mostrar"

Dependencias con Maven

Se han de añadir las siguientes dependencias al proyecto.

```
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-taglibs</artifactId>
  <version>3.2.4.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-config</artifactId>
  <version>3.2.4.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-crypto</artifactId>
  <version>3.2.4.RELEASE</version>
</dependency>
```

Filtro de seguridad

La seguridad con Spring, se basa en una clase de Spring **FilterChainProxy**, este filtro no será más que un Bean de Spring.

```
<bean id="springSecurityFilterChain" class=
"org.springframework.security.web.FilterChainProxy">
```

NOTE

El Bean de Spring de tipo **FilterChainProxy**, no es necesario definirlo directamente, ya que es una de las configuraciones por defecto que se añaden al activar la seguridad en Spring.

Sobre este Bean, delegará un Filtro Web especial, el **DelegatingFilterProxy**, que como su nombre indica, delega en el contexto de Spring lo que intercepta.

Con XML

```
<filter>
  <display-name>springSecurityFilterChain</display-name>
  <filter-name>springSecurityFilterChain</filter-name>
  <filter-class>
    org.springframework.web.filter.DelegatingFilterProxy
  </filter-class>
</filter>
<filter-mapping>
  <filter-name>springSecurityFilterChain</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Con Java Config

```
public class AppInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {
        // ...
        servletContext.addFilter("springSecurityFilterChain",
            new DelegatingFilterProxy("springSecurityFilterChain"))
            .addMappingForUrlPatterns(null, false, "/*");
        // ...
    }
}
```

Contexto de Seguridad

Se ha de añadir la anotación **@EnableWebSecurity** a una clase con **@Configuration** para que se genere el objeto **WebSecurityConfigurer**, que tiene la configuración por defecto de Spring Security.

Esta configuración puede ser sobrescrita haciendo a su vez extender la clase **@Configuration** de **WebSecurityConfigurerAdapter**.

```
@Configuration
@EnableWebSecurity
public class ConfiguracionSpringSecurity extends WebSecurityConfigurerAdapter {

}
```

AuthenticationManagerBuilder

Permite definir de donde se han de obtener los usuarios y roles que se emplearán en la aplicación, para ello sobrescribir el método correspondiente de la clase **WebSecurityConfigurerAdapter**.

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.inMemoryAuthentication().withUser("Fernando").password("1234").roles("AGENTE");
    auth.inMemoryAuthentication().withUser("Mulder").password("fox").roles("AGENTE_ESPECIAL");
    auth.inMemoryAuthentication().withUser("Scully").password("dana").roles("AGENTE_ESPECIAL");
    auth.inMemoryAuthentication().withUser("Skinner").password("walter").roles("DIRECTOR");
}
```

NOTE

El objeto **AuthenticationManagerBuilder**, tiene un método **jdbcAuthentication()** que permite definir la conexión contra una base de datos para obtener los usuarios y roles.

Proteccion de recursos

Permite configurar la seguridad Web sobre las peticiones http.

Patrón de recursos protegido y rol que puede acceder.

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
            .antMatchers("/**").access("hasAnyRole('AGENTE_ESPECIAL', 'DIRECTOR')");
}
```

NOTE

En este caso, está limitado el acceso a cualquier recurso a aquellos usuarios que tienen como ROL = **ROLE_AGENTE_ESPECIAL**, además de que el proceso de Login, se realiza con formulario.

Para excluir recursos de la protección.

```
http
    .authorizeRequests()
        .antMatchers("/paginas/*").permitAll()
        .antMatchers("/css/*").permitAll()
        .antMatchers("/imagenes/*").permitAll();
```

Login

Para definir un proceso de Login a través de formulario con una pagina de login personalizada.

```
http
    .formLogin()
        .loginPage("/paginas/nuestro-login.jsp")
        .failureUrl("/paginas/nuestro-login.jsp?login_error");
```

NOTE

Dado que cuando se intenta acceder a un recurso y este está asegurado, se nos redirigirá a la página de login, para conocer a que recurso se queria realmente acceder, Spring crea una **Session** temporal donde almacena la URL.

Logout

Para activar el logout indicando la url para realizar el logout, definir la pagina a la que se redirecciona una vez realizado el logout y el nombre de las cookies que se han de borrar en el proceso.

```
http
    .logout()
        .logoutUrl("/logout")
        .invalidateHttpSession(true)
        .logoutSuccessUrl("/paginas/desconectado.jsp")
        .deleteCookies("JSESSIONID");
```

Por tanto para realizar el logout, basta con invocar la url **/logout**

```
<a href="<c:url value='/logout'/>">desconectar</a>
```

CSRF

El CSRF (Cross-site request forgery) o control de accesos desde sitio externos, permite mediante la adición de una huella aleatoria en las transacciones entre servidor y cliente, controlar que no se acceda al servidor desde otro sitio que no sea la propia aplicación.

Consiste en añadir un campo oculto para guardar el token en todos los formularios que utilicen el

método POST:

```
<input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}"/>
```

Este Token, también se puede añadir con la librería de etiquetas de Spring security:

```
<sec:csrfInput />
```

Este control se puede desactivar.

```
http
    .csrf().disable();
```

UserDetailsService

Permite personalizar la forma en la que se realiza la autenticación.

Spring proporciona implementaciones de referencia como **InMemoryUserDetailsManager** o **JdbcUserDetailsManager**

```
@Override
public void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.userDetailsService(userDetailsService());
}

public UserDetailsService userDetailsService(){
    Properties usuarios = new Properties();
    usuarios.put("Fernando", "1234,ROLE_AGENTE,enabled");
    usuarios.put("Mulder" , "fox,ROLE_AGENTE_ESPECIAL,enabled");
    usuarios.put("Scully" , "dana,ROLE_AGENTE_ESPECIAL,enabled");
    usuarios.put("Skinner" , "walter,ROLE_DIRECTOR,enabled");

    return new InMemoryUserDetailsManager(usuarios);
}
```

Encriptación

Se trata de posibilitar el medio para que se almacenen las contraseñas encriptadas, pero que se sigan pudiendo resolver, para ello se ha de definir un **Encoder** y asociarlo al **AuthenticationManagerBuilder**.

```

@Override
public void configure(AuthenticationManagerBuilder auth) throws Exception {
    PasswordEncoder encoder = new BCryptPasswordEncoder();
    auth.userDetailsService(userDetailsService()).passwordEncoder(encoder);
}

public UserDetailsService userDetailsService(){
    Properties usuarios = new Properties();
    usuarios.put("Fernando",
"$2a$10$SMPYti17Hs2.cV7nrMjrM.dRAkuoLdYM8NdVrF.GeHfs/MrzcQ/zi,ROLE_AGENTE,enabled");
    usuarios.put("Mulder" ,
"$2a$10$M2JRRHUHTfv4uMR4NWmCLebk1r9DyWSwCMZmuq4LKbImOkfhGFAIa,ROLE_AGENTE_ESPECIAL,enabled");
    usuarios.put("Scully" ,
"$2a$10$cbF5xp0grC0GcI6jZvPhA.asgmILATW1hNbM2MEqGJEFnRhhQd3ba,ROLE_AGENTE_ESPECIAL,enabled");
    usuarios.put("Skinner" ,
"$2a$10$ZFtPIULMcxPe3r/5VunbVujMD7Lw8hkqAmJlxmK5Y1TK3L1bf8ULG,ROLE_DIRECTOR,enabled");

    return new InMemoryUserDetailsManager(usuarios);
}

```

Adicionalmente se puede realizar dicha configuración aprovechando una características de las clases anotadas con **@Configuration**, donde se ejecutan todos los métodos anotados con **@Autowired**, recibiendo la inyección de los parametros definidos.


```

@Bean
public PasswordEncoder passwordEncoder(){
    PasswordEncoder encoder = new BCryptPasswordEncoder();
    return encoder;
}

@Autowired
public void configureGlobalSecurity(AuthenticationManagerBuilder auth, PasswordEncoder
pe) throws Exception {
    auth.userDetailsService(userDetailsService()).passwordEncoder(pe);
}

public UserDetailsService userDetailsService(){
    Properties usuarios = new Properties();
    usuarios.put("Fernando",
"$2a$10$SMPYt1l7Hs2.cV7nrMjrM.dRAkuoLdYM8NdVrF.GeHfs/MrzcQ/zi,ROLE_AGENTE,enabled");
    usuarios.put("Mulder" ,
"$2a$10$M2JRRHUHTfv4uMR4NWmCLebk1r9DyWSwCMZmuq4LKbImOkfhGFAIa,ROLE_AGENTE_ESPECIAL,ena
bled");
    usuarios.put("Scully" ,
"$2a$10$cbF5xp0grC0GcI6jZvPhA.asgmILATW1hNbM2MEqGJEFnRhhQd3ba,ROLE_AGENTE_ESPECIAL,ena
bled");
    usuarios.put("Skinner" ,
"$2a$10$ZFtPIULMcxPe3r/5VunbVujMD7Lw8hkqAmJLxmK5Y1TK3L1bf8ULG,ROLE_DIRECTOR,enabled");

    return new InMemoryUserDetailsManager(usuarios);
}

```

NOTE

Para la configuración anterior, no será necesario definir el método **configure(AuthenticationManagerBuilder auth)**, ya que puede ser sustituido por este otro.

Remember Me

Funcionalidad que permite incluir una **Cookie** para que la aplicación no pida al usuario que realice el proceso de **login**, recordándolo.

```

http
    .rememberMe()
        .rememberMeParameter("remember-me-param")
        .rememberMeCookieName("my-remember-me")
        .tokenValiditySeconds(86400);

```

Seguridad en la capa transporte - HTTPS

Se puede indicar a Spring que peticiones necesitan de un canal seguro (https), así como establecer

redirecciones automaticas entre puertos.

```
http
    .requiresChannel()
        .anyRequest().requiresSecure()
    .and()
        .portMapper()
            .http(8080).mapTo(8443);
```

Para activar HTTPS, se ha de realizar configuraciones en el servidor, en el caso de un tomcat, se haria algo así.

*En el fichero de configuración **server.xml** copiar*

```
<Connector SSLEnabled="true" acceptCount="100"
    connectionTimeout="20000" executor="tomcatThreadPool"
    keyAlias="tcserver" keystoreFile="${catalina.base}/conf/tcserver.keystore"
    keystorePass="changeme"
    maxKeepAliveRequests="15" port="8443" protocol="HTTP/1.1"
    redirectPort="8443" scheme="https" secure="true"/>
```

Sesiones concurrentes

*En el directorio **\${catalina.base}/conf** copiar un fichero de claves **tcserver.keystore** con los datos definidos en la configuración anterior.*

Se puede controlar la concurrencia de sesiones, permitiendo controlar que varios navegadores accedan con el mismo usuario. El procedimiento crea un contador, que cuando cumple con el numero especificado, no deja crear nuevas conexiones.

```
http
    .sessionManagement()
        .maximumSessions(1)
        .maxSessionsPreventsLogin(true);
```

Tiene un problema cuando se cierra el navegador y no se da a desconectar, ya que no se ejecuta la actualización del contador, luego cuando se llegue al máximo ya solo se podría acceder desde los navegadores que consiguieron el acceso.

Para solventar este problema, se dispone de un Listener que se encarga de esta eventualidad.

```
public class AppInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {
        servletContext.addListener(new HttpSessionEventPublisher());
    }
}
```

SessionFixation

Es un efecto que se puede producir en las aplicaciones web, que consiste en la sobrescritura de los roles en el objeto sesión.

La idea basicamente es que al realizarse el proceso de **login** sobre una sesión ya creada (repetir el login), sino se crea de nuevo un objeto sesión con su correspondiente identificador de sesión (JSESSIONID), es decir se recicla el objeto sesión, se puede producir que existan dos usuarios con la misma sesión, pero con los datos de roles del segundo usuario, que puede tener mas permisos.

Spring por defecto lo contempla.

```
http
    .sessionManagement()
    .sessionFixation()
    .migrateSession();
```

A mayores, se puede proporcionar una migración de los datos de la sesión antigua a la nueva, para poder seguir manteniendo información aunque se produzca el cambio de usuario.

Libreria de etiquetas

Para añadir las etiquetas propias de la seguridad se ha de incluir la cabecera

```
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags"%>
```

Permite el acceso al objeto de autenticación para

1. Controlar que partes de la **View** se van a renderizar

```
<sec:authorize access="hasRole('ROLE_DIRECTOR')">
    <a href="<c:url value='/expedientesx/clasificar?id=${expediente.id}'/>">
clasificar</a>
    <a href="<c:url value='/expedientesx/desclasificar?id=${expediente.id}'/>">
>desclasificar</a>
</sec:authorize>
```

NOTE

Este tipo de seguridad, unicamente protege que en el uso normal de la aplicación un usuario pueda ver un enlace o información, a la que no tiene acceso, para evitar errores de acceso, pero no protege la funcionalidad (negocio) en si.

1. Mostrar información del usuario

```
<sec:authentication property="principal.username" />
```

Expresiones SpEL

Para crear las expresiones que se han ido empleando, el lenguaje de expresiones de Spring (SpEL), ofrece una serie de comandos

- `hasRole(role)`
- `hasAnyRole([role1,role2])`
- `permitAll`
- `denyAll`
- `isAnonymous()`
- `isAuthenticated()`

Seguridad de métodos

Para activar la seguridad de los métodos, se ha de incluir la anotación `@EnableGlobalMethodSecurity` en la clase `@Configuration`

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity
public class ConfiguracionSpringSecurity extends WebSecurityConfigurerAdapter {}
```

Tambien es recomendable definir una página de error cuando se produzcan intentos de acceso no autorizados

```
http
    .exceptionHandling()
        .accessDeniedPage("/paginas/acceso-denegado.jsp");
```

Spring Security, es compatible con anotaciones propias y de la especificación java JSR-250. Las propias se dividen en dos grupos

- `@Secured` que se activa con `securedEnabled`
- `prepost` que se activa con `prePostEnabled`

Para activar las de JSR-250 habrá que activar **jsr250Enabled**.

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(securedEnabled = true, jsr250Enabled=true, prePostEnabled
= true)
public class ConfiguracionSpringSecurity extends WebSecurityConfigurerAdapter {}
```

Las anotaciones de Spring son

- **@Secured**

```
@Secured("ROLE_AGENTE_ESPECIAL,ROLE_DIRECTOR")
void clasificar(Expediente expediente);
```

- **@PreAuthorize**

```
@PreAuthorize("hasRole('ROLE_DIRECTOR') or #expediente.investigador ==
authentication.name")
void desclasificar(Expediente expediente);
```

- **@PostAuthorize**

```
@PostAuthorize("hasRole('ROLE_DIRECTOR') or returnObject.investigador ==
authentication.name")
Expediente mostrar(Long id);
```

- **@PreFilter**

- **@PostFilter**

```
@PostFilter("(hasRole('ROLE_AGENTE') and not filterObject.clasificado) " +
"or (hasAnyRole('ROLE_AGENTE_ESPECIAL','ROLE_DIRECTOR') and not
filterObject.informe.contains(principal.username))")
List<Expediente> listarTodos();
```

Las anotaciones de JSR-250 son

- **@RolesAllowed**

```
@RolesAllowed("ROLE_AGENTE_ESPECIAL,ROLE_DIRECTOR")
void desclasificar(Expediente expediente);
```

- **PermitAll**

- DenyAll
- RunAs

Spring Test

Framework, que proporciona un runner para poder ejecutar Test que carguen un contexto de spring

La dependencias de Maven

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-test</artifactId>
  <version>4.3.5.RELEASE</version>
</dependency>
```

Para emplearlo, se han de anotar las clases de Test con

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes=Configuracion.class)
```

Donde en la clase **Configuracion** se definirá el contexto de spring con los beans a probar, de ser una prueba unitaria, solo necesitaremos definir el **SUT**.

Al test se le inyectará el **SUT** con **@Autowired**

```
@Autowired
private Servicio sut;
```

Mocks

Si es una prueba unitaria sobre un componente que tiene dependencias, será necesario cubrir esas dependencias con objetos **Mock** que describan la funcionalidad esperada por el componente a probar en el entorno de pruebas, para ello se puede emplear por ejemplo **Mockito**

La dependencias de Maven

```
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-all</artifactId>
  <version>1.10.19</version>
</dependency>
```

Este framework, permite definir objetos **Mock** con la anotación **@Mock**

```

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = Configuracion.class)
public class TestCliente {
    @Mock
    private Pedido pedido;
}

```

La cual se activa bien ejecutando con el Runner de Mockito **MockitoJUnitRunner** o bien ejecutando

```

@Before
public void setup() {
    MockitoAnnotations.initMocks(this);
}

```

Este último será el caso para **Spring-Test**, ya que el Runner debe ser el de Spring.

Para poder emplear el **Mock**, habrá que relacionar los Bean, cosa que todavía no ha ocurrido, ya que cada uno lo genera un contenedor, para ello, se ha de indicar al **SUT**, que reciba las dependencias de **Mocks** de Mockito, para ello se ha de anotar con **@InjectMocks**

```

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = Configuracion.class)
public class TestCliente {

    @Mock
    private Pedido pedido;

    @Autowired
    @InjectMocks
    private Cliente cliente; //La tipologia Cliente tiene una dependencia con un bean
    Pedido
}

```

Una vez establecida la dependencia, solo falta definir los comportamientos de los **Mocks** ante el entorno de pruebas del **SUT**.

MVC Mocks

El framework proporciona una clase **MockMvc** que permite comprobar el comportamiento de los Controladores simulando su ejecución en un contenedor web.

```

Runner.class)
@ContextConfiguration(classes = Configuracion.class)
public class TestControlador {

    private MockMvc mockMvc;

    @Autowired
    private Controlador sut; //Bean de Spring de tipo @Controller

    @Before
    public void init() {
        mockMvc = MockMvcBuilders.standaloneSetup(controlador).build();
    }
}

```

Una vez inicializado el **MockMvc** y asociado al **SUT**, y dada la siguiente implementación del controlador

```

@Controller
public class Controlador {
    @RequestMapping(value = "/", method = RequestMethod.GET)
    public String metodo(@RequestParam String param, Model model) {
        List<String> models = Arrays.asList(new String[]{"dato", "otro"});
        model.addAttribute("datos", models);
        return "resultado";
    }
}

```

Se relizan las pruebas, para lo cual el API ofrece métodos estaticos en las clases **org.springframework.test.web.servlet.request.MockMvcRequestBuilders** y **org.springframework.test.web.servlet.result.MockMvcResultMatchers**


```

@Test
public void codigoDeRespuestaCorrecto() throws Exception {
    mockMvc.perform(get(URL + "?param=victor")).andExpect(status().isOk());
}

@Test
public void contenidoRespuestaCorrecto() throws Exception {
    mockMvc.perform(get(URL + "?param=info")).andExpect(model().attributeExists("
datos"));
}

@Test
public void methodNoValido() throws Exception {
    mockMvc.perform(put(URL)).andExpect(status().isMethodNotAllowed());
}

@Test
public void peticionMalConstruida() throws Exception {
    mockMvc.perform(get(URL)).andExpect(status().isBadRequest());
}

```

Spring Boot

Introduccion

Framework orientado a la construcción/configuración de proyectos de la familia Spring basado en **Convention-Over-Configuration**, por lo que minimiza la cantidad de código de configuración de las aplicaciones.

Afecta principalmente a dos aspectos de los proyectos

- **Configuracion de dependencias:** Proporcionado por **Starters** Aunque sigue empleando Maven o Gradle para configurar las dependencias del proyecto, abstrae de las versiones de los APIs y lo que es más importante de las versiones compatibles de unos APIs con otros, dado que proporciona un conjunto de librerías que ya están probadas trabajando juntas.
- **Configuracion de los APIs:** Cada API de Spring que se incluye, ya tendrá una preconfiguración por defecto, la cual si se desea se podrá cambiar, además de incluir elementos tan comunes en los desarrollos como un contenedor de servlets embebido ya configurado, estas preconfiguraciones se establecen simplemente por el hecho de que la librería esté en el classpath, como un Datasource de una base de datos, JdbcTemplate, Java Persistence API (JPA), Thymeleaf templates, Spring Security o Spring MVC.

Además proporciona otras herramientas como

- La consola Spring Boot CLI
- Actuator

Instalación de Spring Boot CLI

Permite la creación de aplicaciones Spring, de forma poco convencional, centrandose unicamente en el código, la consola se encarga de resolver dependencias y configurar el entorno de ejecución.

Emplea scripts de Groovy.

Para descargar la distribución pinchar [aquí](#)

Descomprimir y añadir a la variable entorno PATH la ruta `$SPRING_BOOT_CLI_HOME/bin`

Se puede acceder a la consola en modo ayuda (completion), con lo que se obtiene ayuda para escribir los comandos con TAB, para ello se introduce

```
> spring shell
```

Una vez en la consola se puede acceder a varios comandos uno de ellos es el de la ayuda general **help**

```
Spring-CLI# help
```

O la ayuda de alguno de los comandos

```
Spring-CLI# help init
```

Con Spring Boot se puede crear un proyecto MVC tan rapido como definir la siguiente clase Groovy **HelloController.groovy**

```
@RestController
class HelloController {

    @RequestMapping("/")
    def hello() {
        return "Hello World"
    }

}
```

Y ejecutar desde la consola Spring Boot CLI

```
> spring run HelloController.groovy
```

La consola se encarga de resolver las dependencias, de compilar y de establecer las configuraciones por defecto para una aplicacion Web MVC, en el web.xml, ... por lo que una vez ejecutado el comando de la consola, al abrir el navegador con la url <http://localhost:8080> se accede a la

aplicación.

Si se dispone de mas de un fichero **groovy**, se puede lanzar todos los que se quiera con el comando

```
> spring run *.groovy
```

El directorio sobre el que se ejecuta el comando es considerado el root del classpath, por lo que si se añade un fichero **application.properties**, este permite configurar el proyecto.

Si se quiere añadir motores de plantillas, se deberá incluir la dependencia, lo cual se puede hacer con **Grab**, por ejemplo para añadir **Thymeleaf**

```
@Grab(group='org.springframework.boot', module='spring-boot-starter-thymeleaf',  
version='1.5.7.RELEASE')  
  
@Controller  
class Application {  
    @RequestMapping("/")  
    public String greeting() {  
        return "greeting"  
    }  
}
```

Y definir las plantillas en la carpeta **templates**, en este caso **templates/greeting.html**

Si se desea contenido estatico, este se debe poner en la carpeta **resources** o **static**

Creación e implementación de una aplicación

Lo primero a resolver al crear una aplicación son las dependencias, para ellos Spring Boot ofrece el siguiente mecanismo basando en la herencia del POM.

```

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

  ...

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.4.2.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>

  ...

</project>

```

De no poderse establecer dicha herencia, por heredar de otro proyecto, se ofrece la posibilidad de añadir la siguiente dependencia.

```

<project>

  ...

  <dependencyManagement>
    <dependencies>
      <dependency>
        <!-- Import dependency management from Spring Boot -->
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-dependencies</artifactId>
        <version>1.4.2.RELEASE</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>

  ...

</project>

```

Esta dependencia permite a Spring Boot hacer el trabajo sucio para manejar el ciclo de vida de un proyecto Spring normal, pero normalmente se precisarán otras dependencias, para esto Spring Boot ofrece los **Starters**, por ejemplo esta sería la dependencia para un proyecto Web MVC

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
</dependencies>
```

Una vez solventadas las dependencias, habrá que configurar el proyecto, ya hemos mencionado que la configuración quedará muy reducida, en este caso únicamente necesitamos definir una clase anotada con **@SpringBootApplication**

```
@SpringBootApplication
public class HolaMundoApplication {
    ...
}
```

Esta anotación en realidad es la suma de otras tres:

- @Configuration → Se designa a la clase como un posible origen de definiciones de Bean.
- @ComponentScan → Se indica que se buscarán otras clases con anotaciones que definan componentes de Spring como @Controller
- @EnableAutoConfiguration → Es la que incluye toda la configuración por defecto para los distintos APIs seleccionados.

Con esto ya se tendría el proyecto preparado para incluir únicamente el código de aplicación necesario, por ejemplo un Controller de Spring MVC

```
@Controller
public class HolaMundoController {
    @RequestMapping("/")
    @ResponseBody
    public String holaMundo() {
        return "Hola Mundo!!!!";
    }
}
```

Una vez finalizada la aplicación, se podría ejecutar de varias formas

- Como jar autoejecutable, para lo que habrá que definir un método **Main** que invoque **SpringApplication.run()**

```
@SpringBootApplication
public class HolaMundoApplication {
    public static void main(String[] args) {
        SpringApplication.run(HolaMundoApplication.class, args);
    }
}
```

Y posteriormente ejecutandolo con

- Una tarea de Maven

```
mvn spring-boot:run
```

- Una tarea de Gradle

```
gradle bootRun
```

- O como jar autoejecutable, generando primero el jar

Con Maven

```
mvn package
```

O Gradle

```
gradle build
```

Y ejecutando desde la linea de comandos

```
java -jar HolaMundo-0.0.1-SNAPSHOT.jar
```

- O desplegando como WAR en un contenedor web, para lo cual hay que añadir el plugin de WAR
 - En Maven, con cambiar el package bastará

```
<packaging>war</packaging>
```

- En Gradle alicando el plugin de WAR y cambiando la configuracion JAR por la WAR

```

apply plugin: 'war'

war {
    baseName = 'HolaMundo'
    version = '0.0.1-SNAPSHOT'
}

```

En estos casos, dado que no se ha generado el **web.xml**, es necesario realizar dicha inicialización, para ello Spring Boot ofrece la clase **org.springframework.boot.web.support.SpringBootServletInitializer**

```

public class HolaMundoServletInitializer extends SpringBootServletInitializer {
    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder builder) {
        return builder.sources(HolaMundoApplication.class);
    }
}

```

Uso de plantillas

Los proyectos **Spring Boot Web** vienen configurados para emplear plantillas, basta con añadir el starter del motor deseado y definir las plantillas en la carpeta **src/main/resources/templates**.

Algunos de los motores a emplear son Thymeleaf, freemarker, velocity, jsp, ...

Thymeleaf

Motor de plantillas que se basa en la instrumentalización de **html** con atributos obtenidos del esquema **th**

```

<html xmlns:th="http://www.thymeleaf.org"></html>

```

Para añadir esta característica al proyecto, se añade la dependencia de Maven

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>

```

Por defecto cualquier **String** retornado por un **Controlador** será considerado el nombre de un **html** instrumentalizado con **thymeleaf** que se ha de encontrar en la carpeta **/src/main/resources/templates**

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="ISO-8859-1"></meta>
  <title>Insert title here</title>
</head>
<body>
  <span th:text="{mensaje}"></span>
  <span th:text="#"></span>
</body>
</html>

```

NOTE

No es necesario indicar el espacio de nombres en el html

JSP

Para poder emplear **JSP** en lugar de **Thymeleaf**, basta con añadir las siguientes propiedades para indicar donde se encuentran las plantillas de JSP.

```

spring.mvc.view.prefix: /
spring.mvc.view.suffix: .jsp

```

Recursos estaticos

Si se desean publicar recursos estaticos (html, js, css, ...), se pueden incluir en los proyectos en las rutas:

- **src/main/resources/META-INF/resources**
- **src/main/resources/resources**
- **src/main/resources/static**
- **src/main/resources/public**

Siendo el descrito el orden de inspeccion.

Webjars

Desde hace algun tiempo se encuentran disponibles como dependencias de Maven las distribuciones de algunos frameworks javascript bajo el groupid **org.webjars**, pudiendo añadir dichas dependencias a los proyectos para poder gestionar con herramientas de construccion como Maven o Gradle tambien las versiones de los frameworks javascript.

Estos artefactos tienen incluido los ficheros js, en la carpeta **/META-INF/resources/webjars/<artifactId>/<version>**, con lo que las dependencias hacia los ficheros javascript de los framework añadidos con Maven será **webjars/<artifactId>/<version>/<artifactId>.min.js**


```
<html>
<head>
  <script src="webjars/jquery/2.0.3/jquery.min.js"></script>
  ...
```

Recolección de métricas

El API de Actuator, permite recoger información del contexto de Spring en ejecución, como

- Qué beans se han configurado en el contexto de Spring.
- Qué configuraciones automáticas se han establecido con Spring Boot.
- Qué variables de entorno, propiedades del sistema, argumentos de la línea de comandos están disponibles para la aplicación.
- Estado actual de los subprocesos
- Rastreo de solicitudes HTTP recientes gestionadas por la aplicación
- Métricas relacionadas con el uso de memoria, recolección de basura, solicitudes web, y uso de fuentes de datos.

Estas metricas se exponen via Web o via shell.

Para activarlo, es necesario incluir una dependencia

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Y a partir de ahí, bajo la aplicación desplegada, se encuentran los path con la info, que retornan JSON

- Estado

<http://localhost:8080/ListadoDeTareas/health>

- Mapeos de URL

<http://localhost:8080/ListadoDeTareas/mappings>

- Descarga de estado de la memoria de la JVM

<http://localhost:8080/ListadoDeTareas/heapdump>

- Beans de la aplicación

<http://localhost:8080/ListadoDeTareas/beans>

Se pueden configurar las funcionalidades para que sean privadas, modificando la propiedad **sensitive** del endpoint

```
endpoints:  
  info:  
    sensitive: true
```

Si son privadas, se necesitará configurar **Spring Security** para definir el origen de la autenticación.

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-security</artifactId>  
</dependency>
```

Una vez añadido, se han de configurar las siguientes propiedades

```
security.user.name=admin  
  
security.user.password=secret  
  
security.user.role=SUPERUSER  
  
management.security.role=SUPERUSER
```

Para desactivar la seguridad

```
management.security.enabled=false
```

Endpoint Custom

Se pueden añadir nuevos EndPoints a la aplicación para que muestren algún tipo de información, para ello basta definir un Bean de Spring que extienda la clase **AbstractEndPoint**

```

@Component
public class ListEndpoints extends AbstractEndpoint<List<Endpoint>> {

    private List<Endpoint> endpoints;

    @Autowired
    public ListEndpoints(List<Endpoint> endpoints) {
        super("listEndpoints");
        this.endpoints = endpoints;
    }

    public List<Endpoint> invoke() {
        return this.endpoints;
    }
}

```

TIP

Solo esta implementacion, puede dar error, por encontrar valores en los Bean a Null, y el parser de Jackson no aceptarlo, para solventarlo, se puede definir en el application.properties la propiedad **spring.jackson.serialization.FAIL_ON_EMPTY_BEANS** a **false**

Uso de Java con start.spring.io

Es uno de los modos de emplear el API de **Spring Initializr**, al que tambien se tiene acceso desde

- Spring Tool Suite
- IntelliJ IDEA
- Spring Boot CLI

Es una herramienta que permite crear estructuras de proyectos de forma rapida, a través de plantillas.

Desde la pagina start.spring.io se puede generar una plantilla de proyecto.

Generate a Maven Project ▾ with Spring Boot 1.4.2 ▾

Project Metadata

Artifact coordinates

Group

com.example

Artifact

demo

Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies

Web, Security, JPA, Actuator, Devtools...

Selected Dependencies

Generate Project alt + ⌘

Don't know what to look for? Want more options? [Switch to the full version.](#)

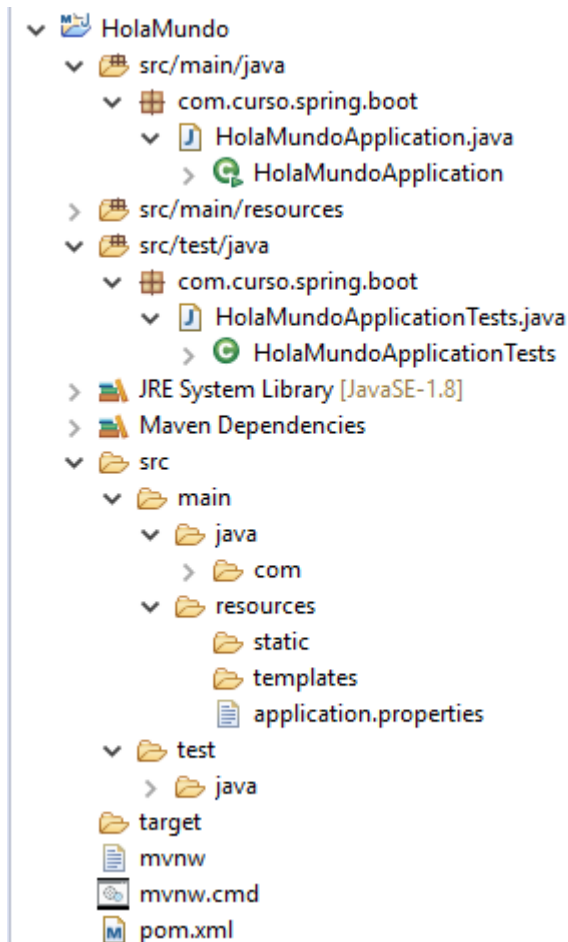
Lo que se ha de proporcionar es

- Tipo de proyecto (Maven o Gradle)
- Versión de Spring Boot
- groupId
- ArtifactId
- Dependencias

Existe una vista avanzada donde se pueden indicar otros parametros como

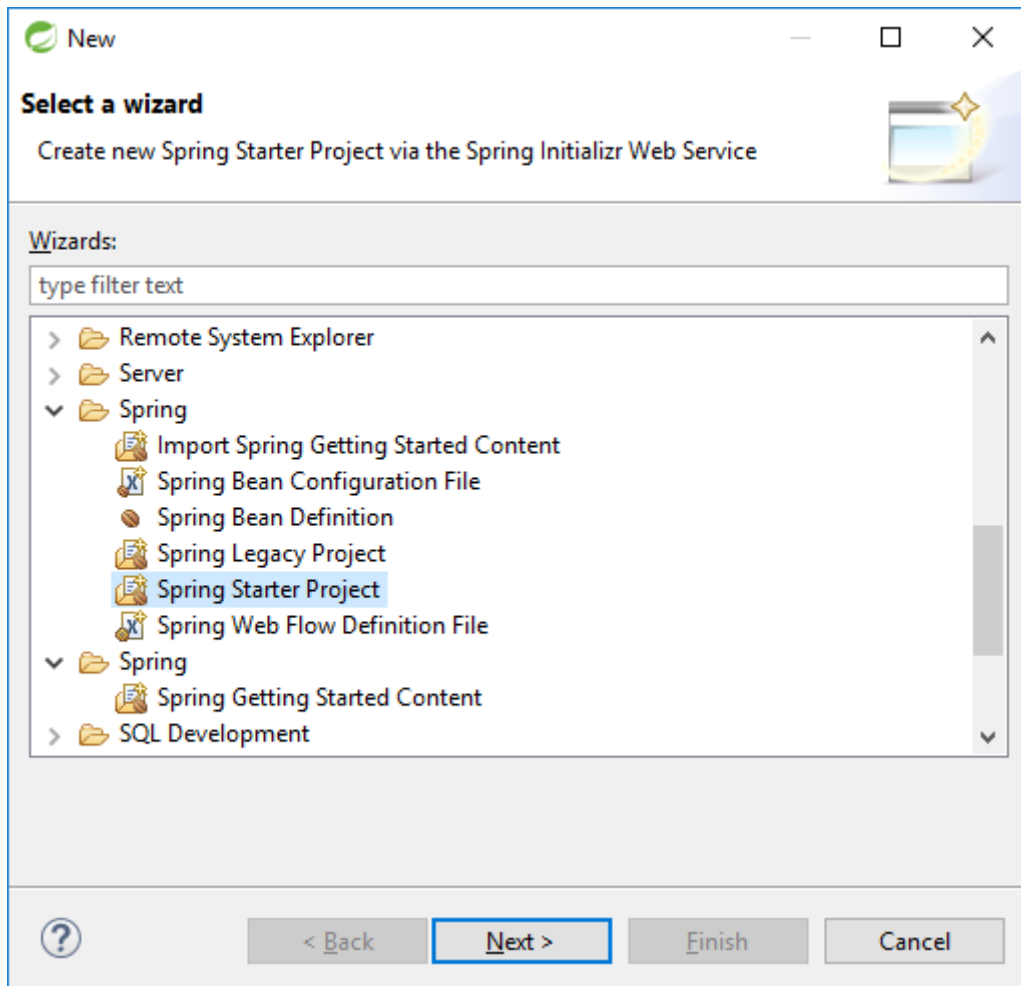
- Versión de java
- El tipo de packaging
- El lenguaje del proyecto
- Selección más detallada de las dependencias

La estructura del proyecto con dependencia web generado será



En esta estructura, cabe destacar el directorio **static**, destinado a contener cualquier recurso estatico de una aplicación web.

Desde Spring Tools Suite, se puede acceder a esta misma funcionalidad desde **New > Other > Spring > Spring Starter Project**, es necesario tener internet, ya que STS se conecta a start.spring.io



Una vez seleccionada la opción, se muestra un formulario similar al de la web

New Spring Starter Project

⚠ A project with name 'ListadoDeLibrosSeguro' already exists in the workspace.

Name: ListadoDeLibrosSeguro

☒ Use default location

Location: D:\workspace\ListadoDeLibrosSeguro Browse

Type: Maven Packaging: War

Java Version: 1.8 Language: Java

Group: com.example.spring.boot

Artifact: ListadoDeLibrosSeguro

Version: 0.0.1-SNAPSHOT

Description: Listado De Libros Seguro con Spring Boot

Package: com.example.spring.boot

Working sets

☐ Add project to working sets New...

Working sets: Select...

? < Back Next > Finish Cancel

Y desde Spring CLI con el comando **init** tambien, un ejemplo de comando seria

```
Spring-CLI# init --build maven --groupId com.ejemplo.spring.boot.web --version 1.0  
--java-version 1.8 --dependencies web --name HolaMundo HolaMundo
```

Que genera la estructura anterior dentro de la carpeta **HolaMundo**

Se puede obtener ayuda sobre los parametros con el comando

```
Spring-CLI# init --list
```

Starters

Son dependencias ya preparadas por Spring, para dotar del conjunto de librerías necesarias para obtener un funcionalidad sin que existan conflictos entre las versiones de las distintas librerías.

Se pueden conocer las dependencias reales con las siguientes tareas

- Maven

```
mvn dependency:tree
```

- Gradle

```
gradle dependencies
```

De necesitarse, se pueden sobrescribir las versiones o incluso excluir librerías, de las que nos proporcionan los **Starter**

- Maven

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <exclusions>
    <exclusion>
      <groupId>com.fasterxml.jackson.core</groupId>
    </exclusion>
  </exclusions>
</dependency>
```

- Gradle

```
compile("org.springframework.boot:spring-boot-starter-web") {
  exclude group: 'com.fasterxml.jackson.core'
}
```

Soporte a propiedades

Spring Boot permite configurar unas 300 propiedades, [aquí](#) una lista de ellas.

Se pueden configurar los proyectos de Spring Boot únicamente modificando propiedades, estas se pueden definir en

- Argumentos de la línea de comandos


```
java -jar app-0.0.1-SNAPSHOT.jar --spring.main.show-banner=false
```

- JNDI

```
java:comp/env/spring.main.show-banner=false
```

- Propiedades del Sistema Java

```
java -jar app-0.0.1-SNAPSHOT.jar -Dspring.main.show-banner=false
```

- Variables de entorno del SO

```
SET SPRING_MAIN_SHOW_BANNER=false;
```

- Un fichero **application.properties**

```
spring.main.show-banner=false
```

- Un fichero **application.yml**

```
spring:
  main:
    show-banner: false
```

Las listas en formato **YAML** tiene la siguiente sintaxis

```
security:
  user:
    role:
      - SUPERUSER
      - USER
```

De existir varias de las siguientes, el orden de preferencia es el del listado, por lo que la mas prioritaria es la linea de comandos.

Los ficheros **application.properties** y **application.yml** pueden situarse en varios lugares

- En un directorio **config** hijo del directorio desde donde se ejecuta la aplicación.
- En el directorio desde donde se ejecuta la aplicación.
- En un paquete **config** del proyecto
- En la raiz del classpath.

Siendo el orden de preferencia el del listado, si aparecieran los dos ficheros, el **.properties** y el **.yml**, tiene prioridad el **properties**.

Algunas de las propiedades que se pueden definir son:

- **spring.main.show-banner** → Mostrar el banner de spring en el log (por defecto true).
- **spring.thymeleaf.cache** → Deshabilitar la cache del generador de plantillas thymeleaf
- **spring.freemarker.cache** → Deshabilitar la cache del generador de plantillas freemarker
- **spring.groovy.template.cache** → Deshabilitar la cache de plantillas generadas con groovy
- **spring.velocity.cache** → Deshabilitar la cache del generador de plantillas velocity
- **spring.profiles.active** → Perfil activado en la ejecución

TIP

La cache de las plantillas, se emplea en producción para mejorar el rendimiento, pero se debe desactivar en desarrollo ya que sino se ha de parar el servidor cada vez que se haga un cambio en las plantillas.

Configuración del Servidor

- **server.port** → Puerto del Contenedor Web donde se exponen los recursos (por defecto 8080, para ssl 8443).
- **server.contextPath** → Permite definir el primer nivel del path de las url para el acceso a la aplicación (Ej: /resource).
- **server.ssl.key-store** → Ubicación del fichero de certificado (Ej: [file:///path/to/mykeys.jks](#)).
- **server.ssl.key-store-password** → Contraseña del almacén.
- **server.ssl.key-password** → Contraseña del certificado.

TIP

Para generar un certificado, se puede emplear la herramienta keytool* que incluye la jdk

```
keytool -keystore mykeys.jks -genkey -alias tomcat -keyalg RSA
```

Configuración del Logger

- **logging.level.root** → Nivel del log para el log principal (Ej: WARN)
- **logging.level.<paquete>** → Nivel del log para un log particular (Ej: logging.level.org.springframework.security: DEBUG)
- **logging.path** → Ubicación del fichero de log (Ej: /var/logs/)
- **logging.file** → Nombre del fichero de log (Ej: miApp.log)

Configuración del Datasource

- **spring.datasource.url** → Cadena de conexión con el origen de datos por defecto de la auto-

configuración (Ej: jdbc:mysql://localhost/test)

- **spring.datasource.username** → Nombre de usuario para conectar al origen de datos por defecto de la auto-configuración (Ej: dbuser)
- **spring.datasource.password** → Password del usuario que se conecta al origen de datos por defecto de la auto-configuración (Ej: dbpass)
- **spring.datasource.driver-class-name** → Driver a emplear para conecta con el origen de datos por defecto de la auto-configuración (Ej: com.mysql.jdbc.Driver)
- **spring.datasource.jndi-name** → Nombre JNDI del datasorce que se quiere emplear como origen de datos por defecto de la auto-configuración.
- **spring.datasource.name** → El nombre del origen de datos
- **spring.datasource.initialize** → Whether or not to populate using data.sql (default:true)
- **spring.datasource.schema** → The name of a schema (DDL) script resource
- **spring.datasource.data** → The name of a data (DML) script resource
- **spring.datasource.sql-script-encoding** → The character set for reading SQL scripts
- **spring.datasource.platform** → The platform to use when reading the schema resource (for example, "schema-{platform}.sql")
- **spring.datasource.continue-on-error** → Whether or not to continue if initialization fails (default: false)
- **spring.datasource.separator** → The separator in the SQL scripts (default: ;)
- **spring.datasource.max-active** → Maximum active connections (default: 100)
- **spring.datasource.max-idle** → Maximum idle connections (default: 8)
- **spring.datasource.min-idle** → Minimum idle connections (default: 8)
- **spring.datasource.initial-size** → The initial size of the connection pool (default: 10)
- **spring.datasource.validation-query** → A query to execute to verify the connection
- **spring.datasource.test-on-borrow** → Whether or not to test a connection as it's borrowed from the pool (default: false)
- **spring.datasource.test-on-return** → Whether or not to test a connection as it's returned to the pool (default: false)
- **spring.datasource.test-while-idle** → Whether or not to test a connection while it is idle (default: false)
- **spring.datasource.max-wait** → The maximum time (in milliseconds) that the pool will wait when no connections are available before failing (default: 30000)
- **spring.datasource.jmx-enabled** → Whether or not the data source is managed by JMX (default: false)

TIP

Solo se puede configurar un unico datasource por auto-configuración, para definir otro, se ha de definir el bean correspondiente

Custom Properties

Se puede definir nuevas propiedades y emplearlas en la aplicación dentro de los Bean.

- Para ello se ha de definir, dentro de un Bean de Spring, un atributo de clase que refleje la propiedad y su método de SET

```
private String prefijo;  
public void setPrefijo(String prefijo) {  
    this.prefijo = prefijo;  
}
```

- Para las propiedades con nombre compuesto, se ha de configurar el prefijo con la anotación **@ConfigurationProperties** a nivel de clase

```
@Controller  
@RequestMapping("/")  
@ConfigurationProperties(prefix="saludo")  
public class HolaMundoController {}
```

- Ya solo falta definir el valor de la propiedad en **application.properties** o en **application.yml**

```
saludo:  
  prefijo: Hola
```

TIP

Para que la funcionalidad de properties funcione, se debe añadir **@EnableConfigurationProperties**, pero con Spring Boot no es necesario, ya que está incluido por defecto.

Otra opción para emplear propiedades, es el uso de la anotación **@Value** en cualquier propiedad de un bean de spring, que permite leer la propiedad si esta existe o asignar un valor por defecto en caso que no exista.

```
@Value("${message:Hello default}")  
private String message;
```

Profiles

Se pueden anotar **@Bean** con **@Profile**, para que dicho Bean sea solo añadido al contexto de Spring cuando el profile indicado esté activo.

```

@Bean
@Profile("production")
public DataSource dataSource() {
    DataSource ds = new DataSource();
    ds.setDriverClassName("org.mysql.Driver");
    ds.setUrl("jdbc:mysql://localhost:5432/test");
    ds.setUsername("admin");
    ds.setPassword("admin");
    return ds;
}

```

También se puede definir un conjunto de propiedades que solo se empleen si un perfil está activo, para ello, se ha de crear un nuevo fichero **application-{profile}.properties**.

En el caso de los ficheros de YAML, solo se define un fichero, el **application.yml**, y en él se definen todos los perfiles, separados por ---

```

---
spring:
  profiles: production
  datasource:
    url: jdbc:mysql://localhost:5432/test
    username: admin
    password: admin
  jpa:
    database-platform: org.hibernate.dialect.MySQLDialect

```

Para activar un **Profile**, se emplea la propiedad **spring.profiles.active**, la cual puede establecerse como:

- Variable de entorno

```
SET SPRING_PROFILES_ACTIVE=production;
```

- Con un parametro de inicio

```
java -jar aplicacion-0.0.1-SNAPSHOT.jar --spring.profiles.active=production
```

TIP

De definirse más de un perfil activo, se indicarán con un listado separado por comas

JPA

Al añadir el starter de JPA, por defecto Spring Boot va a localizar todos los Beans dentro del paquete y subpaquetes donde se encuentra la clase anotada con **@SpringBootApplication** en busca de interfaces Repositorio, que extiendan la interface **JpaRepository**, de no encontrarse la interface

que define el repositorio dentro del paquete o subpaquetes, se puede referenciar con **@EnableJpaRepositories**

Cuando se emplea JPA con Hibernate como implementación, éste último tiene la posibilidad de configurar su comportamiento con respecto al schema de base de datos, pudiendo indicarle que lo cree, que lo actualice, que lo borre, que lo valide... esto se consigue con la propiedad **hibernate.ddl-auto**

```
spring:
  jpa:
    hibernate:
      ddl-auto: validate
```

NOTE

Los posibles valores para esta propiedad son:

- none -> This is the default for MySQL, no change to the database structure.
- update -> Hibernate changes the database according to the given Entity structures.
- create -> Creates the database every time, but don't drop it when close.
- create-drop -> Por defecto para H2. the database then drops it when the SessionFactory closes.

Habrà que añadir al classpath, con dependencias de Maven, el driver de la base de datos a emplear, Spring Boot detectará el driver añadido y conectará con una base de datos por defecto.

La dependencia para MySQL será

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
</dependency>
```

NOTE

Las versiones de algunas dependencias no es necesario que se indiquen en Spring Boot, ya que vienen predefinidas en el **parent**

Para configurar un nuevo origen de datos, se

```
spring.jpa.hibernate.ddl-auto=create
spring.datasource.url=jdbc:mysql://localhost:3306/db_example
spring.datasource.username=springuser
spring.datasource.password=ThePassword
```

Errores

Por defecto Spring Boot proporciona una pagina para represenar los errores que se producen en las aplicaciones llamada **whitelabel**, para sustituirla por una personalizada, basta con definir alguno de los siguientes componentes

- Cuanlquier Bena que implemente **View** con Id **error**, que será resuelto por **BeanNameViewResolver**.
- Plantilla **Thymeleaf** llamada **error.html** si **Thymeleaf** esta configurado.
- Plantilla **FreeMarker** llamada **error.ftl** si **FreeMarker** esta configurado.
- Plantilla **Velocity** llamada **error.vm** si **Velocity** esta configurado.
- Plantilla **JSP** llamada **error.jsp** si se emplean vistas JSP.

Dentro de la vista, se puede acceder a la siguiente información relativa al error

- **timestamp** → La hora a la que ha ocurrido el error
- **status** → El código HTTP
- **error** → La causa del error
- **exception** → El nombre de la clase de la excepción.
- **message** → El mensaje del error
- **errors** → Los errores si hay mas de uno
- **trace** → La traza del error
- **path** → La URL a la que se accedia cuando se produjo el error.

Seguridad de las aplicaciones

Para añadir Spring security a un proyecto, habrá que añadir

- En Maven

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

- En Gradle

```
compile("org.springframework.boot:spring-boot-starter-security")
```

Al añadir Spring Security al Classpath, automaticamente Spring Boot, hace que la aplicación sea segura, nada es accesible.

Se creará un usuario por defecto **user** cuyo password e generará cada vez que se arranque la

aplicación y se pintará en el log

```
Using default security password: ce9dadfa-4397-4a69-9fc7-af87e0580a10
```

Evidentemente esto es configurable, dado que cada aplicación, tendrá sus condiciones de seguridad, para establecer la configuración se puede añadir una nueva clase de configuración, anotada con **@Configuration** y además para que permita configurar la seguridad, debe estar anotada con **@EnableWebSecurity** y extender de **WebSecurityConfigurerAdapter**

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
    private ReaderRepository readerRepository;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
            .antMatchers("/").access("hasRole('READER')")
            .antMatchers("/**").permitAll()
            .and()
            .formLogin()
            .loginPage("/login")
            .failureUrl("/login?error=true");
    }
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth
            .userDetailsService(new UserDetailsService() {
                @Override
                public UserDetails loadUserByUsername(String username) throws
                UsernameNotFoundException {
                    return readerRepository.findOne(username);
                }
            });
    }
}
```

En esta clase, se puede configurar tanto los requisitos para acceder a recursos via web (autorización), como la vía de obtener los usuarios validos de la aplicación (autenticación), como otras configuraciones propias de la seguridad, como SSL, la pagina de login, ...

Soporte Mensajería JMS

Para emplear JMS de nuevo Spring Boot, proporciona un starter, en este caso para varias tecnologías: ActiveMQ, Artemis y HornetQ

Para añadir por ejemplo ActiveMQ, se añadirá a dependencia Maven

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-activemq</artifactId>
  </dependency>
</dependencies>
```

Una vez Spring Boot encuentra en el classpath el jar de **ActiveMQ**, creará los objetos necesarios para conectarse al recurso JMS **Topic/Queue**, simplemente hará falta configurar las siguientes propiedades para indicar donde se encuentra el endpoint de **ActiveMQ**

- **spring.activemq.broker-url** → (Ej: tcp://localhost:61616)
- **spring.activemq.user** → (Ej: admin)
- **spring.activemq.password** → (Ej: admin)

NOTE

Se espera que este configurado un endpoint de ActiveMQ, se puede descargar la distribución de [aquí](#).

Para arrancarlo se ha de ejecutar el comando **/bin/activemq start** que levanta el servicio en local con los puertos **8161** para la consola administrativa y **61616** para la comunicacion de con los clientes.

El usuario y password por defecto son **admin/admin**

Como es habitual en las aplicaciones **Boot**, no será necesario añadir la anotacion **@EnableJms** a la clase de aplicación, ya que estará contemplada con **@SpringBootApplication**.

Consumidores

Para definir un Bean que consuma los mensajes del servicio JMS, se emplea la anotación **@JmsListener**

```
@Component
public class Receiver {
    @JmsListener(destination = "mailbox")
    public void receiveMessage>Email email) {
        System.out.println("Received <" + email + ">");
    }
}
```

NOTE

Debera existir un **Queue** o **Topic** denominado **mailbox**

Productores

Para definir un Bean que envíe mensajes se empleará un **Bean** creado por Spring de tipo **JmsTemplate**, empleando las funcionalidades **send** o **convertAndSend**.

```
@Component
public class MyBean {
    private JmsTemplate jmsTemplate;
    @Autowired
    public MyBean(JmsTemplate jmsTemplate) {
        this.jmsTemplate = jmsTemplate;
    }
}
```

Se puede redefinir la factoria de contenedores

```
@Bean
public JmsListenerContainerFactory<?> myFactory(ConnectionFactory connectionFactory,
DefaultJmsListenerContainerFactoryConfigurer configurer) {
    DefaultJmsListenerContainerFactory factory = new
DefaultJmsListenerContainerFactory();
    // This provides all boot's default to this factory, including the message
converter
    configurer.configure(factory, connectionFactory);
    // You could still override some of Boot's default if necessary.
    return factory;
}
```

Indicandolo posteriormente en los listener

```
@Component
public class Receiver {
    @JmsListener(destination = "mailbox", containerFactory = "myFactory")
    public void receiveMessage(Email email) {
        System.out.println("Received <" + email + ">");
    }
}
```

Soporte Mensajería AMQP

AMQP es una especificación de mensajería asíncrona, donde todos los bytes transmitidos son especificados, por lo que se pueden crear implementaciones en distintas plataformas y lenguajes.

La principal diferencia con JMS, es que mientras que en JMS los productores pueden publicar mensajes sobre: * **Queue** (un consumidor) * y **Topics** (n consumidores)

En AMQP solo hay **Queue** (un receptor), pero se incluye una capa por encima de estas **Queue**, los **Exchange**, que es donde se publican los mensajes por parte de los productores y estos **Exchange**, tienen la capacidad de publicar los mensajes que les llegan en una sola **Queue** o en varias, emulando los dos comportamientos de JMS.

Para trabajar con AMQP, se necesita un servidor de AMQP, como **RabbitMQ**, para instalarlo se necesita instalar [Erlang](#) a parte de [RabbitMQ](#)

Además de instalar **Erlang**, habra que definir a variable de entorno **ERLANG_HOME**.

La configuracion por defecto de **RabbitMQ** es escuchar por el puerto 5672

Receiver

Una vez instalado el Bus, se necesitará un proyecto que defina los **Receiver**, para ello se ha de incluir la dependencia

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

Y añadir un **Bean** al contexto de Spring que haga de **Receiver**, no tiene porque implementar ningun API.

```
@Component
public class Receiver {
    public void receiveMessage(String message) {
        System.out.println("Received <" + message + ">");
    }
}
```

Este **Bean**, se ha de registrar como **Receiver** AMQP, para ello lo primero es indicar que método se ha de ejecutar, para ello se emplea la clase **MessageListenerAdapter**.

```
@Bean
public MessageListenerAdapter listenerAdapter(Receiver receiver) {
    return new MessageListenerAdapter(receiver, "receiveMessage");
}
```

En segundo lugar, hay que definir los objetos que definen la estructura del Broker

- Queue
- Exchange
- Binding

```

final static String queueName = "spring-boot";

@Bean
public Queue queue() {
    return new Queue(queueName, false);
}

@Bean
public TopicExchange exchange() {
    return new TopicExchange("spring-boot-exchange");
}

@Bean
public Binding binding(Queue queue, TopicExchange exchange) {
    return BindingBuilder.bind(queue).to(exchange).with(queueName);
}

```

Y por último asociar el **Receiver**, con la estructura del **Broker** a través de un **ConnectionFactory**, que creará Spring gracias a los Beans anteriormente configurados

```

@Bean
public SimpleMessageListenerContainer container(ConnectionFactory connectionFactory,
MessageListenerAdapter listenerAdapter) {
    SimpleMessageListenerContainer container = new SimpleMessageListenerContainer();
    container.setConnectionFactory(connectionFactory);
    container.setQueueNames(queueName);
    container.setMessageListener(listenerAdapter);
    return container;
}

```

Producer

Para producir nuevos mensajes, se emplea la clase **RabbitTemplate**, que permite enviar mensajes con métodos **convertAndSend**

```

rabbitTemplate.convertAndSend(ConfiguracionAMQP.queueName, "Hello from RabbitMQ!");

```

Este objeto, será creado por el contexto de Spring con los Bean que

Spring Cloud

Arquitectura de Microservicios

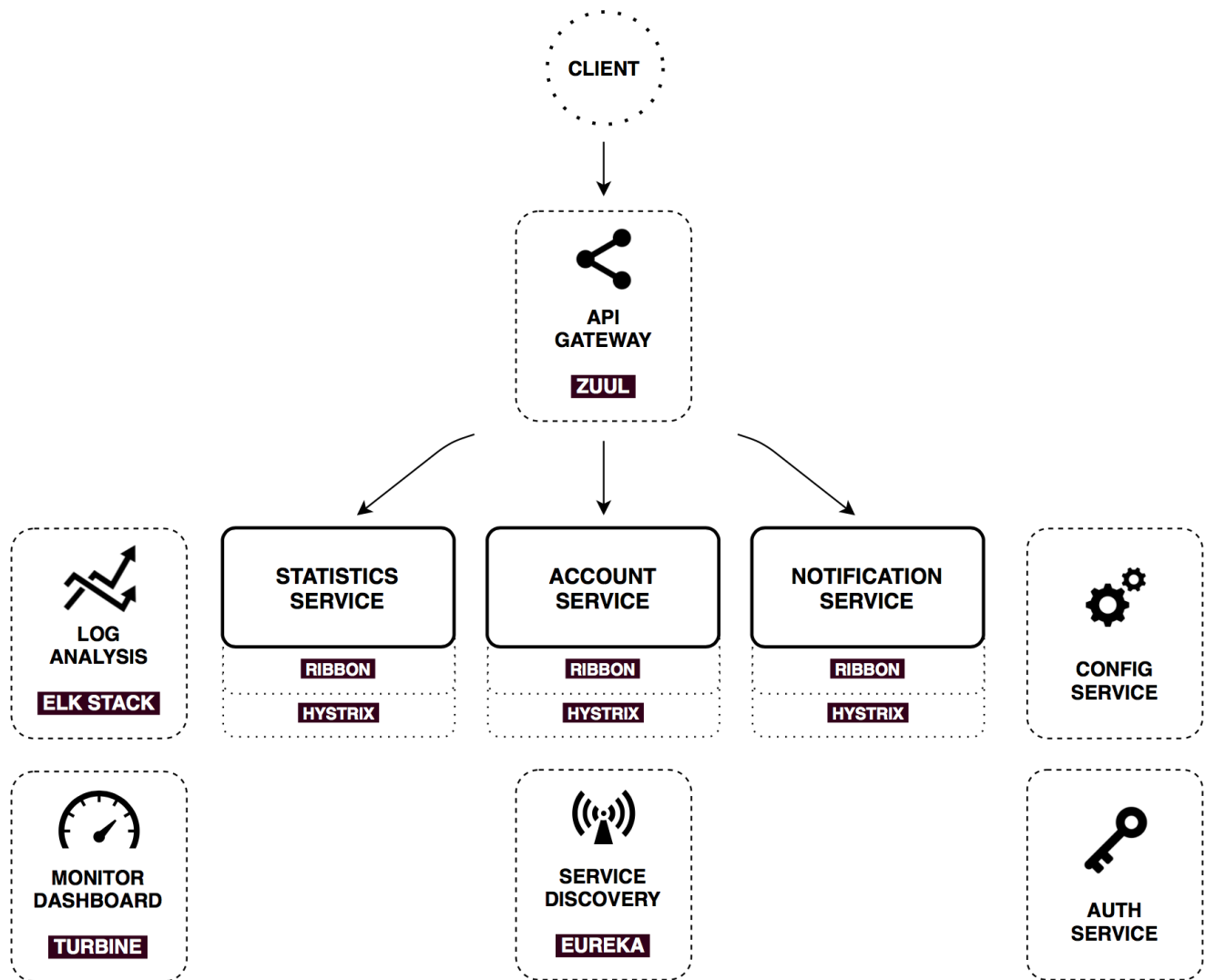
Cuando se habla de arquitectura de microservicios, se habla de **Cloud** o de arquitectura distribuida.

En esta arquitectura, se pueden producir problemas propios de la arquitectura relacionados con

- Monitorización de la arquitectura,
- Configuración de los microservicios,
- Descubrimiento de microservicios,
- Detección de caída de microservicios,
- Balanceo de carga,
- Seguridad centralizada,
- Logs centralizados, ...

Para ello, se suelen emplear herramientas que aplican patrones que permiten solventar/controlar dichos problemas, algunos de ellos son:

- **Configuración distribuida** → Se traduce en un **Servidor de Configuración** que permite centralizar las configuraciones de todos los microservicios que forman el sistema en un único punto, facilitando la gestión y posibilitando cambios en caliente.
- **Registro y autoreconocimiento de servicios** → Permite registrar instancias de servicios y exponerlos de forma integrada, es como unas paginas amarillas de servicios.
- **Enrutado** → Permite definir rutas dentro de la arquitectura para acceder a los microservicios.
- **Balanceo de carga (LoadBalancing)** → Necesidad de que los clientes puedan elegir cual de las instancias de un mismo servicio al que desean conectarse se va a emplear, todo de forma transparente. Esta funcionalidad se basa en obtener las instancias del Servidor de Registro y Descubrimiento.
- **Control de ruptura de comunicación con los servicios (CircuitBreaker)** → Permite controlar que la caída de un microservicio consultado, no provoque la caída de los microservicios que realizan la consulta, proporcionando un resultado estatico para la consulta.
- **Mensajería distribuida** → Permite emplear un bus de mensajería para desacoplar los microservicios entre si, además de poder propagar los cambios en la configuración de los microservicios.
- **Gestión centralizada de logs** → Permite controlar el estado de todas las instancias de todos los microservicios desde una misma aplicación, aunque cada microservicio tiene su propio sistema de logs, además de salvaguardar el tamaño en disco que requiera la aplicación o permitir una trazabilidad de las interacciones entre microservicios, que de otra forma seria compleja.
- **Monitorización** → Permite visualizar de forma centralizada el estado de todas las instancias de todos los microservicios.
- **Servidor de Seguridad centralizado** → Permite gestionar el acceso a los recursos de forma centralizada, sin que los propios microservicios se encarguen de esa logica. Para ello se emplea OAuth.



Orquestacion vs Coreografia

Se habla de **Orquestación**, cuando una aplicación, gestiona como invocar a otras aplicaciones, estableciendo los criterios y orden de invocación.

Se habla de **Coreografía**, cuando una aplicación produce un evento, que hace que otras aplicaciones realicen una acción (Bus de mensajería).

Servidor de Configuración

Las aplicaciones que contienen los microservicios se conectarán al servidor de configuración para obtener configuraciones.

El servidor se conecta a un repositorio **git** de donde saca las configuraciones que expone, lo que permite versionar fácilmente dichas configuraciones.

El OSS de Netflix proporciona para esta labor **Archaius**.

Para levantar un servidor de configuración, debemos incluir la dependencia

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

Para definir una aplicación como **Servidor de Configuración** basta con realizar dos cosas

- Añadir la anotación **@EnableConfigServer** a la clase **@SpringBootApplication** o **@Configuration**.

```
@EnableConfigServer
@SpringBootApplication
public class ConfigurationApplication {
    public static void main(String[] args) {
        SpringApplication.run(ConfigurationApplication.class, args);
    }
}
```

- Definir en las propiedades de la aplicación, la conexión con el repositorio **git** que alberga las configuraciones.

```
spring.cloud.config.server.git.uri=https://github.com/victorherreroazzurro/config-
cloud
spring.cloud.config.server.git.basedir=config
```

NOTE | La uri puede ser hacia un repositorio local.

En el repositorio **git** deberán existir tantos ficheros de **properties** o **yml** como aplicaciones configuradas, siendo el nombre de dichos ficheros, el nombre que se le dé a las aplicaciones

Por ejemplo si hay un microservicio que va a obtener su configuración del servidor de configuración, configurado en el **application.properties** con el nombre

```
spring.application.name=microservicio
```

o **application.yml**

```
spring:
  application:
    name:microservicio
```

Debera existir en el repositorio git un fichero **microservicio.properties** o **microservicio.yml**.

Las propiedades son expuestas via servicio REST, pudiendose acceder a ellas siguiendo estos

```
/{application}/{profile}/{label}]
/{application}-{profile}.yaml
/{label}/{application}-{profile}.yaml
/{application}-{profile}.properties
/{label}/{application}-{profile}.properties
```

Donde

- **application** → será el identificador de la aplicación **spring.application.name**
- **profile** → será uno de los perfiles definidos, sino se ha definido ninguno, siempre estará **default**
- **label** → será la rama en el repo Git, la por defecto **master**

Seguridad

Las funcionalidades del servidor de configuración están securizadas, para que cualquier usuario no pueda cambiar los datos de configuración de la aplicación.

Para configurar la seguridad, hay que añadir la siguiente dependencia

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Cuando se arranca el servidor, se imprimirá un password en la consola

```
Using default security password: 60bc8f1a-477d-484f-aaf8-da7835c207ab
```

Que sirve como password para el usuario **user**. Si se desea otra configuración se habrá de configurar con Spring Security.

Se puede establecer con la propiedad

```
security:
  user:
    password: mipassword
```

Clientes del Servidor de Configuración

Una vez definido el **Servidor de Configuración**, los microservicios se conectarán a él para obtener las configuraciones, para poder conectar estos microservicios, se debe añadir las dependencias


```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Siempre que se añada dependencias de spring cloud, habra que configurar

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>Camden.SR6</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

Y configurar a través del fichero **bootstrap.properties** donde encontrar el **Servidor de Configuración**. Se configura el fichero **bootstrap.properties**, ya que se necesita que los properties sean cargados antes que el resto de configuraciones de la aplicación.

```
spring.application.name=microservicio

spring.cloud.config.enabled=true
spring.cloud.config.uri=http://localhost:8888
```

El puerto 8888 es el puerto por defecto donde se levanta el servidor de configuracion, se puede modificar añadiendo al **application.yml**

```
server:
  port: 8082
```

Dado que el **Servidor de Configuración** estará securizado se debera indicar las credenciales con la sintaxis

```
spring.cloud.config.uri=http://usuario:password@localhost:8888
```

Si se quiere evitar que se arranque el microservicio si hay algun problema al obtener la

configuración, se puede definir

```
spring.cloud.config.fail-fast=true
```

Una vez configurado el acceso del microservicio al **Servidor de Configuración**, habrá que configurar que hacer con las configuraciones recibidas.

```
@RestController
class HolaMundoController {

    @Value("${message:Hello default}")
    private String message;

    @RequestMapping("/")
    public String home() {
        return message;
    }
}
```

En este caso se accede a la propiedad message que se obtendrá del servidor de configuración, de no obtenerla su valor será **Hello default**.

Actualizar en caliente las configuraciones

Dado que las configuraciones por defecto son solo cargadas al levantar el contexto, si se desea que los cambios en las configuraciones tengan repercusión inmediata, habrá que realizar configuraciones, en este caso la configuración necesaria supone añadir la anotación **@RefreshScope** sobre el componente a refrescar.

```
@RefreshScope
@RestController
class HolaMundoController {

    @Value("${message:Hello default}")
    private String message;

    @RequestMapping("/")
    public String home() {
        return message;
    }
}
```

Una vez preparado el microservicio para aceptar cambios en caliente, basta con hacer el cambio en el repo Git e invocar el servicio de refresco del microservicio del cual ha cambiado su configuración

```
(POST) http://<usuario>:<password>@localhost:8080/refresh
```

Este servicio de refresco es seguro por lo que habrá que configurar la seguridad en el microservicio

Servidor de Registro y Descubrimiento

Permite gestionar todas las instancias disponibles de los microservicios.

Los microservicios enviarán su estado al servidor Eureka a través de mensajes **heartbeat**, cuando estos mensajes no sean correctos, el servidor desregistrará la instancia del microservicio.

Los clientes del servidor de registro, buscarán en las instancias disponibles del microservicio que necesiten.

Es habitual que los propios microservicios, a parte de registrarse en el servidor, sean a su vez clientes para consumir otros microservicios.

Se incluyen varias implementaciones en Spring Cloud para servidor de registro/descubrimiento, Eureka Server, Zookeeper, Consul ...

Para configurarlo hay que incluir la dependencia

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka-server</artifactId>
</dependency>
```

Se precisa configurar algunos aspectos del servicio, para ello en el fichero **application.yml** o **application.properties**

```
server:
  port: 8084 #El 8761 es el puerto para servidor Eureka por defecto

eureka:
  instance:
    hostname: localhost
    serviceUrl:
      defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/
  client:
    registerWithEureka: false
    fetchRegistry: false
```

Para arrancar el servicio Eureka, únicamente es necesario lanzar la siguiente configuración.

```

@SpringBootApplication
@EnableEurekaServer
public class RegistrationServer {

    public static void main(String[] args) {
        SpringApplication.run(RegistrationServer.class, args);
    }
}

```

Registrar Microservicio

Lo primero para poder registrar un microservicio en el servidor de descubrimiento es añadir la dependencia de maven

```

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>

```

Para registrar el microservicio habrá que añadir la anotación **@EnableDiscoveryClient**

```

@EnableAutoConfiguration
@EnableDiscoveryClient
@SpringBootApplication
public class GreetingServer {

    public static void main(String[] args) {
        SpringApplication.run(GreetingServer.class, args);
    }
}

```

Y se ha de configurar el nombre de la aplicación con el que se registrará en el servidor de registro Eureka.

```

spring:
  application:
    name: holamundo

eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8084/eureka/ # Ha de coincidir con lo definido en
el Eureka Server

```

El tiempo de refresco de las instancias disponibles para los clientes es de por defecto 30 sg, si se desea cambiar, se ha de configurar la siguiente propiedad

```
eureka:
  instance:
    leaseRenewalIntervalInSeconds: 10
```

NOTE

Puede ser interesante lanzar varias instancias del mismo microservicio, para que se registren en el servidor de Descubrimiento, para ello se pueden cambiar las propiedades desde el script de arranque

```
mvn spring-boot:run -Dserver.port=8081
```

Localizacion de Microservicio registrado en Eureka con Ribbon

El cliente empleará el API de **RestTemplate** al que se proxeara con el balanceador de carga **Ribbon** para poder emplear el servicio de localización de **Eureka** para consumir el servicio.

Se ha de definir un nuevo Bean en el contexto de Spring de tipo **RestTemplate**, al que se ha de anotar con **@LoadBalanced**

```
@Bean
@LoadBalanced
public RestTemplate restTemplate() {
    return new RestTemplate();
}
```

Una vez proxeadado, las peticiones empleando este **RestTemplate**, no se harán sobre el **EndPoint** del servicio, sino sobre el nombre del servicio con el que se registro en **Eureka**.

```
@Autowired
private RestTemplate restTemplate;

public MessageWrapper<Customer> getCustomer(int id) {
    Customer customer = restTemplate.exchange( "http://customer-service/customer/{id}",
    HttpMethod.GET, null, new ParameterizedTypeReference<Customer>() { }, id).getBody();
    return new MessageWrapper<>(customer, "server called using eureka with rest template");
}
```

Si el servicio es seguro, se pueden emplear las herramientas de **RestTemplate** para realizar la autenticación.

```
restTemplate.getInterceptors().add(new BasicAuthorizationInterceptor("user",
"mipassword"));

ResponseEntity<String> respuesta = restTemplate.exchange("http://holamundo",
HttpMethod.GET, null, String.class, new Object[]{});
```

Para que **Ribbon** sea capaz de enlazar la URL que hace referencia al identificador del servicio en **Eureka** con el servicio real, se debe configurar donde encontrar el servidor **Eureka**

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8084/eureka/
```

Y configurar la aplicación para que pueda consumir el servicio de **Eureka**

```
@SpringBootApplication
@EnableDiscoveryClient
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Uso de Ribbon sin Eureka

Se puede emplear el balanceador de carga Ribbon, definiendo un pool de servidores donde encontrar el servicio a consultar, no siendo necesario el uso de Eureka.

```
customer-service:
  ribbon:
    eureka:
      enabled: false
    listOfServers: localhost:8090,localhost:8290,localhost:8490
```

Simplificación de Clientes de Microservicios con Feign

Feign abstrae el uso del API de RestTemplate para consultar los microservicios, encapsulándolo todo con la definición de una interface.

Para activar su uso, lo primero será añadir la dependencia con Maven

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-feign</artifactId>
</dependency>
```

El siguiente paso sera activar el autodescubimiento de las configuraciones de **Feign**, como la anotacion **@FeignClient**, para lo que se ha de incluir la anotacion en la configuracion de la aplicaci3n **@EnableFeignClients**

```
@SpringBootApplication
@EnableFeignClients
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Luego se definen las interaces con la anotacion **@FeignClient**

```
@FeignClient(name="holamundo")
interface HolaMundoCliente {

    @RequestMapping(path = "/", method = RequestMethod.GET)
    public String holaMundo();
}
```

Solo resta asociar el nombre que se ha dado al cliente **Feign** con un sevicio real, para ello en el fichero **application.yml** y gracias a **Ribbon**, se pueden definir el pool de servidores que tienen el servicio a consumir.

```
holamundo:
  ribbon:
    listOfServers: http://localhost:8080
```

Acceso a un servicio seguro

Si al servicio al que hay que acceder es seguro, se pueden realizar configuraciones extras como el usuario y password, haciendo referencia a los **Beans** definidos en una clase de configuracion particular

```

@FeignClient(name="holamundo", configuration = Configuracion.class)
interface HolaMundoCliente {

    @RequestMapping(path = "/", method = RequestMethod.GET)
    public String holaMundo();
}

@Configuration
public class Configuracion {
    @Bean
    public BasicAuthRequestInterceptor basicAuthRequestInterceptor() {
        return new BasicAuthRequestInterceptor("user", "mipassword");
    }
}

```

Uso de Eureka

En vez de definir un pool de servidores en el cliente, se puede acceder al servidor **Eureka** facilmente, basta con tener la precaución de emplear en el **name** del Cliente **Feign**, el identificador en **Eureka** del servicio que se ha de consumir.

Añadir la anotacion **@EnableDiscoveryClient** para poder buscar en **Eureka**

```

@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}

```

Y configurar la direccion de **Eureka**, no siendo necesario configurar el pool de **Ribbon**

```

eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8084/eureka/

```

Servidor de Enrutado

Permite definir paths y asociarlos a los microservicios de la arquitectura, será por tanto el componente expuesto de toda la arquitectura.

Spring Cloud proporciona **Zuul** como Servidor de enrutado, que se acopla perfectamente con

Eureka, permitiendo definir rutas que se enlacen directamente con los microservicios publicados en **Eureka** por su nombre.

Se necesita añadir la dependencia Maven.

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-zuul</artifactId>
</dependency>
```

Lo siguiente es activar el Servidor **Zuul**, para lo cual habrá que añadir la anotación **@EnableZuulProxy** a una aplicación Spring Boot.

```
@SpringBootApplication
@EnableZuulProxy
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Solo restarán definir las rutas en el fichero **application.yml**

Estas pueden ser hacia el servicio directamente por su url

```
zuul:
  routes:
    holamundo:
      path: /holamundo/**
      url: http://localhost:8080/
```

Con lo que se consigue que las rutas hacia **zuul** con path **/holamundo/** se redireccionen hacia el servidor **http://localhost:8080/**

NOTE	Se ha de crear una clave nueva para cada enrutado, dado que la propiedad routes es un mapa, en este caso la clave es holamundo .
-------------	--

O hacia el servidor de descubrimiento **Eureka** por el identificador del servicio en **Eureka**

```
zuul:
  routes:
    holamundo:
      path: /holamundo/**
      #Para mapeo de servicios registrados en Eureka
      serviceId: holamundo
```

Para esto último, habra que añadir la dependencia de Maven

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
```

Activar el descubrimiento en el proyecto añadiendo la anotación **@EnableDiscoveryClient**

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableZuulProxy
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

E indicar en las propiedades del proyecto, donde se encuentra el servidor **Eureka**

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8084/eureka/
```

Seguridad

En el caso de enrutar hacia servicios seguros, se puede configurar **Zuul** para que siendo el que reciba los token de seguridad, los propague a los servicios a los que enruta, esta configuración por defecto viene desactivada dado que los servicios a los que redirecciona o tienen porque ser de la misma arquitectua y en ese caso, no sería seguro.

```
zuul:
  routes:
    holamundo:
      path: /holamundo/**
      #Para mapeo de las url directas a un servicio
      url: http://localhost:8080/

      #No se incluye ninguna cabecera como sensible, ya que todas las definidas
      como sensibles, no se propagan
      sensitive-headers:
        custom-sensitive-headers: true
      #Se evita añadir las cabeceras de seguridad a la lista de sensibles.
      ignore-security-headers: false
```

Circuit Breaker

La idea de este componente es la de evitar fallos en cascada, es decir que falle un componente, no por error propio del componente, sino porque falle otro componente de la arquitectura al que se invoca.

Para ello Spring Cloud integra **Hystrix**.

Para emplearlo, se ha de añadir la dependencia Maven

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-hystrix</artifactId>
</dependency>
```

La idea de este framework, es proxear la llamada del cliente del servicio sensible a caerse proporcionando una vía de ejecución alternativa **fallback**, para así evitar el error en la invocación.

Para ello se ha de anotar el método que haga la petición al cliente con **@HystrixCommand** indicando el método de **fallback**

```
@RestController
class HolaMundoClienteController {

    @Autowired
    private HolaMundoCliente holaMundoCliente;

    @HystrixCommand(fallbackMethod="fallbackHome")
    @RequestMapping("/")
    public String home() {
        return holaMundoCliente.holaMundo() + " con Feign";
    }

    public String fallbackHome() {
        return "Hubo un problema con un servicio";
    }
}
```

El método de **Fallback** deberá retornar el mismo tipo de dato que el método proxead.

NOTE

No deberán aplicarse las anotaciones sobre los controladores, dado que los proxys entran en conflicto

Para activar estas anotaciones se ha de añadir **@EnableCircuitBreaker**.

```
@SpringBootApplication
@EnableCircuitBreaker
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Monitorizacion: Hystrix Dashboard

Se ha de crear un nuevo servicio con la dependencia de Maven

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>
</dependency>
```

Y activar el servicio de monitorizacion con la anotacion **@EnableHystrixDashboard**

```
@SpringBootApplication
@EnableHystrixDashboard
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Se accederá al panel de monitorizacion en la ruta <http://<host>:<port>/hystrix> y allí se indicará la url del servicio a monitorizar <http://<host>:<port>/hystrix.stream>

Para que la aplicación configurada con **Hystrix** proporcione información a través del servicio **hystrix.stream**, se ha de añadir a dicha aplicación **Actuator**, con Maven.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Monitorizacion: Turbine

Se puede añadir un servicio de monitorización de varios servicios a la vez, llamado **Turbine**, para ello se ha de añadir la dependencia

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-netflix-turbine</artifactId>
</dependency>
```

Configuración Distribuida en Bus de Mensajería

Se trata de emplear un bus de mensajería para trasladar el evento de refresco a todos los nodos de los microservicios que emplean una configuración distribuida.

Se precisa por tanto de un bus de mensajería, en este caso Spring Cloud apuesta por implementaciones **AMQP** frente a otras alternativas como podrían ser **JMS**. Y más concretamente **RabbitMQ**.

Para instalar RabbitMQ, se necesita instalar [Erlang](#) a parte de [RabbitMQ](#)

La configuración por defecto de **RabbitMQ** es escuchar por el puerto 5672

Servidor

Se necesitará incluir las siguientes dependencias en el servidor de configuración

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-config-monitor</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

Y la siguiente configuración de la ubicación de RabbitMQ

application.yml

```
server:
  port: 8180

spring:
  cloud:
    bus:
      enabled: true #Habilitamos la publicacion en el bus

  #Indicamos donde esta el repositorio con las configuraciones
  config:
    server:
      git:
        uri: https://github.com/victorherrerocazurro/RepositorioConfiguraciones

  #Se necesita conocer donde esta rabbitMQ para enviar los eventos de cambio de
  #propiedades
  rabbitmq:
    host: localhost
    port: 5672
```

A partir de este punto el servidor aceptará el refresco de las propiedades a traves del bus, empleando el servicio **/monitor**, al cual podrán acceder los repositorio Git a traves de **Hook**

```
curl -v -X POST "http://localhost:8100/monitor" -H "Content-Type: application/json" -H
"X-Event-Key: repo:push" -H "X-Hook-UUID: webhook-uuid" -d '{"push": {"changes": []}}
}'
```

Cliente

Se necesitará incluir la siguiente dependencia en los clientes que servidor de configuración

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
```

Como bus por defecto se emplea **RabbitMQ**, al que habrá que configurar las siguientes propiedades

application.properties

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
```

OAuth2

Características

- Emplea Https en vez de la Criptografía de OAuth 1.
- Permite flujos con aplicaciones nativas y no solo con Navegadores.
- Posibilidad de asignación de Privilegios a los Token basados en la característica SCOPE (LEER, ESCRIBIR, ...)
- Caducidad de los Token, **Refresh Token**.

Que se quiere conseguir

- Diferenciar cuando el acceso a los datos privados lo hace el dueño de los datos y cuando lo hace una aplicación en nombre del usuario.
- Permitir al usuario dar distintos privilegios de acceso a las distintas aplicaciones que acceden en nombre del usuario a sus datos privados.
- Poder revocar los privilegios concedidos.

Actores

- **Resource Owner** - Usuario propietario de los **datos seguros**.
- **Resource Server** - Servidor que alberga los **datos seguros**, el servidor que realiza la autenticación por OAuth puede ser uno al poder permitir el acceso a los datos del perfil del usuario.
- **Authorization Server** - Plataforma OAuth, controla que clientes pueden acceder a los **datos seguros**.
- **Client** - Aplicación que quiere acceder a los **datos seguros** de un usuario, residentes en otra aplicación y cuyo usuario (propietario) es usuario de las dos aplicaciones tanto del cliente como del servidor de recursos.

NOTE

No tiene porque existir el concepto de la aplicación cliente, OAuth se puede emplear unicamente para generar el Token de autenticación, y no realizar peticiones desde una aplicación a los servidores de recursos impersonando al usuario.

Características

El dueño del **dato seguro**, es usuario del **Authorization Server**, donde describe su login y posiblemente otros datos seguros.

La aplicación (cliente), se registra en el servidor OAuth, para delegar en él, el proceso de Login.

El usuario del **Authorization Server**, se hace usuario también de la aplicación (cliente), otorgándole permisos para acceder a determinados **datos seguros** a través de los SCOPE.

Los SCOPE, son algo similar a los permisos (READ, WRITE,), que son definidos por el servidor de

recursos. Se definen de forma análoga a los ROLES asociados a las URLs.

Grants

Describen el dialogo que se produce entre los distintos actores en distintos escenarios.

En **Oauth2** se describen 5 formas por las cuales los clientes pueden obtener el **AccessToken**

- **Authorization Code Grant** → Es el más habitual en las aplicaciones web.
- **Implicit Grant** → Aplicaciones Javascript, que reducen el dialogo.
- **Resource Owner Credential Grant** o **Password Grant** → Para aplicaciones muy confiables, dado que se les da el user/password, es el típico caso de las aplicaciones en movilidad.
- **Client Credential Grant** → Empleada en comunicaciones entre aplicaciones, donde que el usuario especifique los permisos no es necesario.
- **Refresh token grant** → Permite obtener un AccessToken nuevo cuando el antiguo caduca.

Authorization Code Grant

1- Se define el **servidor a autorizacion**.

2- Se define la aplicación **cliente**, que se ha de dar de alta en el **servidor de autorizacion**, obteniendo un **Id** y un **Secret**, que debe almacenar porque estos datos se los tendrá que enviar posteriormente al **servidor de autorizacion**.

3- El usuario se registra en el **servidor a autorizacion**.

4- El usuario accede a la aplicacion **cliente** y está lo redirecciona al **servidor de autorizacion**, enviando a dicho servidor los siguientes datos

- **response_type** → Indica que es lo que espera la aplicacion cliente del proceso de login del usuario en el **servidor de autorizacion**. En este caso se indica que se quiere un **authorization code**, para ello se indica el valor **code**
- **client_id** → con el identificador de la aplicacion cliente en el **servidor de autorizacion**
- **redirect_uri** → con la url a la que el **servidor de autorizacion** deberá redireccionar al usuario una vez terminado el login y aprobación de scopes.
- **scope** → listado de scopes separados por comas que la aplicacion cliente requiere al usuario.
- **state** → con el **CSRFToken**

Si el usuario aprueba los **scopes**, el **servidor de autorizacion** lo redirecciona al **redirect_uri** o a uno especificado por la aplicacion cliente de forma generica, enviando

- **code** → con el **authorization code**
- **state** → con el mismo **CSRFToken** que se recibio

5- Cuando la **aplicacion cliente** recibe la petición, realiza una petición al **Servidor de Autenticacion** enviando

- **grant_type** → Indica en tipo de Grant a emplear, en este caso con valor **authorization_code**
- **client_id** → con el identificador de la aplicacion cliente en el **servidor de autorizacion**
- **client_secret** → con la clave secreta asociada al cliente
- **redirect_uri** → con la misma uri que paso por parametros al **servidor de autorizacion** cuando redirecciono al usuario
- **code** → con el **authorization code** que obtuvo de la anterior redireccion.

Y el **servidor de autorizacion** le retorna un JSON con

- **token_type** → con valor **Bearer**
- **expires_in** → con un entero que indica cuando expira el Token
- **access_token** → con el **accessToken**
- **refresh_token** → con un **refreshToken** que podrá ser empleado para obtener un nuevo **accessToken** cuando el original haya expirado

6- Por último se produce el acceso a los **Datos privados**, ya que la aplicacion **cliente** tiene el **accessToken**, le pide con este Token los datos privados al **Servidor de Recursos**, el cual se los da, dado que internamente es capaz de validarlo contra el **servidor de autorizacion**.

Implicit grant

Similar al anterior, pero dado que es el empleado por lo browser y estos no son capaces de asegurar guardar un **secret** de la aplicacion **cliente**, el flujo se simplifica, dado que en la redireccion del usuario hacia el **servidor de autorizacion** se obtendrá el **accesstoken**, no existen los **authorization code** y ni tampoco **refresh_token** dado que el browser no lo podria almacenar de forma segura.

1- Se define el **servidor a autorizacion**.

2- Se define la aplicación **cliente**, que se ha de dar de alta en el **servidor de autorizacion**, obteniendo un **Id**, que debe almacenar porque este dato se lo tendrá que enviar posteriormente al **servidor de autorizacion**.

3- El usuario se registra en el **servidor a autorizacion**.

4- El usuario accede a la aplicacion **cliente** y está lo redirecciona al **servidor de autorizacion**, enviando a dicho servidor los siguientes datos

- **response_type** con el valor **token**
- **client_id** con el id de la aplicacion cliente
- **redirect_uri** con la url a la que el **servidor de autorizacion** deberá redireccionar al usuario una vez terminado el login y aprobación de scopes.
- **scope** listado de scopes separados por comas que la aplicacion cliente requiere al usuario.
- **state** con el **CSRFToken**

Si el usuario aprueba los **scopes**, el **servidor de autorizacion** lo redirecciona a **redirect_uri**

enviando

- **token_type** con el valor **Bearer**
- **expires_in** con un entero que indica cuando expira el Token
- **access_token** con el **accessToken**
- **state** con el mismo **CSRFToken** que se recibio

Resource owner credentials grant

Es una forma de obtener el **accessToken** en la que hay una completa confianza en la aplicación **cliente**, dado que se le dan **login/password**.

1- Se define el **servidor a autorizacion**.

2- Se define la aplicación **cliente**, que se ha de dar de alta en el **servidor de autorizacion**, obteniendo un **Id**, que debe almacenar porque este dato se lo tendrá que enviar posteriormente al **servidor de autorizacion**.

3- El usuario se registra en el **servidor a autorizacion**.

4- El usuario accede a la aplicacion **cliente** y está le pregunta cual es su **login/password**

5- La aplicacion cliente realiza una peticion al **servidor de autorizacion**, enviando a dicho servidor los siguientes datos

- **grant_type** con el valor **password**
- **client_id** con el identificador de la aplicacion cliente en el **servidor de autorizacion**
- **client_secret** con la clave secreta asociada al cliente
- **scope** listado de scopes separados por comas que la aplicacion cliente requiere al usuario.
- **username** el login obtenido del usuario
- **password** la contraseña obtenida del usuario

El servidor de autorizacion responderá un JSON con los datos

- **token_type** con valor **Bearer**
- **expires_in** con un entero que indica cuando expira el Token
- **access_token** con el **accessToken**
- **refresh_token** con un **refreshToken** que podrá ser empleado para obtener un nuevo **accessToken** cuando el original haya expirado

Client credentials grant

Es el mas simple de los **Grant** de OAuth2

1- Se define el **servidor a autorizacion**.

2- Se define la aplicación **cliente**, que se ha de dar de alta en el **servidor de autorizacion**,

obteniendo un **Id** y un **+secret***, que debe almacenar porque este dato se lo tendrá que enviar posteriormente al **servidor de autorizacion**.

3- La aplicacion **cliente** realiza una peticion POST al **servidor de autorizacion**, enviando a dicho servidor los siguientes datos

- **grant_type** con el valor **client_credentials**
- **client_id** con el identificador de la aplicacion cliente en el **servidor de autorizacion**
- **client_secret** con la clave secreta asociada al cliente
- **scope** listado de scopes separados por comas que la aplicacion cliente requiere al usuario.

El servidor de autorizacion responderá un JSON con los datos

- **token_type** con valor **Bearer**
- **expires_in** con un entero que indica cuando expira el Token
- **access_token** con el **accessToken**

Refresh token grant

Se emplea cuando se puede regenerar el **accessToken** sin la intervencion del usuario

1- La aplicacion **cliente** realiza una peticion POST al **servidor de autorizacion**, enviando a dicho servidor los siguientes datos

- **grant_type** con el valor **refresh_token**
- **refresh_token** con el **refreshToken** que se obtuvo con el ahora caducado **accessToken**
- **client_id** con el identificador de la aplicacion cliente en el **servidor de autorizacion**
- **client_secret** con la clave secreta asociada al cliente
- **scope** listado de scopes separados por comas que la aplicacion cliente requiere al usuario.

El servidor de autorizacion responderá un JSON con los datos

- **token_type** con valor **Bearer**
- **expires_in** con un entero que indica cuando expira el Token
- **access_token** con el **accessToken**
- **refresh_token** con un **refreshToken** que podrá ser empleado para obtener un nuevo **accessToken** cuando el original haya expirado

Implementacion Servidor OAuth con Spring

Se ha de añadir la dependencia

```
<dependency>
  <groupId>org.springframework.security.oauth</groupId>
  <artifactId>spring-security-oauth2</artifactId>
</dependency>
```

Se ha de definir una clase de configuracion del contexto de spring, donde se añada la anotacion **@EnableAuthorizationServer**, ademas la clase deberá heredar de **AuthorizationServerConfigurerAdapter** para facilitar la configuracion.

```
@Configuration
@EnableAuthorizationServer
public class AuthServerOAuth2Config extends AuthorizationServerConfigurerAdapter {}
```

La configuracion a establecer se divide en tres,

- Definir como las aplicaciones clientes pueden acceder a los endpoints que permiten la gestion del **AccessToken**.
 - Obtencion de clave publica del certificado con el que se firma la información que provee el servidor, la cual se configura con el método **tokenKeyAccess** y responde a la url **/oauth/token_key**
 - Verificacion del **AccessToken** con el método **checkTokenAccess** y responde a la url **/oauth/check_token**

```
@Override
public void configure(AuthorizationServerSecurityConfigurer oauthServer) throws
Exception {
    oauthServer
        .tokenKeyAccess("permitAll()")
        .checkTokenAccess("isAuthenticated()");
}
```

- Definir las aplicaciones clientes que emplearán el servidor OAuth, indicando para cada una de ellas
 - IdCliente
 - Secret
 - Grant Types que puede emplear dicha aplicacion
 - Scopes
 - RedirectUri

```

@Override
public void configure(ClientDetailsServiceConfigurer clients) throws Exception {
    clients.inMemory()

        .withClient("confidential") //IdCliente
        .secret("secret")
        .authorizedGrantTypes("client_credentials", "authorization_code",
"refresh_token")
        .scopes("read", "write")
        .redirectUri("http://localhost:8080/client/")

        .and()

        .withClient("public")
        .authorizedGrantTypes("client_credentials", "implicit")
        .scopes("read")
        .redirectUri("http://localhost:8080/client/")

        .and()

        .withClient("trusted")
        .secret("secret")
        .authorities("ROLE_TRUSTED_CLIENT")
        .authorizedGrantTypes("client_credentials", "password", "authorization_code",
"refresh_token")
        .scopes("read", "write")
        .redirectUri("http://localhost:8080/client/");
}

```

- Definir como el servidor OAuth gestiona los **AccessToken**, tanto para el almacenamiento, la transformación (encriptación) y el acceso. Para esta configuración se precisa de la configuración de otros Beans
- **AuthenticationManager**, que será provisto por inyección por el contexto de Spring.
- **JwtAccessTokenConverter**, que será el encargado de definir como el servidor gestiona la encriptación de los Token.
- **JwtTokenStore**, que será el encargado de definir como se almacenan los Token.

```

@Override
public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws
Exception {
    endpoints
        .authenticationManager(authenticationManager)
        .tokenStore(tokenStore())
        .accessTokenConverter(tokenEnhancer());
}

@Value("${config.oauth2.privateKey}")
private String privateKey;

@Value("${config.oauth2.publicKey}")
private String publicKey;

@Autowired
private AuthenticationManager authenticationManager;

@Bean
public JwtAccessTokenConverter tokenEnhancer() {
    log.info("Initializing JWT with public key:\n" + publicKey);
    JwtAccessTokenConverter converter = new JwtAccessTokenConverter();
    converter.setSigningKey(privateKey);
    converter.setVerifierKey(publicKey);
    return converter;
}

@Bean
public JwtTokenStore tokenStore() {
    return new JwtTokenStore(tokenEnhancer());
}

```

Para esta configuracion se está empleando JWT (Json Web Token) para el almacen de los Token y su encriptado, para poder emplear este API, es necesario incluir la dependencia

```

<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-jwt</artifactId>
</dependency>

```

Y en el caso del ejemplo definir una clave publica/privada como propiedades o tambien sería posible el uso de un certificado

```

config:
  oauth2:
    # openssl genrsa -out jwt.pem 2048
    # openssl rsa -in jwt.pem
    privateKey: |
      -----BEGIN RSA PRIVATE KEY-----

MIICXQIBAAKBgQDNQZKqTlO/+2b4ZdhqGJzGBDltb5PZmBz1ALN2YLvt341pH6i5
m01V9cX5Ty1LM70fKfnIoYUP4KCE33dPnC7LkUwE/myh1zM6m8cbL5cYFPyP099t
hbVxzJkjhWqyvwQih/q00jliomKbM9pxG8Z1dB26hL9dSAZuA8xExjLPmQIDAQAB
AoGAImnYGU3ApPOVtBf/T0qLfne+2SZX96eVU06myDY3zA4r03DfbR7CzCLE6qPn
yDAIiw0UQBs0oBDdW0n0qz5YaePZu/yrLyj6KM6Q2e9ywRDtDh3ywrSfGpjdSvvo
aeL1WesBWsgWv1vFKKvES7ILFLUxKwyCRC2Lgh7aI9GGZfECQQD84m98Yrehhin3
fZuRaBNiU348Ci7ZFZmrvyxAIxrV4jBjpACW0RM2BvF5oYM2g0JqIfBOVjmPwUro
bYEFcHRvAkeAz8jsfmxsZVwh3Y/Y47BzhKIC5FLaads541jNjVWfrPirljyCy1n4
sg3WQH2IEyap3WTP84+csCtsfNfyK7fQdwJBAJNRyobY74cupJYkW50K40kXKQQL
Hp2iosJV/Y5jpQeC3JO/gARcSmfIBbbI66q9zKjtmpPYUXI4tc3PtUEY8QsCQQCc
xySyC0sKe6bNzyC+Q8AVkxiTKWiI5idEr8duhJd589H72Zc2wkMB+a2CEGo+Y5H
jy5cvuph/pG/7Qw7sljnAkAy/feCl1mUEiAcWrHRwcQ71AoA0+21yC9VkpPNrn3
w70Eg8gBqPjRlXBNb00QieNeGGSkX0oU6gFschR22Dzy
      -----END RSA PRIVATE KEY-----

    # openssl rsa -in jwt.pem -pubout
    publicKey: |
      -----BEGIN PUBLIC KEY-----

MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDNQZKqTlO/+2b4ZdhqGJzGBDlt
b5PZmBz1ALN2YLvt341pH6i5m01V9cX5Ty1LM70fKfnIoYUP4KCE33dPnC7LkUwE
/myh1zM6m8cbL5cYFPyP099thbVxzJkjhWqyvwQih/q00jliomKbM9pxG8Z1dB26
hL9dSAZuA8xExjLPmQIDAQAB
      -----END PUBLIC KEY-----

```

Además de la configuración de OAuth, se ha de configurar la propia del acceso de los usuarios al Servidor OAuth

NOTE

Recordad que hay dos tipos de login en el servidor de OAuth, el de los usuarios y el de las aplicaciones cliente.

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class ConfiguracionSeguridadWeb extends WebSecurityConfigurerAdapter {

    private static final Logger log = LoggerFactory.getLogger
(ConfiguracionSeguridadWeb.class);

    @Override
    @Autowired
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        log.info("Defining inMemoryAuthentication (2 users)");
        auth.inMemoryAuthentication()

            .withUser("user").password("password").roles("USER")

            .and()

            .withUser("admin").password("password").roles("USER", "ADMIN");
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.formLogin()

            .and()

            .httpBasic().disable()
            .anonymous().disable()
            .authorizeRequests().anyRequest().authenticated();
    }
}
```

Implementacion Aplicacion Cliente OAuth con Spring

Se ha de añadir la misma dependencia que en el servidor.

```
<dependency>
    <groupId>org.springframework.security.oauth</groupId>
    <artifactId>spring-security-oauth2</artifactId>
</dependency>
```

El primer paso será definir la aplicación como Cliente OAuth


```
@Configuration
@EnableOAuth2Client
public class ConfiguracionOAuth {}
```

El siguiente paso será definir un **Bean** de tipo **OAuth2RestOperations** que será el encargado de realizar las peticiones al **Servidor OAuth**

```
@Bean
public OAuth2RestOperations restTemplate(OAuth2ClientContext oauth2ClientContext) {
    return new OAuth2RestTemplate(resource(), oauth2ClientContext);
}

private OAuth2ProtectedResourceDetails resource() {
    AuthorizationCodeResourceDetails resource = new AuthorizationCodeResourceDetails(
    );
    resource.setClientId(clientID);
    resource.setClientSecret(clientSecret);
    resource.setAccessTokenUri(accessTokenUri);
    resource.setUserAuthorizationUri(userAuthorizationUri);
    resource.setScope(Arrays.asList("read"));

    return resource;
}
```

Que será el objeto a emplear cuando la aplicación Cliente desee acceder a información privada

```
@RestController
public class UserController {

    @Autowired
    private OAuth2RestOperations restTemplate;

    @Value("${config.oauth2.resourceURI}")
    private String resourceURI;

    @RequestMapping("/")
    public JsonNode home() {
        return restTemplate.getForObject(resourceURI, JsonNode.class);
    }
}
```

Implementacion Servidor de Recursos OAuth con Spring

Como en el Servidor de OAuth, son necesarias las dependencias

```
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-jwt</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.security.oauth</groupId>
  <artifactId>spring-security-oauth2</artifactId>
</dependency>
```