Spring Boot

Victor Herrero Cazurro

Contenidos

1. ¿Que es Spring?

Framework para el de desarrollo de aplicaciones java.

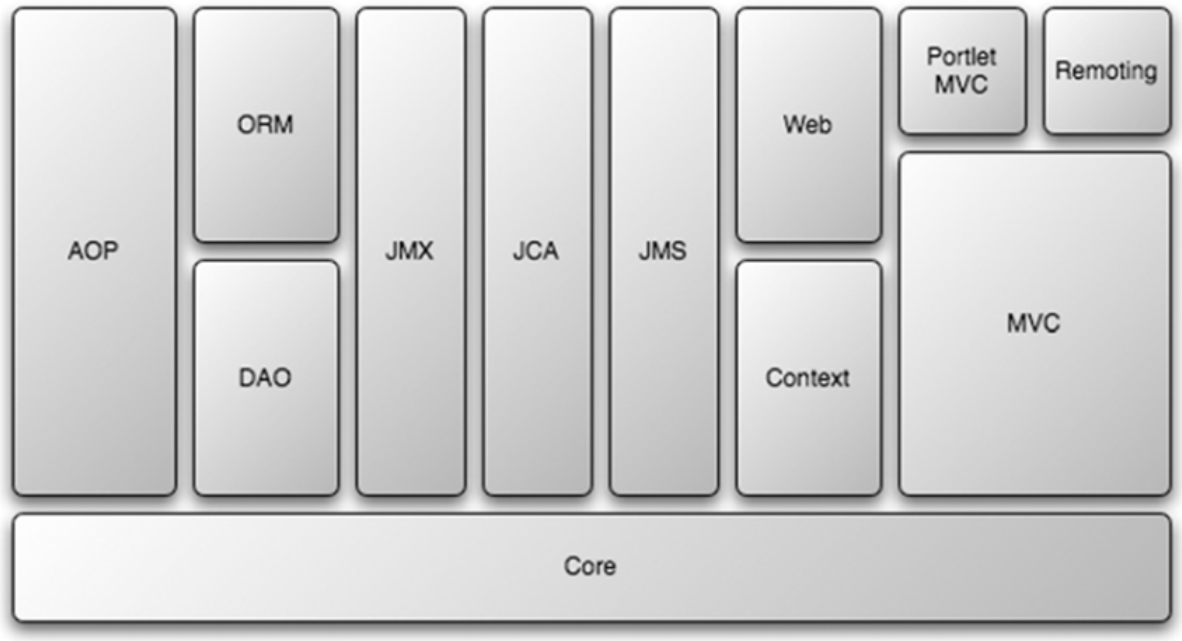
Es un contenedor ligero de POJOS que se encarga de la creación de beans (Factoría de Beans) mediante la Inversión de control y la inyección de dependencias.

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 crecerá **WEB-INF**, sera **src/main/resources/META-INF/resources/** o **src/main/webapp/**

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**

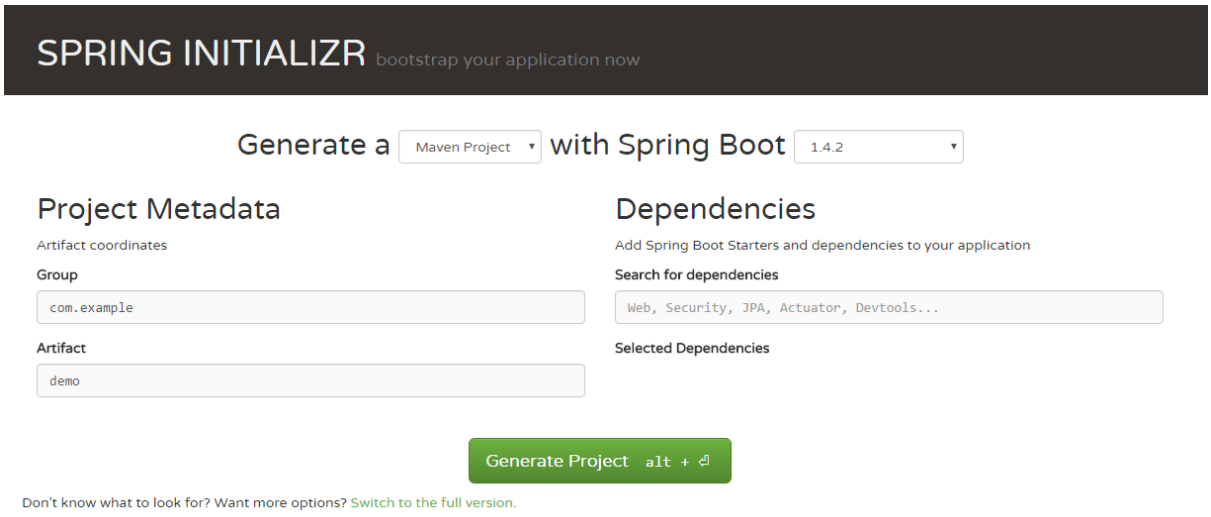
4.7. Uso de Java con start.spring.io

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

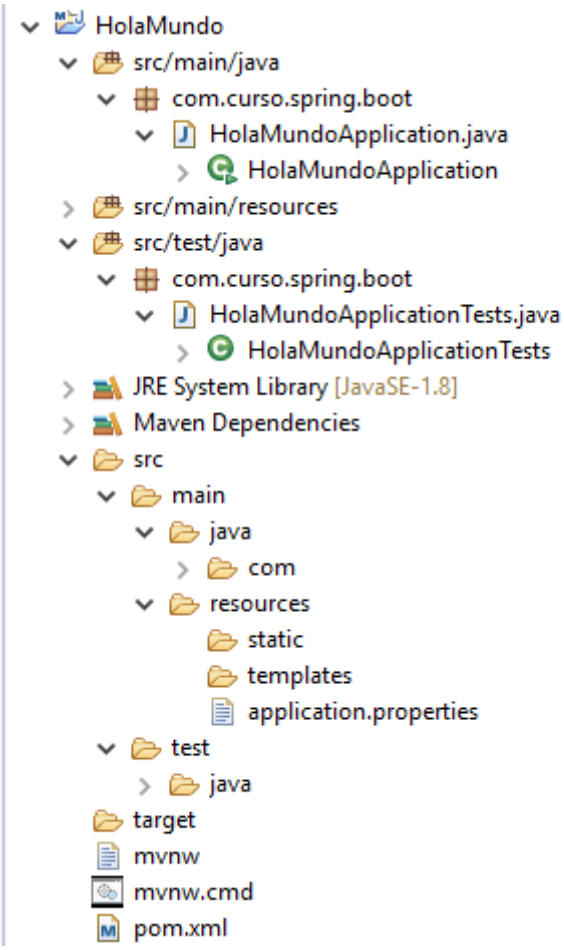
- Tipo de proyecto (Maven o Gradle)
- Versión de Spring Boot
- GroupId
- ArtifactId
- Dependencias

Existe una vista avanzada donde se pueden indicar otros parametros como

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

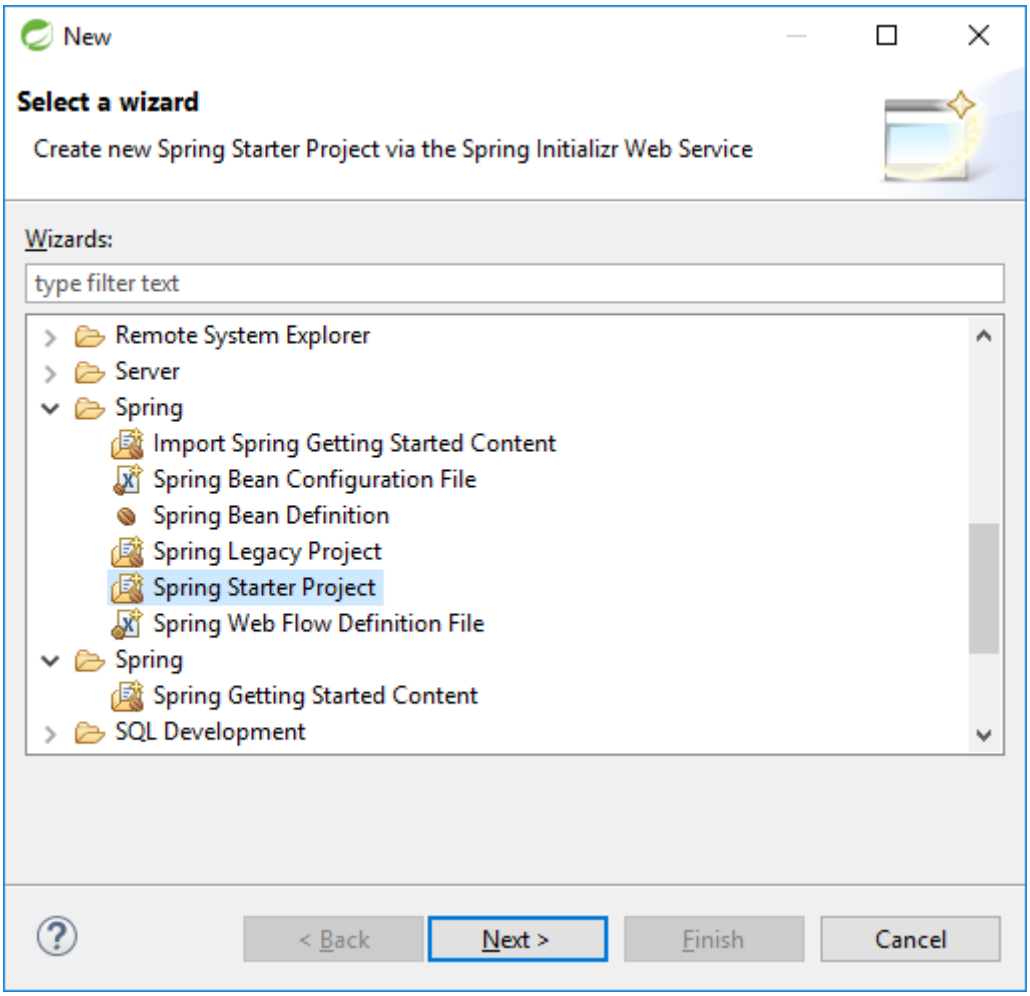
- Seleccion mas detallada de las dependencias

La estructura del proyecto con dependencia web generado será

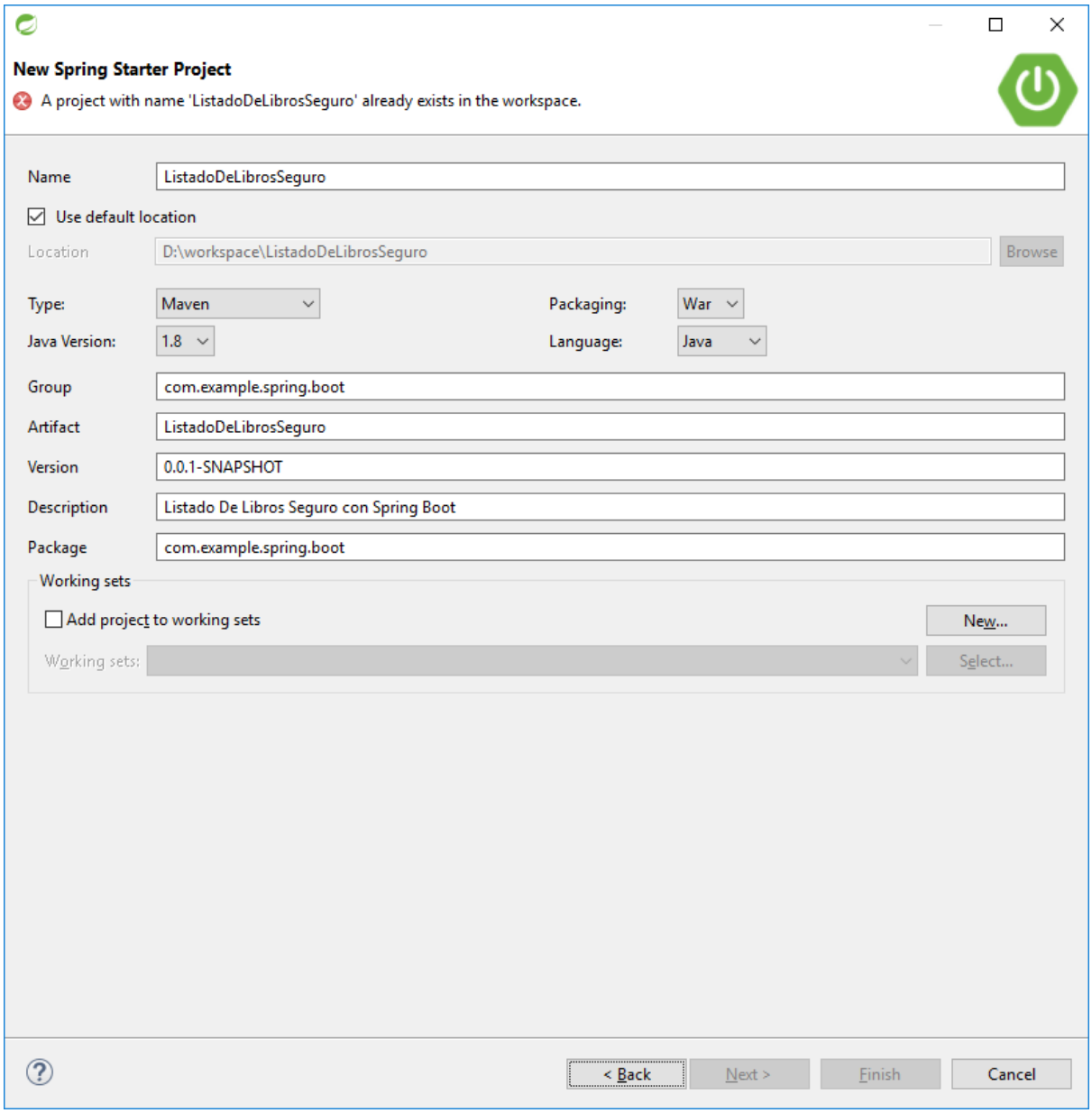


En esta estructura, cabe destacar el directorio **static**, destinado a contener cualquier recurso estatico de una aplicación web.

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 {
    @Autowired
    private JmsTemplate jmsTemplate;

    public void enviar(){
        jmsTemplate.convertAndSend("mailbox", new Email("Hello"));
    }
}
```

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 + ">");
    }
}
```

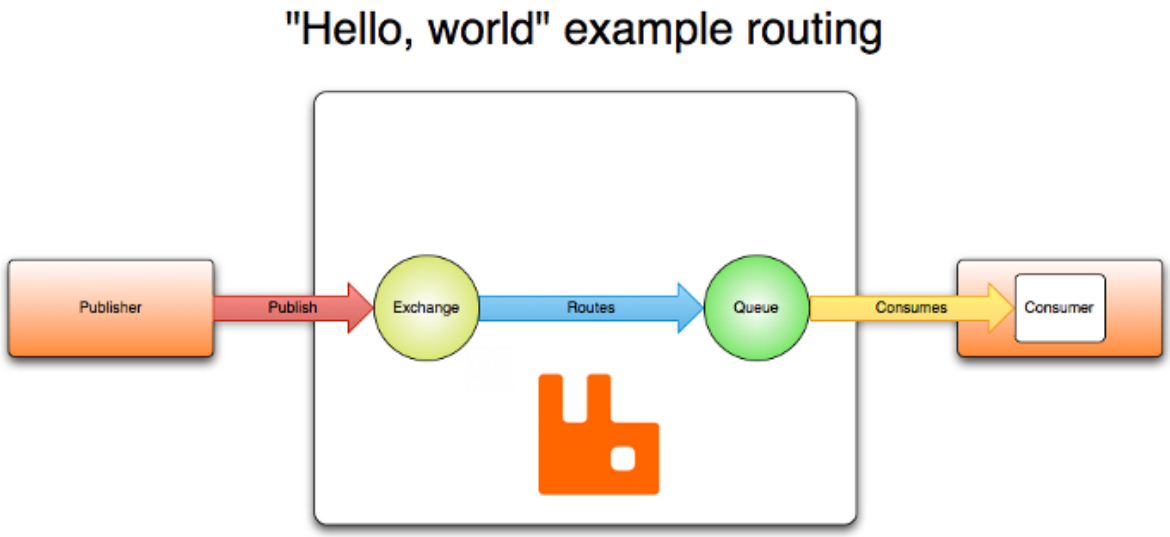
4.16. Soporte Mensajeria AMQP

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

4.16.1. 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, solamente 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 otros dos beans:

- **MessageListenerAdapter**: Envuelve el anterior bean indicando que es un listener y que el método **receiveMessage** será el que procesara los mensajes.

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

- **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("spring-boot");
    container.setMessageListener(listenerAdapter);
    return container;
}
```

Otra forma de hacerlo, es en vez de definir el Bean que procesa por un lado y el Adaptador por otro, se puede añadir la anotacion **@RabbitListener** al bean que procesa y juntar ambas configuraciones

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

El API de **RabbitMQ**, permite a la aplicacion, la creacion de los componentes del servicio AMQP, sin mas que definiendo los Beans correspondientes en el contexto de spring.

- **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.#");
}
```

4.16.2. 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!");
```

4.17. Testing

Para realizar pruebas en las aplicaciones Spring Boot, se ha de añadir el starter de test

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

Y crear clases de Test, anotadas con

```
@RunWith(SpringRunner.class)
public class SmokeTest {
}
```

4.17.1. Definicion del Contexto

La anotacion **@SpringBootTest**, crea el contexto de Spring empleando la clase aplicación de Spring Boot, aquella anotada con **@SpringBootApplication**, por lo que el contexto para las pruebas estará compuesto por los mismos beans que el de la aplicación.

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class SmokeTest {
}
```

Otra forma de definir el contexto es con la anotacion **@ContextConfiguration** donde en el atributo **classes** se le indican las clases que queremos que participen en el contexto de la prueba.

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

Y otra alternativa más es el uso de la anotacion **@TestConfiguration**.

```
@RunWith(SpringRunner.class)
public class EmployeeServiceImplTest {

    @TestConfiguration
    static class EmployeeServiceImplTestContextConfiguration {

        @Bean
        public EmployeeService employeeService() {
            return new EmployeeServiceImpl();
        }
    }

    @Autowired
    private EmployeeService employeeService;
}
```

Para obtener los Beans del contexto emplearemos la anotacion **@Autowired**

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class SmokeTest {
    @Autowired
    private HomeController controller;
}
```

NOTE

Una caracteritisca de estos Test, es que cachean el contexto de Spring a lo largo de la ejecucion de todos los metodos de testing, aunque esto se puede cambiar con **@DirtiesContext**, que creará un nuevo contexto para cada prueba.

NOTE

El Starter de Test, incluye la libreria **AssertJ**, que permite definir predicados para las aserciones, basandose en los métodos estaticos de la clase **org.assertj.core.api.Assertions**. Estos predicados son más legibles que los que se pueden definir con la libreria **JUnit** y los métodos estaticos de la clase **org.junit.Assert**

```
assertThat(this.restTemplate.getForObject("http://localho
st:8080/", String.class)).contains("Hello World");

assertThat(controller).isNotNull();
```

Se puede proporcionar una configuracion a emplear unicamente en los test con la anotacion **@TestPropertySource**

```
@RunWith(SpringRunner.class)
@TestPropertySource(locations = "classpath:application-
integrationtest.properties")
public class ExampleRepositoryTests {}
```

4.17.2. Mocks

Tambien se pueden definir Beans alternativos a los definidos en el contexto, creando **Mocks** con **Mockito**, para ello se proporciona la anotacion **@MockBean**.

```
@MockBean
private EmployeeRepository employeeRepository;
```

```
@RunWith(SpringRunner.class)
public class EmployeeServiceImplTest {

    //Este bean encesitará de un bean de tipo EmployeeRepository
    @Autowired
    private EmployeeService employeeService;

    @MockBean
    private EmployeeRepository employeeRepository;

    //Se definen los comportamientos del Mock antes de iniciar las
pruebas
    @Before
    public void setUp() {
        Employee alex = new Employee("alex");

        Mockito.when(employeeRepository.findByName(alex.getName()))
          .thenReturn(alex);
    }

    @Test
    public void whenValidName_thenEmployeeShouldBeFound() {
        String name = "alex";
        Employee found = employeeService.getEmployeeByName(name);

         assertThat(found.getName())
          .isEqualTo(name);
     }
}
```

4.17.3. Testing Web

Para testear la capa Web de una aplicacion Spring Boot, se ha de configurar la propiedad **webEnvironment** de la anotacion **@SpringBootTest**, pudiendo tener los siguientes valores

- **WebEnvironment.DEFINED_PORT**: Crea un entorno web, empleando las configuraciones del application.properties.
- **WebEnvironment.NONE**: No crea un entorno web.
- **WebEnvironment.MOCK**: Se define un entorno web mockeado, pero no arranca un servidor. Es el por defecto. Para probar los controladores se han de emplear

los objetos **MockMvc** o **WebTestClient**, y para inicializarlos las anotaciones **@AutoConfigureMockMvc** o **@AutoConfigureWebTestClient** respectivamente.

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureMockMvc
public class MockMvcExampleTests {

    @Autowired
    private MockMvc mvc;

    @Test
    public void exampleTest() throws Exception {
        this.mvc.perform(get("/")).andExpect(status().isOk())
                .andExpect(content().string("Hello World"));
    }
WebTestClient
}
```

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureWebTestClient
public class MockWebTestClientExampleTests {

    @Autowired
    private  webClient;

    @Test
    public void exampleTest() {
        this.webClient.get().uri("/").exchange().expectStatus().isOk()
                .expectBody(String.class).isEqualTo("Hello World");
    }

}
```

NOTE

Para la construccion de los predicados, se dispone de los métodos de las clases **org.springframework.test.web.servlet.request.MockMvcRequestBuilders**, **org.springframework.test.web.servlet.result.MockMvcResultHandlers** y **org.springframework.test.web.servlet.result.MockMvcResultMatchers**

```
this.mockMvc.perform(get("/"))
                .andDo(print())
                .andExpect(status().isOk())
                .andExpect(content().string
(containsString("Hello World")));
```

- **WebEnvironment.RANDOM_PORT**: Crea un entorno web con un puerto aleatorio, pudiendo obtener el puerto generado por inyeccion dentro de los Test, con la anotacion **@LocalServerPort**

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class HttpRequestTest {

    @LocalServerPort
    private int port;
}
```

Una vez configurada la propiedad **webEnvironment**, en el contexto de Spring se dispone de un Bean de tipo **TestRestTemplate**, que funciona de forma similar a **RestTemplate**.

```
@Autowired
private TestRestTemplate restTemplate;
```

NOTE

Tambien se pueden emplear los objetos **MockMvc** o **WebTestClient**.

NOTE

Se pueden definir Mocks con la anotacion **@MockBean**

4.17.4. Testing Json

Se ofrece en el API una anotacion **@JsonTest**, que facilita la serializacion y deserializacion a json

```
@RunWith(SpringRunner.class)
@SpringBootTest
@JsonTest
public class ApplicationTests {

    @Autowired
    private JacksonTester<Vehiculo> json;

    @Test
    public void testSerialize() throws Exception {
        Vehiculo details = new Vehiculo("Honda", "Civic");
        // Assert against a `.json` file in the same package as the
test
        assertThat(this.json.write(details)).isEqualToJson(
"expected.json");
        // Or use JSON path based assertions
        assertThat(this.json.write(details)).hasJsonPathStringValue(
"@.marca");
        assertThat(this.json.write(details))
.extractingJsonPathStringValue("@.marca").isEqualTo("Honda");
    }

    @Test
    public void testDeserialize() throws Exception {
        String content = "{\"marca\":\"Ford\",\"modelo\":\"Focus\"}";
        //Se ha de implementar el equals en vehiculo
        assertThat(this.json.parse(content)).isEqualTo(new Vehiculo(
"Ford", "Focus"));
        assertThat(this.json.parseObject(content).getMarca()).
isEqualTo("Ford");
    }

}
```

4.17.5. Testing Cliente Web

El API proporciona una anotacion que permite definir un mock de la respuesta que se obtendria del servidor empleando **RestTemplate**. Se habra de configurar el

Mock, que será un objeto de tipo **MockRestServiceServer**

```
@RunWith(SpringRunner.class)
@RestClientTest(RemoteVehicleDetailsService.class)
public class ExampleRestClientTest {

    @Autowired
    private RemoteVehicleDetailsService service;

    @Autowired
    private MockRestServiceServer server;

    @Test
    public void
getVehicleDetailsWhenResultIsSuccessShouldReturnDetails()
            throws Exception {
        this.server.expect(requestTo("/greet/details"))
                .andRespond(withSuccess("hello", MediaType.TEXT_PLAIN)
);
        String greeting = this.service.callRestService();
        assertThat(greeting).isEqualTo("hello");
    }

}
```

El objeto **RestTemplate**, se deberá obtener con **RestTemplateBuilder**

```
@Service
public class RemoteVehicleDetailsService {

    private final RestTemplate restTemplate;

    public RemoteVehicleDetailsService(RestTemplateBuilder
restTemplateBuilder) {
        restTemplate = restTemplateBuilder.build();
    }

    public String callRestService() {
        return restTemplate.getForObject("/greet/details",
          String.class, null);
    }
}
```

4.17.6. Testing BD

Para la configuracion del contexto de ejecución de los repositorios de **JPA Data**, se tiene la anotacion @**DataJpaTest**, que define

- Configura una base de datos H2 en memoria
- Setea Hibernate, Spring Data y el DataSource
- Realiza el escaneo de entidades

```
@RunWith(SpringRunner.class)
@DataJpaTest
public class ExampleRepositoryTests {

    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private UserRepository repository;

    @Test
    public void testExample() throws Exception {
        this.entityManager.persist(new User("sboot", "1234"));
        User user = this.repository.findByUsername("sboot");
        assertThat(user.getUsername()).isEqualTo("sboot");
        assertThat(user.getVin()).isEqualTo("1234");
    }

}
```

Si se desea se puede obtener el objeto **TestEntityManager** por inyeccion y emplearlo en los test.

```
    @Autowired
    private TestEntityManager entityManager;

    @Test
    public void test_cliente() {
        entityManager.persist(new Cliente("Bruno", "Garcia"));
    }
```

Si se desea precargar información en la base de datos, se puede definir en

src/test/resources un fichero **data.sql** con sentencias para poblar la base de datos.