Aplicaciones web con Spring Boot capa a capa

Por **Natalia Roales González** - 22 diciembre, 2016

En este tutorial vamos a aprender a desarrollar una aplicación web con los recursos que nos brinda Spring Web.

Índice de contenidos

1. Introducción

Muchos de vosotros habréis oído hablar de **Spring Boot**. Para los más despitadillos, os diré que se trata de un proyecto creado a partir de Spring, el cual nos permite **desarrollar y arrancar de forma muy rápida** aplicaciones basadas en Spring. Hoy os voy a demostrar que realmente esto es así, y para ello vamos a desarrollar una aplicación web muy sencillita, paso a paso. La aplicación consistirá en un pequeño servicio que nos muestra un mensaje de bienvenida al ser invocado.

Bueno, ¡pues comencemos a programar! 😀

2. Entorno

Este tutorial ha sido realizado en un entorno con las siguientes características:

- Hardware: MacBook Pro Retina 15' (2,5 GHz Intel Core i7, 16 GB DDR3)
- Sistema Operativo: OS X El Capitan 10.11.5
- Entorno de desarrollo: IntelliJ IDEA Ultimate 2016.1
- Java 1.8

3. Creando el proyecto

En primer lugar, creamos un **proyecto maven**. Para indicar que queremos utilizar Spring Web con Spring Boot, añadimos lo siguiente al fichero pom.xml:

XHTML

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.4.1.RELEASE</version>
```

```
</parent>

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

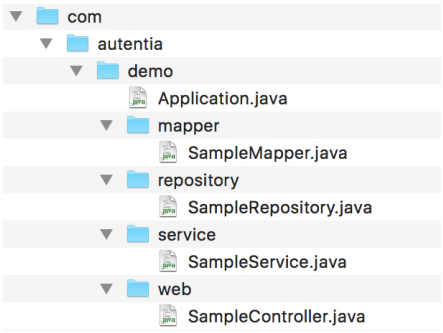
De este modo, el proyecto que acabamos de crear extiende del proyecto padre spring-boot-starter-parent, e incluye las dependencias agrupadas en el starter **spring-boot-starter-web**. Un starter es un **conjunto de dependencias** que nos ayudan a cubrir las necesidades de un tipo de proyecto concreto. Por ejemplo, el starter que estamos utilizando sirve para cubrir las necesidades de un proyecto web. Más adelante utilizaremos otros starters, entre ellos el que nos ayuda a integrar MyBatis en la aplicación.

Podemos definir en qué **versión de Java** desarrollaremos nuestra aplicación, añadiendo lo siguiente al fichero pom.xml:

XHTML

```
<properties>
    <java.version>1.8</java.version>
</properties>
```

En este caso, la versión que vamos a emplear será la 1.8. Otra cosa importante es la definición de una buena estructura de paquetes. Un buen ejemplo puede ser el siguiente:



Esta estructura de paquetes agrupa las clases en cuatro paquetes principales: mapper para la capa de acceso a datos, repository para la capa de repositorio, service para la capa de servicio, y web para la capa controlador.

No hay que seguir este ejemplo al pie de la letra ni mucho menos, es más, puede que la estructura de paquetes de otros proyectos sea muy distinta pero totalmente válida. Lo que pretendo mostraros es que debe existir una estructura de paquetes ordenada para que la aplicación sea mantenible y para que la responsabilidad de las clases quede bien clara.

4. Definiendo la clase principal

Toda aplicación en java debe contener una clase principal con un método main. Dicho método, en caso de implementar una aplicación con Spring, deberá llamar al método **run** de la clase **SpringApplication**. A continuación, definimos de una forma muy fácil la clase principal de nuestra aplicación.

Java

```
@Configuration
@EnableAutoConfiguration
```

```
@ComponentScan
public class Application {

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

La etiqueta **@Configuration**, indica que la clase en la que se encuentra contiene la configuración principal del proyecto.

La anotación **@EnableAutoConfiguration** indica que se aplicará la configuración automática del starter que hemos utilizado. Solo debe añadirse en un sitio, y es muy frecuente situarla en la clase main.

En tercer lugar, la etiqueta **@ComponentScan**, ayuda a localizar elementos etiquetados con otras anotaciones cuando sean necesarios.

Para no llenar nuestra clase de anotaciones, podemos sustituir las etiquetas @Configuration, @EnableAutoConfiguration y @ComponentScan por **@SpringBootApplication**, que engloba al resto.

Java

```
@SpringBootApplication
public class Application {

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

5. Empaquetando el proyecto y arrancando el servidor

Tras haber definido la clase principal del proyecto, podemos proceder a empaquetarlo y arrancar el servidor con nuestra aplicación. Si hemos añadido correctamente las dependencias al fichero pom.xml no debería haber ningún problema de empaquetado, y maven no nos devolvería ningún error en tiempo de compilación.

Para salir de dudas, vamos a ejecutar el siguiente comando en el directorio raíz del proyecto:

Shell

```
$ mvn clean package
```

Si todo va bien, tras ejecutar esta instrucción, se generarán los ficheros .class a partir de las clases .java y se empaquetará el proyecto en un fichero .jar. Podemos ver también qué dependencias han sido incluidas en el proyecto, es decir, qué dependencias engloban tanto los starters añadidos como el proyecto padre. Esto es posible con el siguiente comando:

Shell

```
$ mvn dependency:tree
```

Por otro lado, los proyectos de tipo Spring Boot integran un servidor de aplicaciones, por lo que arrancar una aplicación Spring Boot es muy fácil.

En el directorio raíz del proyecto ejecutamos el siguiente comando:

```Shell
$ mvn spring-boot:run
```

Si no se produce ningún error en tiempo de ejecución, el servidor estaría levantado y listo para recibir peticiones.

6. Desarrollando la capa controlador

Definamos ahora el comportamiento de la aplicación implementando el resto de clases. Vamos a comenzar por la capa de más alto nivel, la de los controladores, donde expondremos los servicios de la aplicación.

El servicio que vamos a crear tendrá un comportamiento muy simple. Recuperará de base de datos un mensaje de bienvenida cuyo contenido variará en función del idioma del usuario, recibiendo como parámetro el mismo nombre de usuario. El comportamiento del controlador será aún más sencillo, ya que lo único que hará será llamar a la capa de servicio y devolver lo que ésta nos retorne.

Comenzaremos por el desarrollo del test que valide el controlador. Nuestro controlador se llamará SampleController, así que el test del controlador se llamará SampleControllerTest y estará en el mismo paquete que SampleController pero en el directorio test. Necesitaremos incluir una serie de dependencias englobadas en el starter **spring-boot-starter-test**:

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

El contenido de la clase de test será el siguiente:

```Java
public class SampleControllerTest {

    private SampleController sampleController;

    private SampleService sampleService;

    @Before
    public void init(){
        sampleService = mock(SampleService.class);
        sampleController = new SampleController(sampleService);
    }

    @Test
    public void sampleControllerShouldCallService() {
        String userName = "nroales";
        String expectedMessage = "message";

        when(sampleService.welcome(userName)).thenReturn(expectedMessage);

        String message = sampleController.welcome(userName);

        verify(sampleService).welcome(userName);
        assertTrue(message.equals(expectedMessage));
    }
}
```

Hemos declarado un atributo de la clase SampleService, ya que lo que vamos a probar es que el controlador invoque a la capa de servicio y que devuelva lo mismo, así que debemos crear SampleService para que el test compile. Además, lo mockeamos para simular su comportamiento, pues el objetivo de este test no es probar la capa de servicio.

Vamos a hacer que el test pase de rojo a verde con la siguiente implementación de SampleController:

Java

```java
@Controller
public class SampleController {

    @Autowired
    private SampleService sampleService;

    public SampleController(SampleService sampleService) {
        this.sampleService = sampleService;
    }

    @RequestMapping(value = "/welcome/{userName}", method = RequestMethod.GET)
    @ResponseBody
    public String welcome(
            @PathVariable("userName") String userName
    )
    {
        return sampleService.welcome(userName);
    }
}
```

En el fragmento de código anterior aparecen algunas anotaciones. Vamos a ver qué significa cada una de ellas:

- **@Controller:** Con esta anotación Spring podrá detectar la clase SampleController cuando realice el escaneo de componentes.
- **@Autowired:** A través de esta anotación Spring será capaz de llevar a cabo la inyección de dependencias sobre el atributo marcado. En este caso, estamos inyectando la capa de servicio, y por eso no tenemos que instanciarla.
- **@RequestMapping:** Con esta anotación especificamos la ruta desde la que escuchará el servicio, y qué método le corresponde.
- **@ResponseBody:** Con ella definimos lo que será el cuerpo de la respuesta del servicio.
- **@PathVariable:** Sirve para indicar con qué variable de la url se relaciona el parámetro sobre el que se esté usando la anotación.

Podemos también utilizar la etiqueta **@RestController** en lugar de @Controller, que sustituye al uso de @Controller + @ResponseBody, quedando el controlador de la siguiente forma:

Java

```java
@RestController
public class SampleController {

    @Autowired
    private SampleService sampleService;

    public SampleController(SampleService sampleService) {
        this.sampleService = sampleService;
    }

    @RequestMapping(value = "/welcome/{userName}", method = RequestMethod.GET)
    public String welcome(
            @PathVariable("userName") String userName
    )
```

```
    {
        return sampleService.welcome(userName);
    }
}
```

7. Desarrollando la capa de servicio

Aunque a partir de este punto no aparezcan las clases de test, no quiere decir que no sean necesarias para completar el desarrollo que estamos realizando. Sin embargo, he decidido omitirlas para que el tutorial no se extienda demasiado, pero siempre es recomendable respaldar nuestra aplicación con una batería de pruebas (y más aún **hacer TDD**).

Vamos implementar la capa de servicio. Un método de servicio definirá una operación a nivel de negocio, por ejemplo, dar un mensaje de bienvenida. Los métodos de servicio estarán formados por otras operaciones más pequeñas, las cuales estarán definidas en la capa de repositorio. El mapper, por último, contendrá las operaciones de acceso a datos que serán invocadas por el repositorio.

En este caso, el servicio realizará una sola llamada al repositorio, pasándole como parámetro el nombre de usuario. Lo llamaremos SampleService.

Java

```java
@Service
public class SampleService {

    @Autowired
    private SampleRepository sampleRepository;

    public SampleService(SampleRepository sampleRepository) {
        this.sampleRepository = sampleRepository;
    }

    public String welcome(String userName) {
        return sampleRepository.getMessageByUser(userName);
    }

}
```

La anotación **@Service** funciona de forma parecida a la anotación **@Controller**, ya que permite que Spring reconozca a SampleService como servicio al escanear los componentes de la aplicación.

8. Desarrollando la capa de repositorio

Ahora tenemos que desarrollar el repositorio al que ha invocado el servidor. Por tanto, crearemos la clase SampleRepository e implementaremos el método getMessageByUser.

Java

```java
@Repository
public class SampleRepository {

    @Autowired
    private SampleMapper sampleMapper;

    public SampleRepository(SampleMapper sampleMapper) {
        this.sampleMapper = sampleMapper;
    }

    public String getMessageByUser(String userName) {
        String language = sampleMapper.getLanguageByUser(userName);

        return sampleMapper.getMessageByLanguage(language);
```

```
    }
}
```

Para recuperar el mensaje de bienvenida dado el nombre de usuario tendremos dos métodos en la capa de acceso a datos, siendo uno para recuperar el idioma dado el usuario, y otro para recuperar el mensaje dado el idioma. Desde el repositorio llamamos a los dos.

9. Desarrollando la capa de acceso a datos

Ya solo nos queda implementar la capa de acceso a datos. En esta capa es en donde se definen las consultas a base de datos, a través de interfaces denominadas mappers. Vamos a crear el mapper con los dos métodos invocados en el repositorio, que son getLanguageByUser y getMessageByLanguage.

Java

```java
@Mapper
public interface SampleMapper {

    String getLanguageByUser(@Param("userName") String userName);

    String getMessageByLanguage(@Param("language") String language);
}
```

Utilizamos la etiqueta **@Mapper** para indicar que una interfaz es un mapper, y así Spring pueda localizarla. También utilizamos la etiqueta **@Param** para que MyBatis identifique los campos a la hora de procesar las consultas.

Para poder trabajar con MyBatis debemos incluir algunas dependencias, agrupadas dentro del starter **mybatis-spring-boot-starter**:

XHTML

```xml
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>1.1.1</version>
</dependency>
```

También debemos añadir el conector correspondiente a la base de datos que vayamos a utilizar. Podemos hacerlo incluyendo su dependencia maven en el pom.xml. En caso de utilizar una base de datos MySQL añadimos la siguiente dependencia:

XHTML

```xml
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.6</version>
</dependency>
```

Y en el fichero **application.properties**, localizado en el directorio resources, añadiremos la información de nuestra base de datos, siendo testdb el nombre de la base:

Shell

```
spring.datasource.url=jdbc:mysql://localhost:3306/testdb
spring.datasource.username=user
spring.datasource.password=pass
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

Una vez tenemos todo lo necesario para trabajar con MyBatis, definimos cada uno de los dos métodos del mapper. Dentro de la carpeta resources, creamos una estructura de directorios idéntica a la estructura de paquetes donde se encuentra SampleMapper.java, y creamos el fichero SampleMapper.xml.

Tendrá el siguiente contenido:

```XHTML
<mapper namespace="com.autentia.demo.mapper.SampleMapper">

    <select id="getLanguageByUser" resultType="String">
        SELECT USL_LANGUAGE FROM USER_LANGUAGE WHERE USL_USER = #{userName}
    </select>

    <select id="getMessageByLanguage" resultType="String">
        SELECT MSG_DESCRIPTION FROM MESSAGES WHERE MSG_MESSAGE = "welcome" AND MSG_LANGUAGE = #
    </select>

</mapper>
```

La primera query recupera de la tabla USER_LANGUAGE el campo USL_LANGUAGE dado el valor del campo USL_USER, obteniéndose el idioma asociado al usuario userName. La segunda query, recupera el campo MSG_DESCRIPTION de la tabla MESSAGES dado el valor del campo MSG_LANGUAGE para MSG_MESSAGE igual a “welcome”, obteniéndose el mensaje de bienvenida en el idioma del usuario. Aunque parezca obvio, vuestras consultas deberán concordar con el diseño de la base de datos.

Lo que aparece dentro de #{userName} y #{language} son los identificadores que hemos designado a los parámetros de entrada con las anotaciones @Param.

10. Todo listo, ¡invoquemos al servicio!

Es el momento de la verdad… ¡Vamos a invocar a nuestro servicio! Ya sabéis, para ello **levantamos el servidor** con el siguiente comando:

```Shell
$ mvn spring-boot:run
```

Una vez haya arrancado el servidor procedemos a invocar al servicio, y para ello accedemos a la url http://localhost:8080/welcome/userName. Tenemos que sustituir userName por el nombre de usuario que hayamos guardado en nuestra base de datos. Como respuesta, veremos en el navegador el **mensaje de bienvenida** que hayamos definido.

11. Conclusiones

Gracias a Spring Boot nos acabamos de marcar en un momento una **aplicación totalmente funcional**.

Hemos comenzado creando un proyecto maven e indicando en el pom.xml que el proyecto es de tipo Spring Boot, heredando de spring-boot-starter-parent el proyecto creado. Después hemos elegido el starter que más se ajusta a las necesidades de nuestro tipo de proyecto, en este caso spring-boot-starter-web, y lo hemos añadido como dependencia. Luego hemos creado la clase principal de la aplicación, implementando posteriormente

las clases e interfaces que definen su comportamiento, y por último hemos arrancado el servidor.

El tiempo que hemos perdido en la **configuración del proyecto** es mínimo, y solo nos hemos tenido que preocupar de implementar los métodos que definan el comportamiento de los servicios. Tampoco hemos perdido tiempo en montar el servidor de aplicaciones, ya que Spring Boot cuenta con un **Tomcat embebido**.

Ya no tenéis excusa para no desarrollar proyectos web en Java. ¿Habéis visto qué **fácil** y **rápido** es tener una aplicación Java web funcional **desde cero**? Os animo a que lo probéis 😉

¡Hasta el próximo tutorial!

12. Referencias

- Guía de referencia de Spring Boot
- Listado de starters no oficiales de Spring Boot

Natalia Roales González

Consultora tecnológica de desarrollo de proyectos informáticos.

Graduada en Ingeniería Informática en la Universidad Autónoma de Madrid.

Puedes encontrarme en Autentia: Ofrecemos servicios de soporte a desarrollo, factoría y formación.

Somos expertos en Java/Java EE