Microservicios con Spring

Victor Herrero Cazurro

Contenidos

¿Que es Spring?

Framework para el de desarrollo de aplicaciones java.

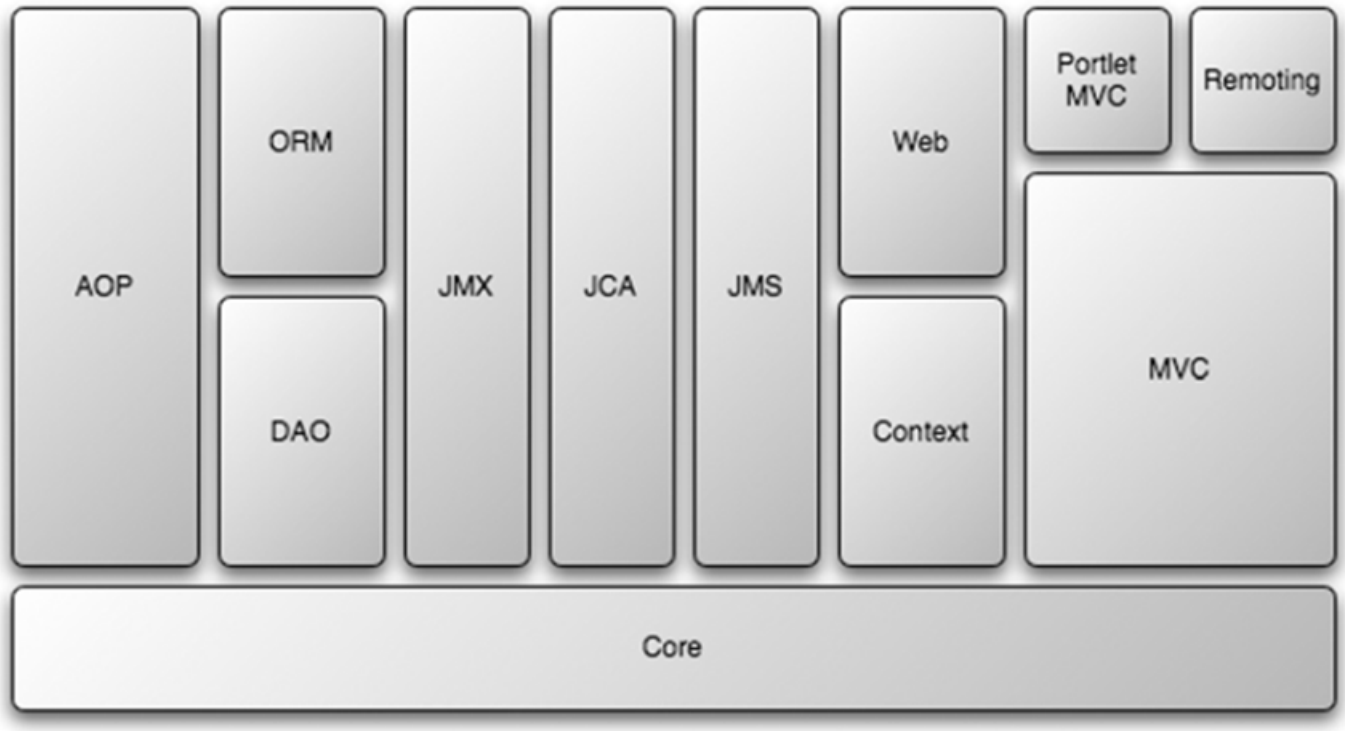
Es un contenedor ligero de POJOS que se encarga de la creación de beans (Factoría de Beans) mediante la Inversión de control y la inyección de dependencias.

Creado en 2002 por Rod Johnson

Spring se centra en proporcionar mecanismos de gestión de los objetos de negocio.

Spring es un framework idóneo para proyectos creados desde cero y orientados a pruebas unitarias, ya que permite independizar todos los componentes que forman a arquitectura de la aplicación.

Esta estructurado en capas, puede introducirse en proyectos de forma gradual, usando las capas que nos interesen, permaneciendo toda la arquitectura consistente.



IoC

Patrón de diseño que permite quitar la resposabilidad a los objetos de crear aquellos otros objetos que necesitan para llevar a cabo una acción, delegandola en otro componente, denominado Contedor o Contexto.

Los objetos simplemente ofrecen una determinada lógica y es el Contenedor que el orquesta esas logicas para montar la aplicación.

Inyección de dependencias

Patrón de diseño que permite desacoplar dos algortimos que tienen una relación de necesidad, uno necesita de otro para realizar su trabajo.

Se basa en la definición de Interfaces que definan que son capaces de hacer los objetos, pero no como realizan dicho trabajo (implementación).

```
interface GestorPersonas {

    void alta(Persona persona);

}

interface PersonaDao {

    void insertar(Persona persona);

}
```

Y la definición de propiedades en los objetos, que representarán la necesidad, la dependencia, de tipo la Interface antes creada, asociado a un método de **Set**, inyeccion por setter y/o a un constructor, inyección por construcción, para proporcionar la capacidad de que un elemeno externo, el contenedor, inyecte la dependencia al objeto.

```
class GestorPersonasImpl {

    private PersonaDao dao;

    //Inyeccion por contruccion
    public GestorPersonasImpl(PersonaDao dao){
        this.dao = dao;
    }

    //Inyeccion por setter
    public setDao(PersonaDao dao){
        this.dao = dao;
    }

    public void alta(Persona persona){
        dao.insertar(persona);
    }

}
```

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 permtite 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 tambien 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 definiciendo 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 anotacion **@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

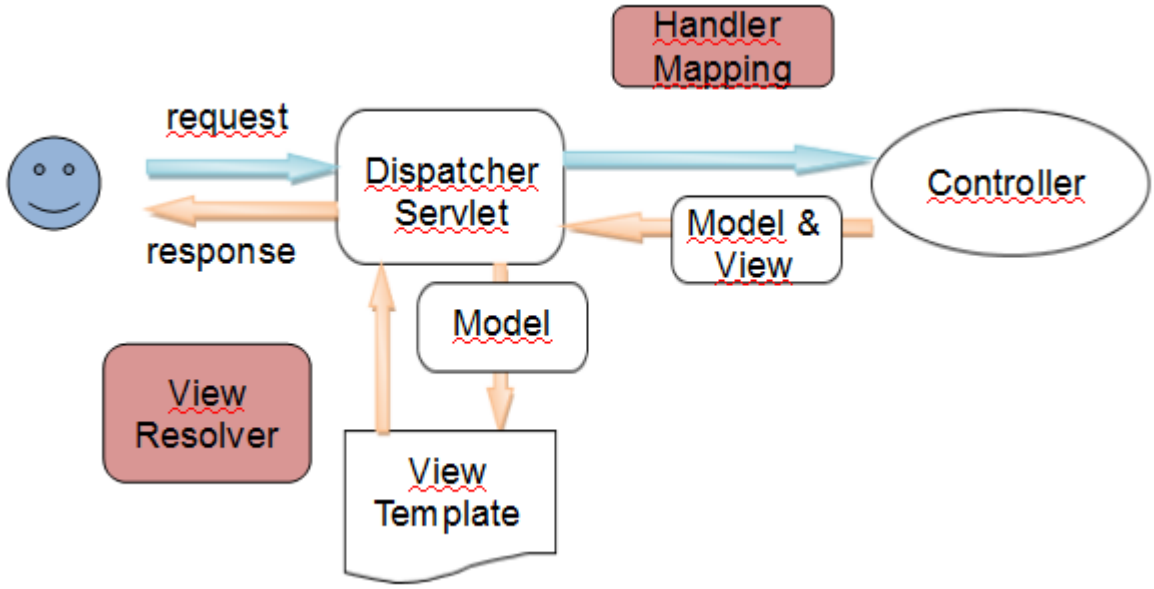
Spring MVC, como su nombre indica es un framework que implementa Modelo-Vista-Controlador, esto quiere decir que proporcionará componentes especializados en cada una de esas tareas.

Para incorporar las librerias 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/>
```

o

```
@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 Srping 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 sustituyéndo 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 descubrimineto 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 descubrimineto 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

Analogo 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

Tambien se puede asociar a la **session**, para ello se emplea la anotación **@SessionAttributes("nombreDelBeanDelModeloAAlmacenarEnLosAtributosDeLaSession")**, incluyendola 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 defirnir 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 raiz 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 resuesta a la petición procesada por Spring MVC.

Existen diversas implementaciones dependiendo de la tecnologia 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 libreria 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 PoiExcellView 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/reporte.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 libreria de etiquetas

```
<%@ taglibprefix="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 podria 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>
            <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 **modelAtribute**

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 librerias 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 etquetas 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 caracteristica 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 defido 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 porporciona 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 despues de que se ejecute la petición, permite manipular el objeto **ModelAndView** antes de pasarselo a la **View**.
- **afterCompletion()** –> Called after the complete request has finished. Seldom use, cant 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 emplearia la etiqueta del namespace **mvc**

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

Con JavaConfig, sobreescribiendo 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** → Compreba 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 idiomatico 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 configuracion

```
<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 tipologias complejas a formatos de representacion 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 retornanrlo en el cuerpo de la respuesta.
- **@RequestBody** → Indica que se debe leer el cuerpo de la peticion como un objeto cuyo tipo de representacion 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

```
@EnableWebMvc
```

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 sí 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 sí 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 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 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());
```

Rest

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

La palabra REST viene de

- **Representacion**: Permite representar los recursos en multiples formatos, aunque el mas habitual es JSON.
- **Estado**: Se centra en el estado del recurso y no en las operaciones que se pueden realizar con el.
- **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 anotacion **@RestController**, que auna las anotaciones **@Controller** y **@ResponseBody**, esta ultima 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 codigo de estado de HTTP.

Para incluir este codigo en las respuestas, se puede encapsular las entidades retornadas con **ResponseEntity**, el cual es capaz de representar tambien el codigo 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, peticion POST, se ha de retornar en la cabera **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** → Envia 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);
}
```