Microservicios con Spring

Victor Herrero Cazurro

Contenidos

1. ¿Que es Spring?

Framework para el de desarrollo de aplicaciones java.

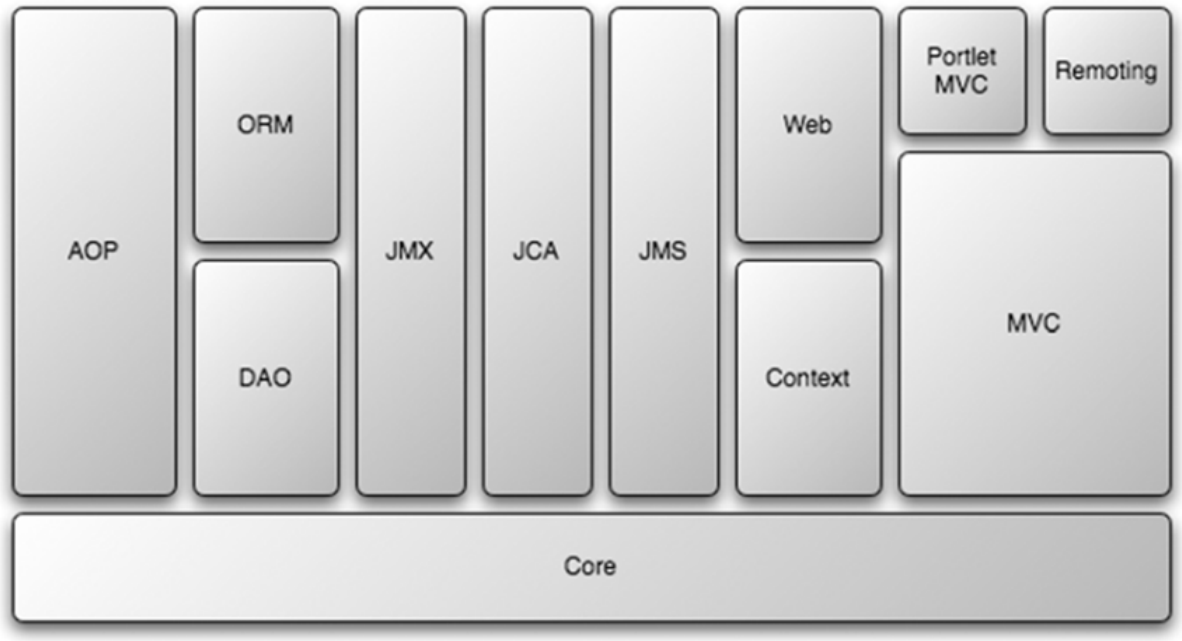
Es un contenedor ligero de POJOS que se encarga de la creación de beans (Factoría de Beans) mediante la Inversión de control y la inyección de dependencias.

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.



2. 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.

3. 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);
    }

}
```

4. Spring Boot

4.1. 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 librerias que ya están probadas trabajado 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 libreria 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

4.2. Introducción a Groovy

4.2.1. Reglas basicas

Es 100% compatible con java.

El **;** al final de cada sentencia es opcional, solo es necesario, si en una misma linea se ponen varias sentencias.

Acepta tipos dinamicos, por lo que no es necesario indicar los tipos de las variables/atributos/parametros/retornos de métodos, se puede emplear **def**.

```
class Clase {
    def metodo(parametros){
        def variable
    }
}
def instancia = new Clase
```

La visibilidad por defecto es **public**, luego se puede omitir dicha palabra.

No es necesario poner la palabra reservada **return**, si la firma del método establece que ha de retornar, el retorno es el valor de la última linea.

```
class Clase {
    String metodo(parametros){
        "Hola Mundo!"
    }
}
```

Los parentesis al invocar un método son opcionales.

```
def clase = new Clase
clase.metodo "Este es un parametro"
```

Se pueden definir clases como en java, pero tambien se pueden definir scripts, es decir codigo sin estar encerrado en una clase.

```
println "Hola Mundo!!!"
```

4.2.2. Tipos de Datos

No existen tipos primitivos, todos son objetos, por lo que el numero **4** será un objeto y no un primitivo y por tanto tendrá métodos asociados.

```
4.times {
    println "Esto se repetirá cuatro veces"
}
```

Se definen sintaxis especiales para definir listas y mapas basada en los **[]** y los **:**.

```
def lista = [1,2,3]
def mapa = ["clave": 3, "otra clave": "otro valor"]
```

Se permite acceder a los elementos de las colecciones como si fueran arrays

```
lista[0]
lista.get(0)

mapa["clave"]
mapa."clave"
mapa.clave
mapa.get("clave")
```

Las listas crecen de forma dinamica, no hay que realizar una reserva de espacio al inicializarlas.

```
def lista = []
lista[9] = "se situa en el decimo lugar dentro de la lista"
assert lista.size() == 10
```

Tambien se define un nuevo tipo de dato, los **rangos**, que permiten definir una sucesion de valores

```
println "Groovy"[0..3]
//Se imprime Groo

(0..9).each(num -> print num)
```

4.2.3. Closures

Se pueden definir **Closures**, que son funciones independientes reusables, que pueden ser pasados por parametro a otros funciones.

```
def number = 0
new File('data.txt').eachLine { line ->
    number++
    println "$number: $line"
}
```

Pueden tanto recibir parametros, como no hacerlo, pero en este último caso, recibiran siempre un parametro accesible con la variable **it**

```
def printout {print it}

(0..9).each printout
```

Si desde un lugar cualquiera del codigo se quiere invocar una closure, se ha de invocar el metodo **call**.

```
for (num in [0,1,2,3]) printout.call(num)
```

4.2.4. GString

Se puede acceder a las variables dentro de los **String** con el operador **$** (GString)

```
def saludar(nombre) {
    println "Hola $nombre!!!"
}
```

4.2.5. Expresiones regulares

Las expresiones regulares se definen entre **/**

```
def texto = "Groovy es un lenguaje de programacion"
assert texto =~ /ua/ //La expresion retorna true
```

Y se pueden emplear los operadores

- =~ búsqueda de ocurrencias (produce un java.util.regex.Matcher)
- ==~ coincidencias (produce un Boolean)
- ~ patrón

Se pueden redefinir operadores, redefiniendo metodos siguiendo las siguientes equivalencias

- El operador + equivale al metodo plus

- El operador - equivale al metodo minus
- El operador * equivale al metodo multiply
- El operador / equivale al metodo div

```
class Clase {
    Clase plus(Object other) {
        return this;
    }
}
```

4.3. 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 creear un poryecto 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**

4.3.1. Comandos

- init: Crea un proyecto de tipo **start.spring.io**

```
> spring init --dependencies=web --extract
```

- run

Partiendo del siguiente fichero **HolaMundo.groovy**

```
@RestController
class HolaMundoController {

    @RequestMapping("/")
    def saludar() {
        return "Hola Mundo!!!!"
    }

}
```

Se puede ejecutar con el comando

```
> spring run HolaMundo.groovy
```

- test

Partiendo del anterior fichero y añadiendo el siguiente

HolaMundoControllerTest.groovy

```
class HolaMundoControllerTest {
  @Test
  void pruebaHolaMundo() {
    def respuesta = new HolaMundoController().saludar()
    assertEquals("Hola Mundo!!!!", respuesta)
  }
}
```

Se pueden ejecutar las pruebas con el comando

```
> spring test HolaMundo.groovy HolaMundoControllerTest.groovy
```

- jar

```
> spring jar HolaMundo.jar HolaMundo.groovy
> java -jar HolaMundo.jar
```

- war

```
> spring war HolaMundo.war HolaMundo.groovy
> java -jar HolaMundo.war
```

- grab
- install
- uninstall
- shell: Permite acceder a una consola para ejecutar los comandos.

```
> spring shell
```

4.4. 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 dependedencia.

```
<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**

4.4.1. Aplicación Web

Para crear una aplicación web con Spring Web MVC, se ha de añadir la siguiente dependencia.

```
<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 unicamente necesitamos definir una clase anotada con **@SpringBootApplication**

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

Esta anotacion 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 unicamente 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);
    }
}
```

4.4.2. Aplicación de consola

Si se desea lanzar una serie de comandos en el proceso de arranque de la aplicación, tipicamente cuando se quiere realizar alguna demo de algún API, se puede implementar la interface **CommandLineRunner**

```
@SpringBootApplication
public class Application implements CommandLineRunner{

    private static final Logger logger = LoggerFactory.getLogger
(Application.class);

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

    @Override
    public void run(String... args) throws Exception {
        logger.info("Args : {}", args);
    }
}
```

O la interface **ApplicationRunner**

```
@SpringBootApplication
public class Application implements ApplicationRunner {

    private static final Logger logger = LoggerFactory.getLogger
(Application.class);

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

    @Override
    public void run(ApplicationArguments args) throws Exception {
        logger.info("Option names : {}", args.getOptionNames());
    }
}
```

4.5. 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, freemaker, velocity, jsp, …¬

4.5.1. 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 caracteristica 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

4.5.2. JSP

Para poder emplear **JSP** en lugar de **Thymeleaf**, hay dos opciones, la primera es definir el proyecto de Spring Boot como War en el pom.xml, definiendo la siguiente configuracion en el contexto de Spring

```
@SpringBootApplication
public class SampleWebJspApplication extends
SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure
(SpringApplicationBuilder application) {
        return application.sources(SampleWebJspApplication.class);
    }

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

}
```

y las siguientes propiedades en el fichero **application.properties**

```
spring.mvc.view.prefix: /WEB-INF/views/
spring.mvc.view.suffix: .jsp
```

NOTE	El directorio desde donde creacerá **WEB-INF**, sera **src/main/webapp**

La segunda opcion, será mantener el tipo de proyecto como Jar y añadir las siguientes dependencias al **pom.xml**

```
<dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-jasper</artifactId>
</dependency>
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
</dependency>
```

Por último indicar donde encontrar los ficheros mediante las siguientes propiedades en el fichero **application.properties**

```
spring.mvc.view.prefix: /WEB-INF/views/
spring.mvc.view.suffix: .jsp
```

NOTE	El directorio desde donde creacerá **WEB-INF**, sera **src/main/resources/META-INF/resources/**

4.5.3. 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.

4.5.4. 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>
..
```

4.6. 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 **HTTP** y/o **JMX**.

Para activarlo, es necesario incluir una dependencia

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

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

- Estado

http://localhost:8080/ListadoDeTareas/actuator/health

- Mapeos de URL

http://localhost:8080/ListadoDeTareas/actuator/mappings

- Descarga de estado de la memoria de la JVM

http://localhost:8080/ListadoDeTareas/actuator/heapdump

- Beans de la aplicacion

http://localhost:8080/ListadoDeTareas/actuator/beans

Por defecto casi todos los **endpoint** estas deshabilitados, para activarlos se tienen las propiedades

```
management:
  endpoint:
    <id>:
      enabled: true
```

Tambien se puede configurar la visibilidad de los **endpoint** a traves de **HTTP** y/o **JMX** con las propiedades

Valores por defecto para la exposicion de servicios de actuator

```
management:
  endpoints:
    jmx:
      exposure:
        include: "*"
        exclude:
    web:
      exposure:
        include: info, health
        exclude:
```

Además se puede configurar el acceso, si en el classpath esta presente **spring-security** los endpoints serán seguros por defecto.

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

Para controlar quien puede acceder se deberá definir una regla **requestMatcher**, ofreciendo el API un método para hacer referencia a los endpoint de actuator

```
http.requestMatcher(EndpointRequest.toAnyEndpoint()).authorizeRequest()
.anyRequest().hasRole("ENSDPOINT_ADMIN")
```

4.6.1. 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 anotado con **@Endpoint**, **@JmxEndpoint** o **@WebEndpoint** y sus métodos expuestos, anotados con **@ReadOperation**, **@WriteOperation** o **@DeleteOperation**

```
@Component
@Endpoint
public class ListEndPoints{

    private List<Endpoint> endpoints;

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

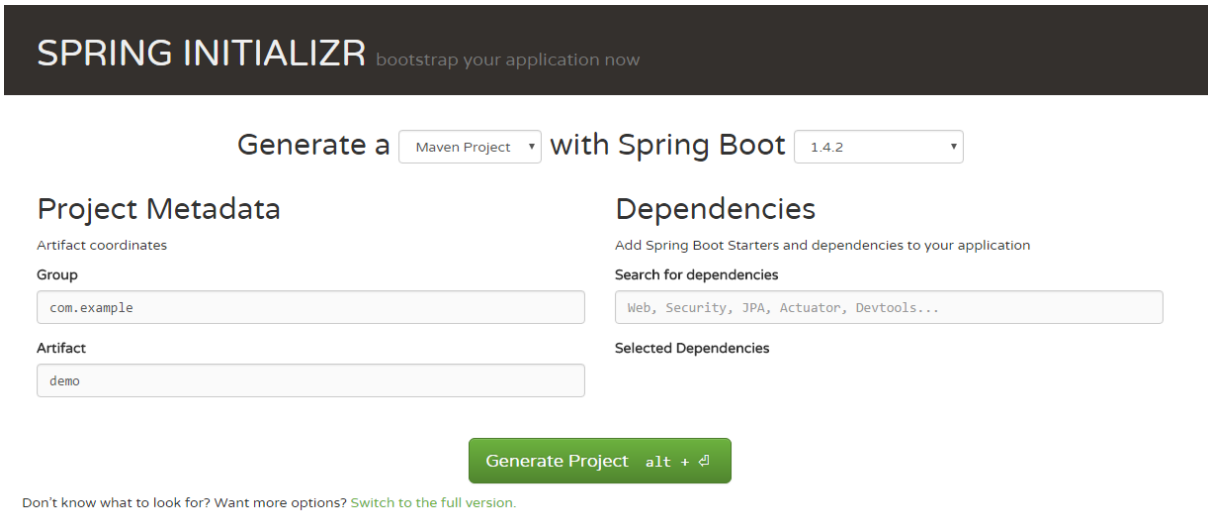
4.7. Uso de Java con start.spring.io

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.



Lo que se ha de proporcionar es

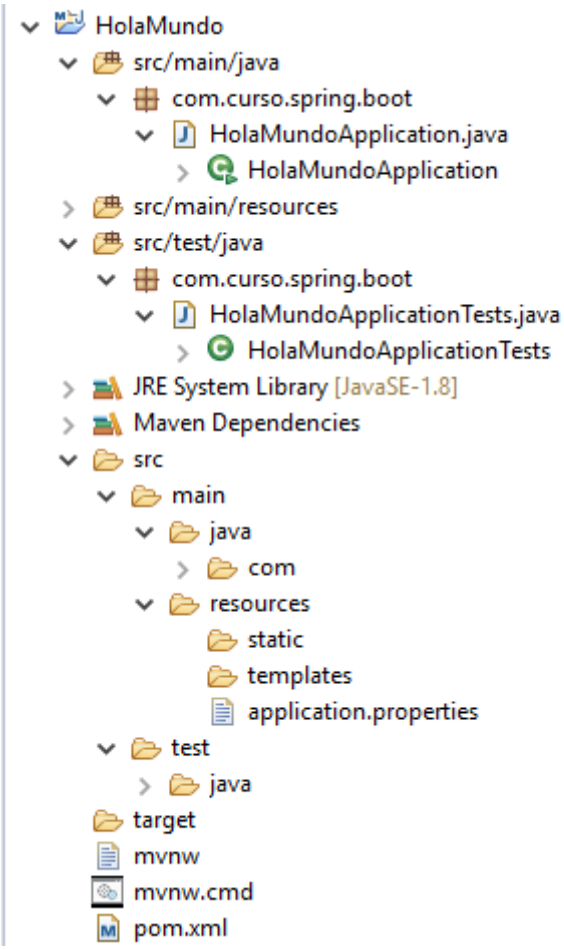
- Tipo de proyecto (Maven o Gradle)
- Versión de Spring Boot
- GroupId
- ArtifactId
- Dependencias

Existe una vista avanzada donde se pueden indicar otros parametros como

- Versión de java
- El tipo de packaging
- El lenguaje del proyecto

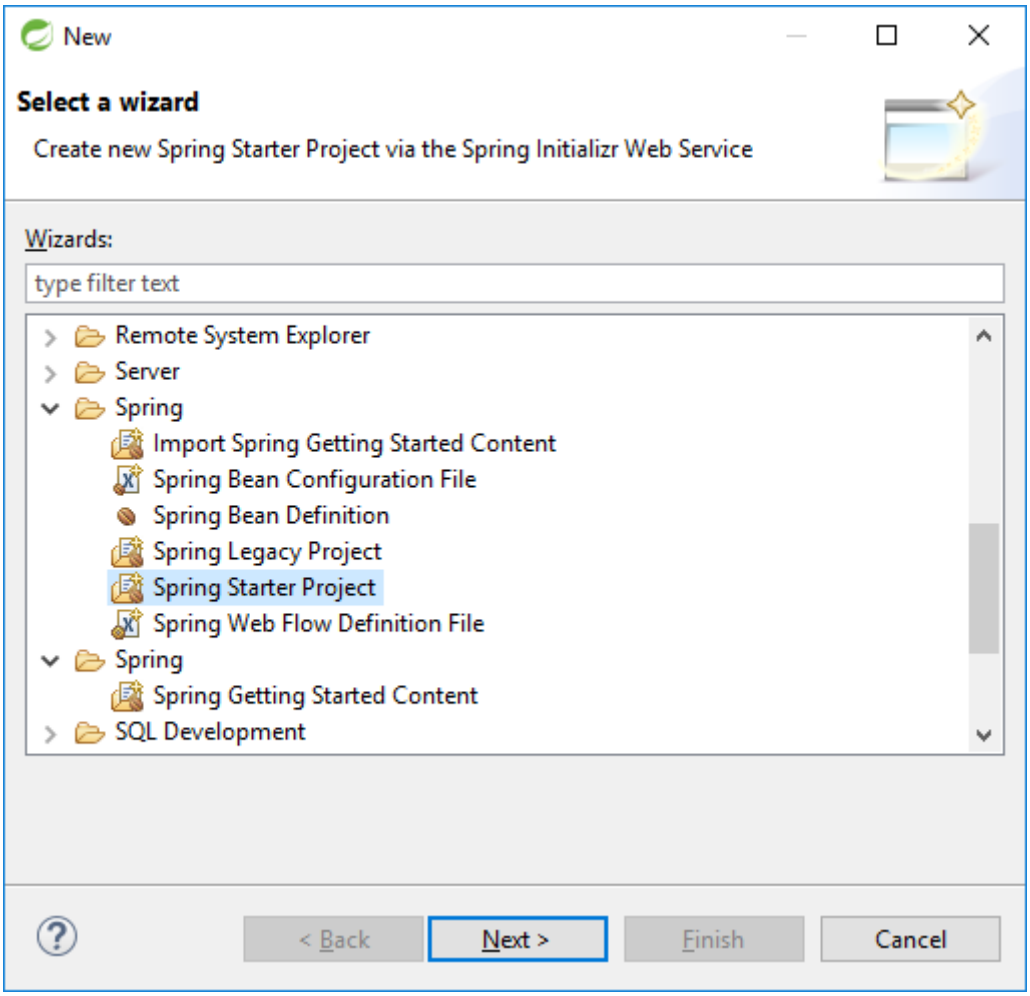
- Seleccion mas detallada de las dependencias

La estructura del proyecto con dependencia web generado será

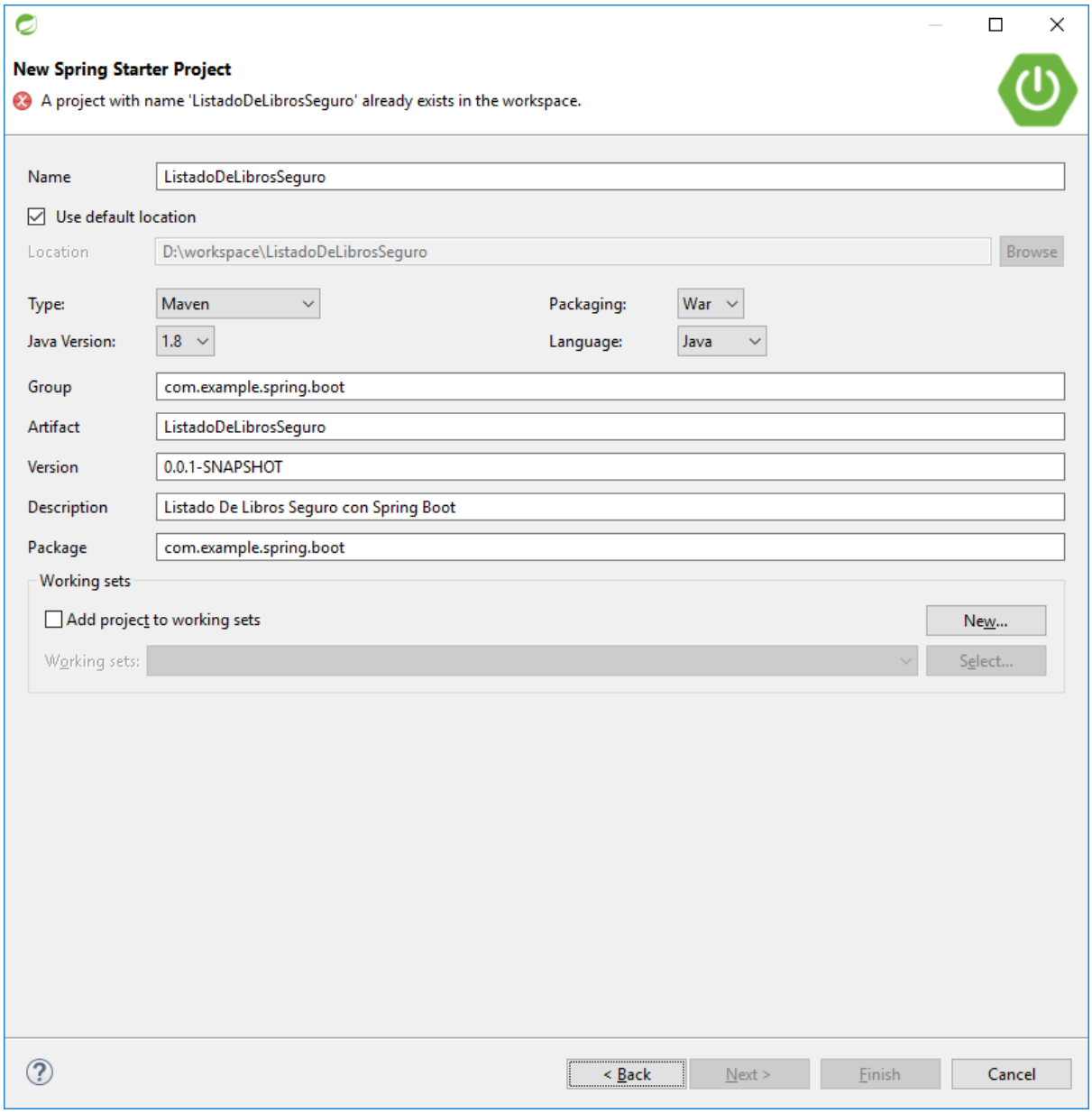


En esta estructura, cabe destacar el directorio **static**, destinado a contener cualquier recurso estatico de una aplicación web.

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 opcion, se muestra un formulario similar al de la web



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
```

4.8. Starters

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

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 librerias, 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'
}
```

4.9. Soporte a propiedades

Spring Boot permite configurar unas 300 propiedades, aqui una lista de ellas.

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

- Argumentos de la linea 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 **.propertes** 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.

4.9.1. Configuracion 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 aplicacion (Ej: /resource).
- **server.ssl.key-store**: Ubicación del fichero de certificado (Ej: file:///path/to/mykeys.jks).
- **server.ssl.key-store-password**: Contraseña del almacen.
- **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
```

4.9.2. Configuracion 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**: Ubicacion del fichero de log (Ej: /var/logs/)
- **logging.file**: Nombre del fichero de log (Ej: miApp.log)

4.9.3. Configuracion 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 conectra 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

4.9.4. 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 opcion para emplear propiedades, es el uso de la anotacion **@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;
```

4.10. 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;
}
```

Tambien se puede definir un conjunto de propiedades que solo se empleen si un perfil esta acivo, 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 el 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 applicacion-0.0.1-SNAPSHOT.jar
--spring.profiles.active=production
```

TIP De definirse mas de un perfil activo, se indicaran con un listado separado por comas

4.11. JPA

Al añadir el starter de JPA, por defecto Spring Boot va a localizar todos los Bean detro 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 subapaquetes, se puede referenciar con **@EnableJpaRepositories**

Cuando se emplea JPA con Hibernate como implementación, esté 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.

Habrá 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 depedencias no es necesario que se indiquen en Spring Boot, ya que vienen predefinidas en el **parent**

Para configurar un nuevo origen de datos, se indican las siguientes propiedades

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

4.12. Mongo

Para trabajar con Mongo se puede recurrir a contenedores de cocker, pudiendo levantar uno con mongo con el siguiente comando

```
> docker run -d --rm -p 27017:27017 mongo
```

Framework que extiende las funcionalidades de Spring ORM, permitiendo definir **Repositorios** de forma mas sencilla, sin repetir código, dado que ofrece numerosos métodos ya implementados y la posibilidad de crear nuevos tanto de consulta como de actualización de forma sencilla.

El framework, se basa en la definición de interfaces que extiendan la jerarquia de **MongoRepository**, concretando el tipo entidad y el tipo de la clave primaria.

```
public interface CustomerRepository extends MongoRepository<Customer,
BigInteger> {}
```

La clase entidad a la que se hace referencia, en el ejemplo **Customer**, será una clase donde:

- La clave primaria estará anotada con **org.springframework.data.annotation.Id**
- Se definirá el constructor por defecto.
- Si se desea modificar el nombre de la coleccion de **Mongo** donde se almacenaran los documentos de tipo **Customer**, se deberá emplear la anotación **org.springframework.data.mongodb.core.mapping.Document**, que es opcional, dado que por defecto la colección se llamará como la clase.

```
@Document
public class Cliente {
    @Id
    private String id;
    private String firstName;
    private String lastName;
}
```

Se pueden añadir más anotaciones para realizar un mapeo mas preciso, las mas basicas son

- **@Field**: Permite renombrar y ordenar los campos.

```
    @Field("nombre")
    private String firstname;
```

- **@Indexed**: Permite establecer constraints, entre otras cosas la unicidad del campo.

```
    @Indexed(unique = true)
    private Email email;
```

- **@DBRef**: Permite establecer una relación con otra entidad.

```
@Document
public class Pedido {
    @Id
    private BigInteger id;
    @DBRef
    private Cliente cliente;
}
```

- **@PersistenceConstructor**: Permite indicar al API que use un constructor distinto al por defecto.

```
    @PersistenceConstructor
    public Pedido(Cliente cliente, Direccion direccionFacturacion,
Direccion direccionEntrega) {
        super();
        this.cliente = cliente;
        this.direccionFacturacion = direccionFacturacion;
        this.direccionEntrega = direccionEntrega;
    }
```

Con **Spring Boot**, habrá que añadir el starter de **Mongo**, con este starter en el classpath **Spring Boot** va a localizar todos los Bean detro del paquete y subpaquetes donde se encuentra la clase anotada con **@SpringBootApplication** en busca de interfaces Repositorio, que extiendan la interface **MongoRepository**, de no encontrarse la interface que define el repositorio dentro del paquete o subapaquetes, se puede referenciar con **@EnableMongoRepositories**

4.12.1. Querys

Con el uso de los repositorios de **Data mongo**, se obtienen varias funcionalidades de forma directa, veamos algunas de ellas.

- Inserciones

```
Customer customer = new Customer();
customer.setName("Victor");
customerRepository.insert(customer);
customerRepository.save(user);
```

- Actualizaciones

```
customer = customerRepository.findOne(1);
customer.setName("Pedro");
customerRepository.save(customer);
```

- Borrado

```
customerRepository.delete(customer);
```

- Busqueda

```
userRepository.findOne(user.getId())

List<User> users = userRepository.findAll(new Sort(Sort.Direction.ASC,
"name"));
```

- Existencia

```
boolean isExists = userRepository.exists(user.getId());
```

- Paginacion

```
Pageable pageableRequest = PageRequest.of(0, 1);
Page<User> page = userRepository.findAll(pageableRequest);
List<User> users = pages.getContent();
```

4.13. 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.

4.14. 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, ..

4.15. Soporte Mensajeria JMS

Para emplear JMS de nuevo Spring Boot, proporciona un starter, en este caso para varias tecnologias: 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 inicar 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**

Se puede utilizar un contenedor de Docker, para arrancarlo se puede emplear la siguiente sentencia

```
docker run --name=activemq -d -e
'ACTIVEMQ_CONFIG_DEFAULTACCOUNT=true' -e
'ACTIVEMQ_CONFIG_TOPICS_topic1=mailbox' -p 8161:8161 -p
61616:61616 -p 61613:61613 webcenter/activemq:5.14.3
```

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**.

4.15.1. 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**

4.15.2. Productores

Para definir un Bean que envie 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

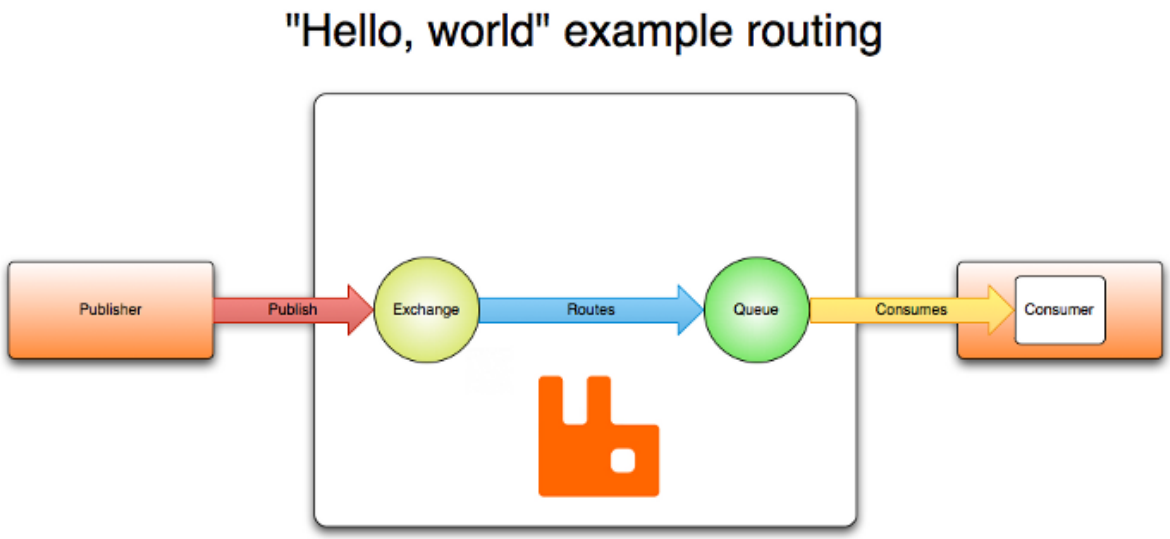
```
@Component
public class Receiver {
    @JmsListener(destination = "mailbox", containerFactory = "
myFactory")
    public void receiveMessage(Email email) {
        System.out.println("Received <" + email + ">");
    }
}
```

4.16. Soporte Mensajeria AMQP

AMQP es una especificacion de mensajeria asincrona, 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 (queue y topic).



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**.

Otra opcion es emplear un contenedor de Docker

NOTE

Sentencia de creación de contenedor con rabbitmq y con plugin de gestion web habilitado

```
> docker run --name=rabbitmq -d -p 5672:5672 -p
15672:15672 rabbitmq:3.7.8-management-alpine
```

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

El puerto 15672, es el puerto donde escucha la herramienta de gestion

4.17. Receiver

Los **Receiver** son los que recibiran y procesaran los mensajes que se dejan en el bus.

Para definirlos se ha de incluir la dependencia

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

NOTE

Ojo que en el servicio **start.spring.io** se define la dependecia como **RabbitMQ**

Y añadir un **Bean** al contexto de Spring que haga de **Receiver**, no tiene porque implementar ningun API, solamnete recibir por parametro el tipo de objeto que se deja en el bus.

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

Una vez definido el bean que procesara los mensajes, se han de definir una serie de beans extras que permiten configurar como ese bean recibe los mensajes, los beans a configurar son

- **MessageListenerAdapter**: Que indica que el bean anterior recibe mensajes y que el método **receiveMessage** será el que se ejecutará.

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

- **Queue**: Bean que modela el componente dentro del bus AMQP que almacena los mensajes

```
@Bean
public Queue queue() {
    return new Queue("spring-boot", false);
}
```

- **Exchange**: Bean que modela el componente dentro del bus amqp que recibe los mensajes y los propaga a los **queue**.

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

- **Binding**: Bean que se encarga de asociar los queue y los exchange, definiendo una expresión para filtrar los mensajes que le llegan al exchange que ha de direccionar al queue, en este caso la expresion es **foo.bar.#**

```
@Bean
public Binding binding(Queue queue, TopicExchange exchange) {
    return BindingBuilder.bind(queue).to(exchange).with("foo.bar.#");
}
```

- **ConnectionFactory**: Bean que asocia el receiver al queue, para que se escuchen los mensajes que llegan al queue.

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

4.18. Producer

Es el que se encarga de generar y enviar los mensajes al exchange.

Para producir nuevos mensajes, se emplea la clase **RabbitTemplate**, que permite enviar mensajes con métodos **convertAndSend**, para lo cual se indican el nombre del **exchange** y una expresion para poder canalizar el mensaje al queue deseado.

```
rabbitTemplate.convertAndSend("spring-boot-exchange", foo.bar.baz",
"Hello from RabbitMQ!");
```

5. Spring MVC

5.1. Introduccion

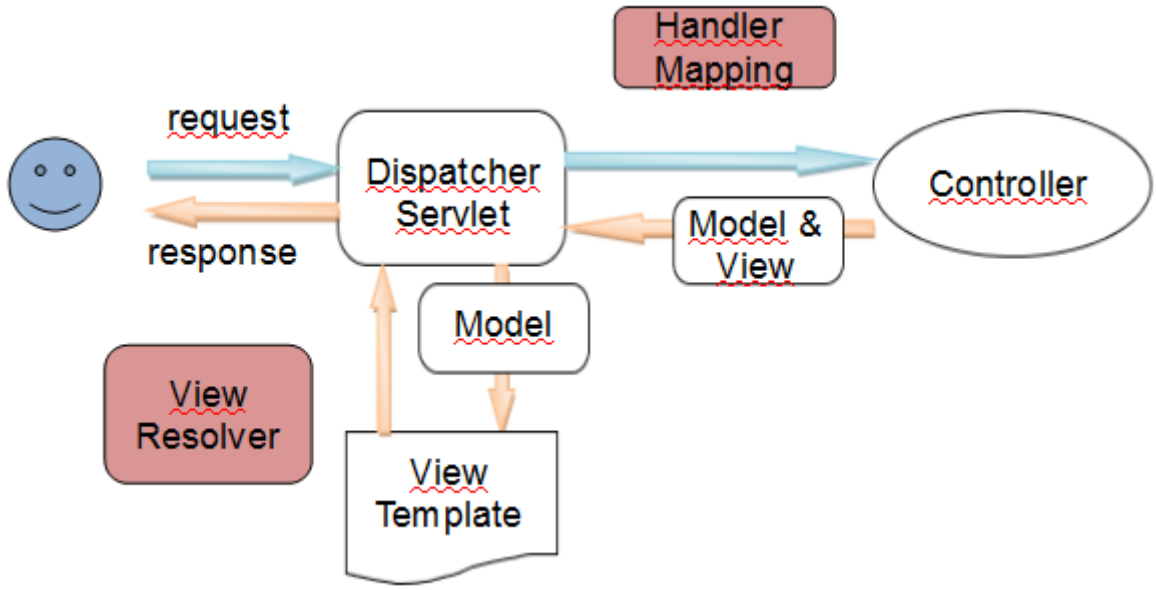
Spring MVC, como su nombre indica es un framework que implementa Modelo-Vista-Controlador, esto quiere decir que proporcionará componentes especializados en cada una de esas tareas.

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>
```

5.2. 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**.



5.2.1. 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("/");
    }

    private AnnotationConfigWebApplicationContext getContext() {
        AnnotationConfigWebApplicationContext context = new
AnnotationConfigWebApplicationContext();
        context.setConfigLocation(this.getClass().getPackage().getName
());
        return context;
    }
}
```

5.2.2. 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**

5.3. 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"
```

5.4. 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 dentro de **src/main/webapp** 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.

5.4.1. 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();
    }
}
```

5.5. 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(ViewControllerRegistry 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**

5.6. 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**

5.6.1. 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" />
```

5.6.2. 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" />
```

5.6.3. ControllerClassNameHandlerMapping

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

5.6.4. DefaultAnnotationHandlerMapping

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

5.6.5. 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/**

5.7. 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**

5.7.1. @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)

Si se desea que el retorno provoque una redireccion, basta con incluir el prefijo **redirect:**

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

NOTE Cuando se retorna el id de una View, se hace participe a esa View de la actual Request, cuando se redirecciona, se crea una Request nueva.

Y puede recibir como parámetro

- **Model**: Datos a emplear en la **View**.
- **Map<String, Object> model**: Lo resuelve como Model, permite el desacoplamiento con el API de Spring.
- 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.

- Parametros anotados con @**CookieValue**: Dato que llega en un HTTP cookie
- Parametros anotados con @**RequestHeader**: Dato que llega en un HTTP Header
- Parametros anotados com @**SessionAttribute**: Atributo de la Sesion Http que se desea inyectar en el controlador
- **HttpServletRequest**
- **HttpServletResponse**
- **HttpSession**
- **Locale**
- **Principal**
- Parametros anotados com @**Validation**: Aplica las reglas de validacion sobre el parametro.
- **Errors**: Resultado de aplicar las reglas de validacion sobre los parametros.
- **BindingResult**
- **UriComponentsBuilder**

5.7.2. 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

5.7.3. @RequestMapping

Es la anotacion que permite resolver si la **HttpRequest** llega o no a un controlador, define un filtro de seleccion de Controladores basado en las caracteristicas del Request.

Se pueden definir

- **method**: Method HTTP
- **path**: Url
- **consumes**: Corresponde a la cabecera Content-Type
- **produces**: Corresponde a la cabecera Accept
- **params**: Permite indicar la obligatoriedad de la presencia de un parametro (params="myParam"), asi como de su ausencia (params="!myParam") o un valor determinado (params="myParam=myValue")
- **headers**: Igual que la anterior pero para cabeceras

5.7.4. @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) { // …¬ }

5.7.5. @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**

5.7.6. @SessionAttribute

Anotacion que permite recibir en un método de controlador, un atributo insertado con anterioridad en la Sesion.

```
@GetMapping("/nuevaFactura")
public String nuevaFactura(@SessionAttribute("login") Login login,
@ModelAttribute Factura factura) {
    return "factura/formulario";
}
```

5.7.7. @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>
```

5.7.8. @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.

5.7.9. @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");
    }
}
```

5.7.10. @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";
}
```

5.7.11. @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) {

}
```

5.7.12. @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;
}
```

5.7.13. @ControllerAdvice

Permiten definir en una clase independiente configuraciones de **@ExceptionHandler**, **@InitBinder** y **@ModelAttribute** que afectaran a los controladores que se desee, siempre que sean procesados por **RequestMappingHandlerMapping**, por ejemplo los **ViewControllers** no se ven afectados por esta funcionalidad.

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

5.8. 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. Aunque es configurable, emplea por defecto las **View** de tipo **InternalResourceView**, de emplearse **JstlView**, se necesitaria añadir al classpath la dependencia con **jstl**
- **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.

5.8.1. 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>
```

5.8.2. 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"/>
```

5.8.3. 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.Jasp
erReportsPdfView
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**

5.9. 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

5.9.1. 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");
    }
}
```

5.9.2. 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);
    }
}
```

5.9.3. 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.JasperReportsPdfVie
w">
    <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";
    }
}
```

5.9.4. 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.

5.10. 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";
}
```

Si se desea recibir un fichero desde el cliente, se empleará la tipologia **CommonsMultipartFile[]**

```
@RequestMapping(value = "/uploadFiles", method = RequestMethod.POST)
public String handleFileUpload(@RequestParam CommonsMultipartFile[]
fileUpload) throws Exception {
    for (CommonsMultipartFile aFile : fileUpload){
        // stores the uploaded file
        aFile.transferTo(new File(aFile.getOriginalFilename()));

    }
    return "Success";
}
```

5.10.1. 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:textarea path=""/>**:
- **<form:password path=""/>**:
- **<form:radiobutton path=""/>**:
- **<form:radiobuttons path=""/>**:
- **<form:select path=""/>**:
- **<form:option value=""/>**:
- **<form:options/>**:
- **<form:button/>**:
- Core

- **<spring:argument/>**:
- **<spring:bind path=""/>**:
- **<spring:escapeBody/>**:
- **<spring:eval expression=""/>**:
- **<spring:hasBindErrors name=""/>**:
- **<spring:htmlEscape defaultHtmlEscape=""/>**:
- **<spring:message/>**:
- **<spring:nestedPath path=""/>**:
- **<spring:param name=""/>**:
- **<spring:theme/>**:
- **<spring:transform value=""/>**:
- **<spring:url value=""/>**:

5.10.2. 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" />
```

5.10.3. 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, dado que por defecto un objeto definido como **ModelAttribute** se situa en **HttpServletRequest** que es donde se ira a buscar al renderizar la JSP del formulario.

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

5.11. 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;
}
```

5.11.1. 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;
```

5.11.2. Anotaciones JSR-303

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

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

5.11.3. 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.

5.12. 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.ReloadableResourceBundleMessageSou
rce">
    <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>"/>
```

5.13. 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.

5.13.1. 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"/>
```

5.13.2. 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" />
```

5.14. 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
```

5.15. 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
```

5.15.1. 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).

5.15.2. 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.MappingJackson2HttpMessageConv
erter"/>
    </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());
```

6. 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**.

6.1. 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**.

6.2. 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);
}
```

6.3. 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)
```

6.4. 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);
}
```

7. Spring Cloud

7.1. ¿Que son los 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.

Las aplicaciones modernas presentan habitualmente la necesidad de ser accedidas por diversos tipos de clientes, browser, moviles, aplicaciones nativas, para ello deben implementar un API para ser consumidas, además de para ser integradas con otras aplicaciones a traves de servicios web o brokers de mensajeria.

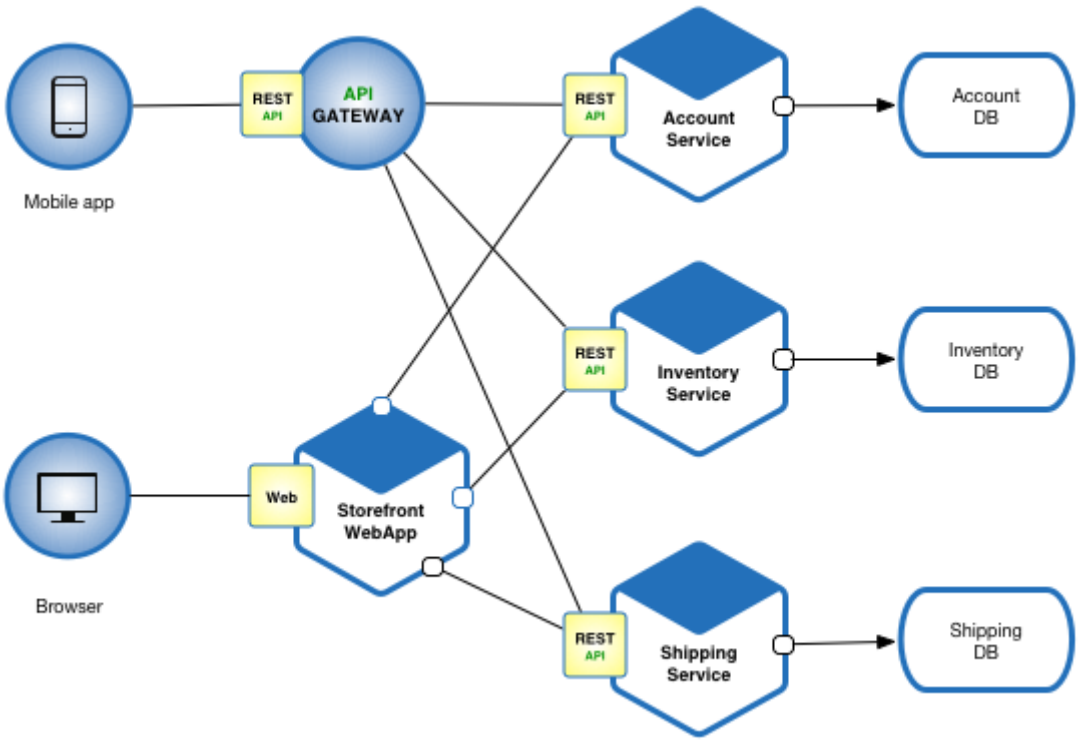
Tambien se busca que los nuevos miembros del equipo puedan ser productivos rapidamente, para lo cual se necesita que la aplicacion sea facilmente comprensible y modificable.

En ocasiones se aplican tecnicas de desarrollo agil, realizando despligue continuo, para lo cual la aplicacion debe poder pasar de desarrollo a producción de forma rapida.

Generalmente las aplicaciones necesitan ser escaladas y tienen criterios de disponiblidad, por lo que se necesitan ejecutar multiples copias de la aplicación.

Y siempre es interesante poder aplicar las nuevas tendencias en frameworks y tecnologias qeu mejoran los desarrollos, por lo que es interesante que los nuevos desarrollos no se vean obligados a seguir tecnologias de los antiguos.

Por estos motivos, será interesante aplicar una arquitectura con bajo acoplamiento y servicios colaborativos, el bajo acoplamiento lo conseguiremos empleando servicios HTTP y protocolos asincronos como AMQP y haciendo que cada microservicio tenga su propia base de datos.



7.2. Ventajas de los Microservicios

- Son ligeros, desarrollados con poco codigo.
- Permiten aprovechar de forma mas eficiente los recursos, ya que al ser cada

microservicio una aplicación en sí, se pueden aumentar los recursos de dicha funcionalidad sin tener que aumnetar los recursos de otras piezas que quizas no los necesiten, por lo que los recursos son asignados de forma más granular.

- Permiten emplear tecnologias distintas, aprovechando las ventajas puntuales de cada una de ellas, sin que esas ventajas puedan lastrar otros componentes.
- Permiten realizar despliegues más rapidos, ya que evoluciones que se produzcan en un microservicio, al ser independientes no deben afectar a otras piezas del puzle y por tanto según esten terminados se pueden poner en producción, sin tener que esperar a hacer un despliegue de una version completa de toda la aplicación, esto unido a que los microservicios son **pequeños**, hace que el periodo desde que se comienza a desarrollar la mejora hasta la puesta en producción, sea corto y por tanto facilite la aplicación de **Continuous Delivery** (entrega continua).
- Mejoran el comportamiento de las aplicaciones frente a los fallos, ya que estos quedan mas aislados

7.3. Desventajas de los Microservicios

- Exigen mayor esfuerzo en el despliegue, control del versionado, copatibilidad entre versiones, actualización, monitorización, ..
- Es mas complejo realizar test de integracion.
- No existe un ambito transaccional entre todos los microservicios que participan.
- Con aplicaciones existentes, es mas complicado de aplicar.

7.4. Arquitectura de Microservicios

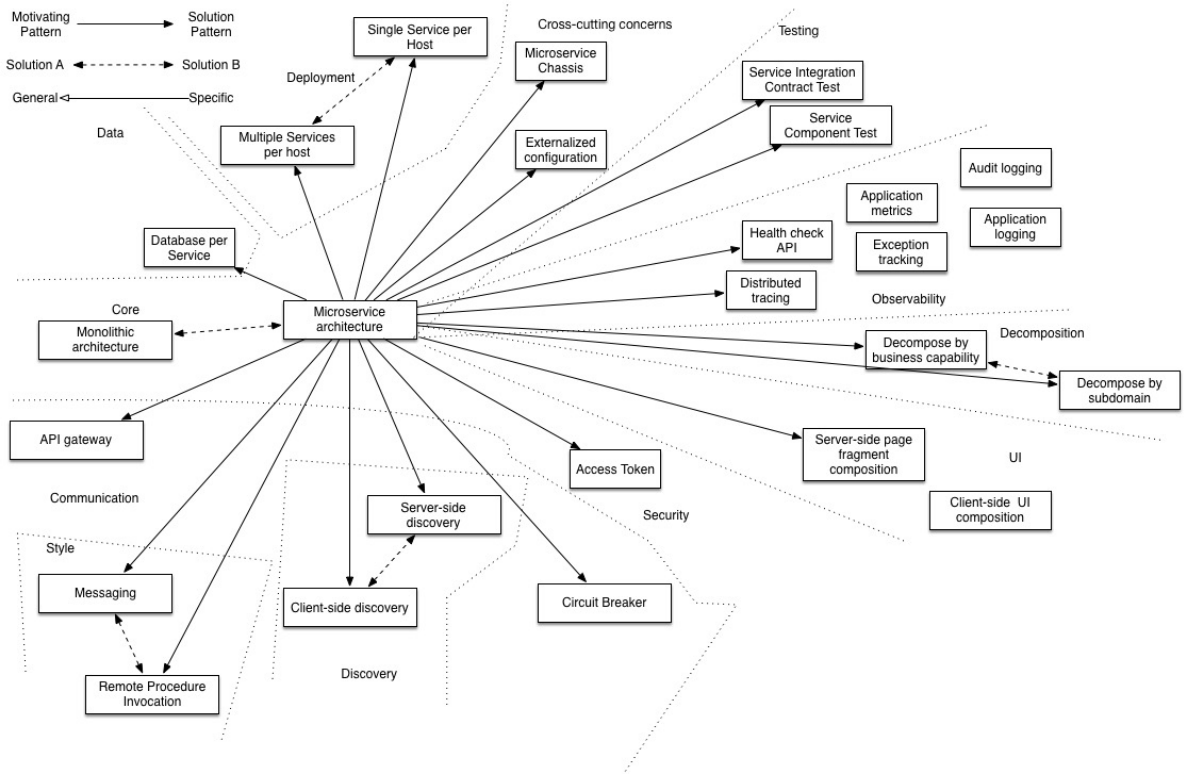
Cuando se habla de arquitectura de microservicios, se habla de **Cloud** o de arquitectura distribuida.

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

- Monitorización de la arquitectura,
- Configuración de los microservicios,
- Descubrimiento de microservicos,
- Detección de caida de microservicios,

- Balanceo de carga,
- Seguridad centralizada,
- Logs centralizados, ..

7.4.1. Patrones



Para solventarlos, se suelen emplear patrones, aqui tenemos algunos de los mas importantes agrupados

- **Principio de la responsabilidad unica**: Cada servicio debe tener responsabilidad sobre una sola parte de la funcionalidad total. La responsabilidad debe estar encapsulada en su totalidad por el servicio. Se dice que un **Servicio** debe tener solo una razón para cambiar.
- **Common Closure Principle**: Los componentes que cambian juntos, deben de estar empaquetados juntos, por lo que cada cambio afecta a un solo servicio.

Descomposicion

- **Descomposicion por capacidades de negocio**: Cada funcionalidad del negocio se encapsula en un unico sercivio. Por ejemplo: **Gestor del catalogo de productos**, **Gestor del inventario**, **Gestor de compras**, **Gestor de envios**, …¬
- **Descomposicion por acciones**: Descomponer la aplicacion en microservicios

por las acciones/casos de uso que ha de implementar. Por ejemplo: **Servicio de Envios**.

- **Descomposicion por recursos**: Descomponer la aplicacion en microservicios por los recursos que manejan. Por ejemplo: **Servicio de Usuario**.

Despliegue

- **Multiples servicios por servidor**: Se aprovechan mas los recursos, en cambio es mas dificil medir los recursos que necesita cada servicio, a parte de posibles conflictos en cuanto a los recursos compartidos.
- **Un Servicio por servidor**: Es mas sencillo redesplegar, gestionar y monitorizar el servicio, no existiendo conflictos con otros servicios, como pega es el no aprovechamiento optimo de los recursos, dado que el contenedor, consume parte de los recursos.
- **Un servicio por VM**: Facilita el despliegue dado que permite definir los requisitos del servicio de forma aislada, sin tener sorpresas, tambien permite gestionar aspectos del despliegue como memoria, CPU, …¬ El escalado es mas facil, como pega se tiene el tiempo que se tarda en montar la VM.
- **Un servicio por Contenedor**: Se trata de encapsular el despliegue del servicio dentro de un contenedor como Docker, los pros y contras son similares a los de la VM.
- **Serverless deployment**: Se basa en el empleo de una infraestructura de servicios, que abstrae el concepto de entorno de despliegue, basicamente se pone la aplicacion en el servicio y este reserva recursos y la pone en funcionamiento. Algunas son: AWS Lambda, Google Cloud Functions o Azure Functions.
- **Service deployment platform**: Se basa en el empleo de una plataforma de despliegue, que abstrae el servicio, porporcionando mecanismos de alta disponibilidad por nombre de servicio. Algunos ejemplo de plataformas son: Docker Swarm, Kubernetes, Cloud Foundry o AWS Elastic Beanstalk.

Cross cutting concerns (Conceptos Transversales)

- **Chassis**: Establece la conveniencia de usar una herramienta que gestione la configuracion de los aspectos trasversales del desarrollo como son: Externalizar configuraciones como credenciales de acceso o localizacion de los servicios como son Bases de datos o Brokers de mensajeria, configuracion del framework de Logging, Servicios de monitorizacion del estado de la aplicacion, Servicios de

metricas que permiten ajustar el rendimiento, Servicios de traza distribuida.

- **Externalizar configuraciones**: Establece la necesidad de externalizar la configuracion de tal forma que la aplicación pueda obtener su configuracion en la fase de arranque, para independizar su despliegue del entorno, que la aplicación no tenga que cambiar en cada despliegue. 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.

Formas de comunicacion

- **Remote Procedure Invocation**: Permite definir una interface comunicacion entre las aplicaciones clientes y los servicios, asi como entre los propios servicios cuando han de colaborar para satisfacer la peticion del cliente. La implementacion mas habitual es REST.
- **Messaging**: Permite la comunicacion asincrona y desacoplada entre los servicios. Algunas implementaciones son Apache Kafka y RabbitMQ.
- **Domain-specific protocol**: Describe el uso de un protocolo especifico para el dominio tratado, como puede ser SMTP para los correos electronicos.

Exponer APIs

- **API Gateway**: Es el punto de entrada para los clientes, este puede simplemente actuar como un enrutador o bien componer una respuesta basada en peticiones a varios servicios.
- **Backend for front-end**: Caso particular del anterior, donde se proporciona un *API Gatway para cada tipo de cliente.
- **Enrutador**: Permite exponer todos los servicios que los clientes necesitan, independientemente del tipo de cliente.
- **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.

Descubrir Servicios

- **Service registry**: Servicio que mantiene las ubicaciones de las distintas instancias de los microservicios y que es consultable.

- **Client-side discovery**: La forma en la que un cliente (API Gateway u otro mecroservicio) obtiene la referencia a un microservicio, es a traves del servicio de registro, donde todas las instancias de todos los microervicios se han de registrar para que el resto los encuentre.
- **Server-side discovery**: El cliente no accede directamente al servicio, sino que lo accede a traves de un enrutador, que consulta al servicio de registro.
- **Self registration**: Capacidad de los servicios para registrarse en el servicio de registro de forma autonoma, para quitar responsabilidad y complejidad al servicio de registro.
- **3rd party registration**: Cuando el registro lo realiza una herramienta independiente.

Confiabilidad (Reliability)

- **Control de ruptura de comunicación con los servicios (CircuitBreaker)**: Permite controlar que la caida de un microservicio consultado, no provoque la caida de los microservicios que realizan la consulta, proporcionando un resultado estatico para la consulta.

Gestion de Datos

- **Database per Service**: Cada servicio tiene su propia base de datos, que solo es accesible mediante el API que proporciona el servicio, para ello se pueden tener tablas privadas por servicio, esquemas por servicio o incluso base de datos por servicio. Con esta aproximación se consigue un bajo acoplamiento entre servicios, cada servicio puede emplear la tecnologia de persistencia mas adecuada. Existen dos desventajas principalmente, la no posibilidad de trabajar con transacciones distribuidas, que se puede paliar con el patrón **Saga** y la complejidad de implementar consultas con **joins**.
- **Shared database**: Al contrario que la anterior, indica compartir una misma base de datos por todos los microservicios, proporciona posibilidad de realizar transacciones tradicionales y es mas facilmente de operar. Como desventajas estan la necesidad de que los equipos coordinen cambios en el esquema, problemas de rendimiento al acceder tantos microservicios a la misma base de datos y que algunos microservicios no puedan adaptar sus necesidades a esa situación.
- **Patrón Saga**: Permite mantener la consistencia de los datos, que queda afectada por el echo de que cada microservicio tiene su propia base de datos y

no es posible aplicar transacciones distribuidas. La idea es que cuando el servicio ve afectado sus datos, envia un evento que permite al resto actualizar sus datos.

- **API Composition**: Describe la creación de un nuevo componente (servicio), que se encarga de consultar a cada servicio propietario de los datos, realizando un join en memoria. Un API Gateway puede realizar esta operacion.
- **CQRS (Command Query Responsibility Segregation)**: Describe la separación de las operaciones sobre los datos en dos partes **command** (actualizan el estado) y **query** (consultan el estado). Esto permite aprovechar dos ventajas, la primera es poder escalar independientemente las consultas y la administracion, generalmente las consultas exigen muchos mas recursos y la segunda poder emplear una tecnologia mas adecuada para cada caso, por ejemplo una RDBS para la administracion y una NoSQL para las consultas. Se habrá de implementar un sistema que permita la sincronización de los sistemas, de tal forma que el sistema de las **query** se mantenga actualizado, para ello una de las aproximaciones mas empleadas es **Event Sourcing**, que permite que dicha actualizacion de provoque bajo la subscripcion a los eventos generados por los **command**.
- **Event sourcing**: Describe como se persiste el estado de una entidad, guardando un listado de eventos que han llevado a la entidad al estado actual, recomoniendo el estado cada vez que es necesario, para mejorar el rendimiento ante entidades que modifican su estado con asiduidad, se realizan snapshot que sirven para recomponer el estado aplicando unicamente los ultimos eventos.
- **Application events**: Describe como se insertan registros en una tabla **events** por cada operacion y un proceso los publica en un **message broker**.
- **Traza de seguimiento de transacciones (Transaction log tailing)**: Describe la publicacion del log generado por cada transaccion como evento para trasladar los cambios al resto de microservicios.

Seguridad

- **Access Token**: Define la autenticación centralizada en el **API Gateway**, ya que es el que interaciona con el cliente, que obtiene un **token** que traslada al resto de microservicios para que estos puedan asegurar que el cliente que realiza la petición tiene permisos para invocarlos.

Observacion

- **Gestión centralizada de logs**: Define la generación de una traza formada por todas las trazas generadas por los microservicios participantes en una petición.
- **Metricas de aplicacion**: Define la creación de servicios que permiten obtener el estado de los microservicios.

7.4.2. Orquestacion vs Coreografia

Se habla de **Orquestación**, cuando una aplicacion, gestiona como invocar a otras aplicaciones, estableciendo los criterios y orden de invocacion (API Gateway).

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

7.5. Servidor de Configuracion

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

El servidor se conecta a un repositorio **git** de donde saca las configuraciones que expone, lo que permite versionar facilmente 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/victorherrerocazurro/config-cloud
spring.cloud.config.server.git.basedir=config
```

NOTE La uri puede ser hacia un repositorio local.

En el repositorio **git** deberan 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 configuracion 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 patrones

```
/{application}/{profile}[/{label}]
/{application}-{profile}.yml
/{label}/{application}-{profile}.yml
/{application}-{profile}.properties
/{label}/{application}-{profile}.properties
```

Donde

- **application**: será el identificador de la aplicacion **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**

7.5.1. Seguridad

Las funcionalidades del servidor de configuracion estan securizadas, para que cualquier usuario no pueda cambiar los datos de configuracion 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 configuracion se habra de configrar con Srping Security.

Se puede establecer con la propiedad

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

7.5.2. Clientes del Servidor de Configuracion

Una vez definido el **Servidor de Configuración**, los microservicios se conectarán a el 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 Configuracion**. 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 Configuracion**, 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 obtendra del servidor de configuración, de no obtenerla su valor será **Hello default**.

7.5.3. 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 repercusion inmediata, habrà que realizar configuraciones, en este caso la configuracion necesaria supone

añadir la anotacion **@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 configuracion

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

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

7.6. Servidor de Registro y Descubrimiento

Permite gestionar todas las instancias disponibles de los microservicios.

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

Los clientes del servidor de registro, buscarán en el 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 micoservicios.

Se incluyen varias implementaciones en Spring Cloud para serrvidor de registro/descubrimiento, Eureka Server, Zookeper, 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, unicamente es necesario lanzar la siguiente configuración.

```
@SpringBootApplication
@EnableEurekaServer
public class RegistrationServer {

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

7.6.1. Registrar Microservicio

Lo primero para poder registrar un microservicio en el servidor de descubrimineto 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
```

7.7. 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 proxeado, 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 ientificador del servicio en **Eureka** con el servicio real, se debe configurar donde encntrar 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);
  }
}
```

7.7.1. 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
```

7.8. Simplificación de Clientes de Microservicios con Feign

Feign abstrae el uso del API de RestTemplate para consultar los microservicios, encapsulandolo 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 aplicación **@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
```

7.8.1. 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");
    }
}
```

7.8.2. 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/
```

7.9. 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 siguientee 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/
```

7.9.1. Seguridad

En el caso de enrutar hacia servicios seguros, se puede configrar **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 desacivada 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
```

7.10. Circuit Breaker

La idea de este componente es la de evitar fallos en cascadaa, es decir que falle un componente, no por error propio del componente, sino porque falle otro componente de la aquitectura al que se invoca.

Para ello Spring Cloud integra **Hystrix**.

Para emplearlo, se ha de añadr 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 via de ejecución alterantiva **fallback**, para así evitar que se propague el error en la invocacion (try-catch).

Cuando se detectan una serie de errores en la respuesta, el proxy abre la conexión deribando al **falback**, aunque mantiene una validación periodica del estado de la respuesta, cerrando nuevamente la conexión cuando la respuesta vuelve a ser la correcta, este proceso puede ser monitorizado ya que se genera un stream con el estado de las conexiones.

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** debera retornar el mismo tipo de dato que el método proxeado, generalmente retornará una cache con el resultado de la última petición que se resolvio de forma correcta.

NOTE	No deberan 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);
  }
}
```

7.10.1. 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 traves 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>
```

7.10.2. 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>
```

7.11. Configuracion Distribuida en Bus de Mensajeria

Se trata de emplear un bus de mensajeria para trasladar el evento de refresco a todos los nodos del los microservicios que emplean una configuracion distribuida.

Se precisa por tanto de un bus de mensajeria, en este caso Spring Cloud apuesta por implementaciones **AMQP** frente a otras alternativas como podrian ser **JMS**. Y mas concretamente **RabbitMQ**.

Para instalar RabbitMQ, se necesita instalar Erlang a parte de RabbitMQ

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

7.11.1. Servidor

Se necesitará incluir las siguientes dependencias en el servidor de configuracion

```
<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 configuracion de la ubicacion 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 idealmente deberá acceder el repositorio de código donde se alberguen las configuraciones, para que cuando se produzca un commit nuevo, invocar el servicio, por ejemplo, GitHub proporciona **WebbHooks** para ello.

```
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": []} }'
```

7.11.2. 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 **RabitMQ**, al que habrá que configurar las siguientes propiedades

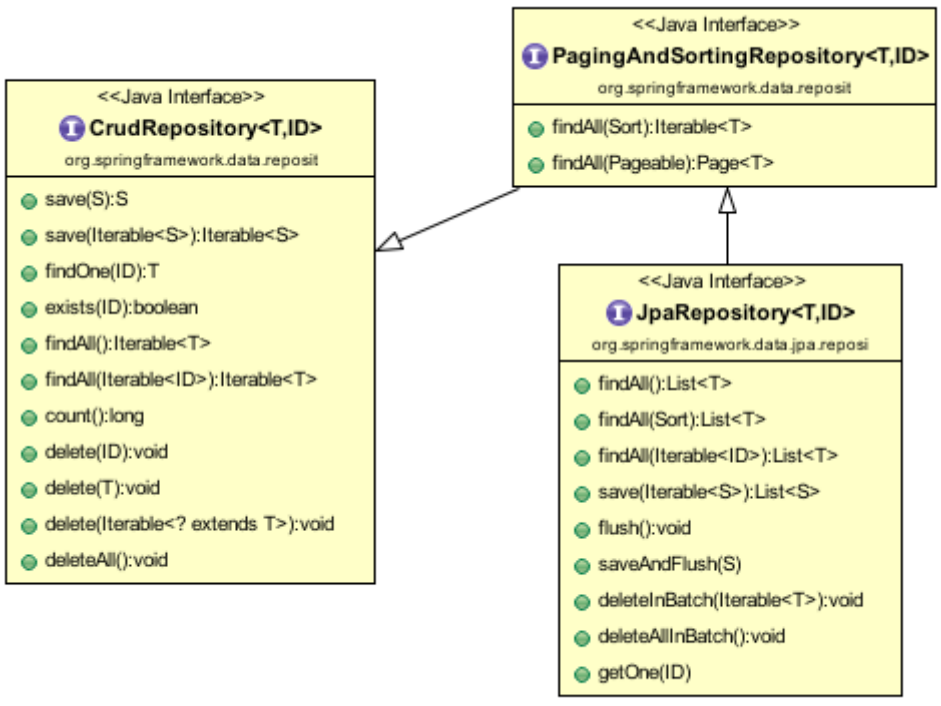
application.properties

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
```

8. Spring Data Jpa

Framework que extiende las funcionalidades de Spring ORM, permitiendo definir **Repositorios** de forma mas sencilla, sin repetir código, dado que ofrece numerosos métodos ya implementados y la posibilidad de crear nuevos tanto de consulta como de actualización de forma sencilla.

El framework, se basa en la definición de interfaces que extiendan la siguiente jerarquia, concretando el tipo entidad y el tipo de la clave primaria.



De forma paralela a las interfaces Jpa, existen para Mongo y Redis.

Por lo que la creación de un repositorio con Spring Data Jpa, se basará en la implementación de la interface **JpaRepository**

```
public interface CustomerRepository extends JpaRepository<Customer,
Long> {}
```

El convenio indica que el nombre de la interface, será **<entidad>Repository**.

La magia de Spring Data, es que no es necesario definir las implementaciones, estas se crean en tiempo de ejecución según se defina el Repositorio.

Para que esto suceda, se tendrá que añadir a la configuración con JavaConfig

```
@EnableJpaRepositories(basePackages="com.cursospring.persistencia")
```

o para configuraciones con XML

```
<jpa:repositories base-package="com.cursospring.persistencia" />
```

Además Spring Data Jpa, exige la definición de un bean de tipo **AbstractEntityManagerFactoryBean** que se llame **entityManagerFactory**.

```
<bean id="entityManagerFactory" class=
"org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="ds" />
    <property name="jpaVendorAdapter" ref="jpaVendorAdapter" />
    <property name="packagesToScan" value="com.curso.modelo.entidad"/>
    <property name="jpaProperties">
        <props>
            <prop key="hibernate.dialect">org.hibernate.dialect
.DerbyDialect</prop>
            <prop key="hibernate.show_sql">true</prop>
            <prop key="hibernate.format_sql">true</prop>
            <prop key="hibernate.hbm2ddl.auto">validate</prop>
            <prop key="hibernate.default_schema">CLIENTES</prop>
        </props>
    </property>
</bean>
```

Y otro de tipo **TransactionManager** que se llame **transactionManager**

```
<bean id="transactionManager" class=
"org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>
```

Ya que son dependencias de la implementación autogenerada que proporciona Spring Data.

8.1. Querys Personalizadas

Se pueden extender los métodos que ofrezca el Repositorio, siguiendo las siguientes reglas

- Prefijo del nombre del método **findBy** para busquedas y **countBy** para conteo de coincidencias.
- Seguido del nombre de los campos de busqueda concatenados por los operadores correspondientes: And, Or, Between, …¬ Todos los operadores aquí

```
List<Person> findByLastname(String lastname);

List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress,
String lastname);

List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname,
String firstname);
List<Person> findPeopleDistinctByLastnameOrFirstname(String lastname,
String firstname);

List<Person> findByLastnameIgnoreCase(String lastname);

List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname,
String firstname);

List<Person> findByLastnameOrderByFirstnameAsc(String lastname);
List<Person> findByLastnameOrderByFirstnameDesc(String lastname);
```

Tambien se pueden definir consultas personalizadas con JPQL

```
@Query("from Country c where lower(c.name) like lower(?1)")
List<Country> getByNameWithQuery(String name);

@Query("from Country c where lower(c.name) like lower(?1)")
Country findByNameWithQuery(@Param("name") String name);

@Query("select case when (count(c) > 0)  then true else false end from
Country c where c.name = ?1)")
boolean exists(String name);
```

8.2. Paginación y Ordenación

De forma analoga a la definición

A las consultas se puede añadir un último parametro de tipo **Pageable** o **Sort**, que permite definir el criterio de paginación o de ordenación.

```
Country findByNameWithQuery(Integer population, Sort sort);

@Query("from Country c where lower(c.name) like lower(?1)")
Page<Country> findByNameWithQuery(String name, Pageable page);
```

Definiendose el criterio a aplicar en tiempo de ejecución

```
countryRepository.findByNameWithQuery("%i%", new Sort( new Sort.Order
(Sort.Direction.ASC,"name")));

Page<Country> page = countryRepository.findByNameWithQuery("%i%", new
PageRequest(0, 3, new Sort( new Sort.Order(Sort.Direction.ASC,"name"))
));
```

8.3. Inserción / Actualización

Se pueden definir con JPQL, siempre que se cumpla

- El método debe estar anotado con @Modifying si no Spring Data JPA interpretará que se trata de una select y la ejecutará como tal.
- Se devolverá o void o un entero (int\Integer) que contendrá el número de objetos modificados o eliminados.

- El método deberá ser transaccional o bien ser invocado desde otro que sí lo sea.

```
@Transactional
@Modifying
@Query("UPDATE Country set creation = (?1)"
int updateCreation(Calendar creation);
```

```
@Transactional
int deleteByName(String name);
```

8.4. Procedimientos almacenados

Se han de declarar en la clase **@Entity** con las anotaciones

- **@NamedStoredProcedureQueries**
- **@NamedStoredProcedureQuery**
- **@StoredProcedureParameter**

```
@Entity
@Table(name = "MYTABLE")
@NamedStoredProcedureQueries({
    @NamedStoredProcedureQuery(
        name = "in_only_test",
        procedureName = "test_pkg.in_only_test",
        parameters = {
            @StoredProcedureParameter(
                mode = ParameterMode.IN,
                name = "inParam1",
                type = String.class)
        }
    ), @NamedStoredProcedureQuery(
        name = "in_and_out_test",
        procedureName = "test_pkg.in_and_out_test",
        parameters = {
            @StoredProcedureParameter(
                mode = ParameterMode.IN,
                name = "inParam1",
                type = String.class),
            @StoredProcedureParameter(
                mode = ParameterMode.OUT,
                name = "outParam1",
                type = String.class)
        })
})
public class MyTable implements Serializable { }
```

Y para emplearlas en los repositorios de **JPA Data**, se han de utilizar las siguientes anotaciones

- **@Procedure**
- **@Param**

```
public interface MyTableRepository extends CrudRepository<MyTable,
Long> {

    @Procedure(name = "in_only_test")
    void inOnlyTest(@Param("inParam1") String inParam1);

    @Procedure(name = "in_and_out_test")
    String inAndOutTest(@Param("inParam1") String inParam1);
}
```

8.5. Spring Boot

Si se emplea con Spring Boot, unicamente con añadir el starter **spring-boot-starter-data-jpa**

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

Y el driver de la base de datos, por ejemplo en este caso para una base de datos H2 embebida

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
</dependency>
```

Se tendrán configurados los objetos de JPA necesarios en el contexto de Spring y se buscarán las interfaces que hereden de **Repository** en los subpaquetes del paquete base de la aplicacion Spring Boot.

Simplemente con añadir el driver de las bases de datos embebidas (H2, Derby y HSQL), por defecto se define una cadena de conexion a una base de datos del tipo correspondiente en memoria, por ejemplo para H2, los datos de conexion son los siguientes.

- **DriverClass**: org.h2.Driver
- **JDBC URL**: jdbc:h2:mem:testdb

- **UserName**: sa
- **Password**: <blank>

Se puede emplear una pequeña aplicación web que se incluye con el driver de H2 para acceder a esta base de datos y realizar pequeñas tareas de administracion.

NOTE

Al ser una consola Web, se ha de añadir tambien el starter web

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

Una vez añadido, se ha de registrar el Servlet **org.h2.server.web.WebServlet** que se proporciona en dicho jar, que es el que da acceso a la aplicacion de gestion de H2.

Una alternativa para registrar el servlet es a través del **ServletRegistrationBean** *de Spring*

```
@Configuration
public class H2Configuration {
    @Bean
    ServletRegistrationBean h2servletRegistration() {
        ServletRegistrationBean registrationBean = new
ServletRegistrationBean(new WebServlet(), "/console/*");
        return registrationBean;
    }
}
```

Estando accesible la herramienta en la ruta **http://localhost:8080/console**.

8.6. Query DSL

API que permite unificar la creación de consultas en java para consumir distintos almacenes de datos, como: JPA, SQL, JDO, Lucene, MongoDB

Para poder emplear este API, se han de añadir las siguientes dependencias Maven

```
<dependency>
    <groupId>com.querydsl</groupId>
    <artifactId>querydsl-apt</artifactId>
    <version>4.1.4</version>
</dependency>

<dependency>
    <groupId>com.querydsl</groupId>
    <artifactId>querydsl-jpa</artifactId>
    <version>4.1.4</version>
</dependency>
```

En Spring Boot, ya se contempla esta libreria, por lo que unicamente habrá que añadir

```
<dependency>
    <groupId>com.querydsl</groupId>
    <artifactId>querydsl-apt</artifactId>
</dependency>

<dependency>
    <groupId>com.querydsl</groupId>
    <artifactId>querydsl-jpa</artifactId>
</dependency>
```

Y el siguiente plugin que permite generar el modelo de tipos empleado en la generación de consultas con **QueryDSL**, que son clases denominadas como las clases **Entity** con prefijo **Q**, y que son generadas a partir de las clases **@Entity**.

```
<plugin>
    <groupId>com.mysema.maven</groupId>
    <artifactId>apt-maven-plugin</artifactId>
    <version>1.1.3</version>
    <executions>
        <execution>
            <phase>generate-sources</phase>
            <goals>
                <goal>process</goal>
            </goals>
            <configuration>
                <outputDirectory>target/generated-
sources/java</outputDirectory>
                <processor>
com.querydsl.apt.jpa.JPAAnnotationProcessor</processor>
            </configuration>
        </execution>
    </executions>
</plugin>
```

Una vez definido el plugin y definidas las **Entity**, se ha de lanzar el comando

```
> mvn compile
```

Para que se generen las clases **Q**.

Este modelo de clases, proporciona un objeto estatico por cada clase que permite definir las consultas.

```
Customer customer = QCustomer.customer;
LocalDate today = new LocalDate();
BooleanExpression customerHasBirthday = customer.birthday.eq(today);
BooleanExpression isLongTermCustomer = customer.createdAt.lt(today
.minusYears(2));
```

Una vez generado el modelo de clases de **QueryDSL**, en proyecto **Data JPA**, se puede incluir la interface **QueryDslPredicateExecutor** a los repositorios.

```
public interface CustomerRepository extends JpaRepository<Customer,
Long>, QueryDslPredicateExecutor<Customer> {
}
```

Obteniendo un método **findAll** que permite ejecutar consultas de tipo **QueryDSL**.

```
BooleanExpression customerHasBirthday = customer.birthday.eq(today);
BooleanExpression isLongTermCustomer = customer.createdAt.lt(today
.minusYears(2));
customerRepository.findAll(customerHasBirthday.and(isLongTermCustomer))
;
```