Java Spring Boot

Unidad 3. Microservicios con Spring Boot

Índice

MADRID – BARCELONA – BILBAO – VALENCIA - SEVILLA info@formadoresit.es www.formadoresit.es

Introducción

Los microservicios constituyen uno de los elementos software de los que más se está escuchando hablar en los últimos años. Gracias a ellos, la programación distribuida puede llevarse a cabo de forma real, eficiente y sencilla.

Y para hacerse este nuevo paradigma de programación posible, tenemos a nuestra disposición el framework Spring Boot, nacido precisamente para crear aplicaciones autónomas y autodesplegables, una característica esencial de los microservicios. Nunca fue tan fácil crear y desplegar un servicio Rest como lo es ahora gracias a Spring Boot.

Objetivos

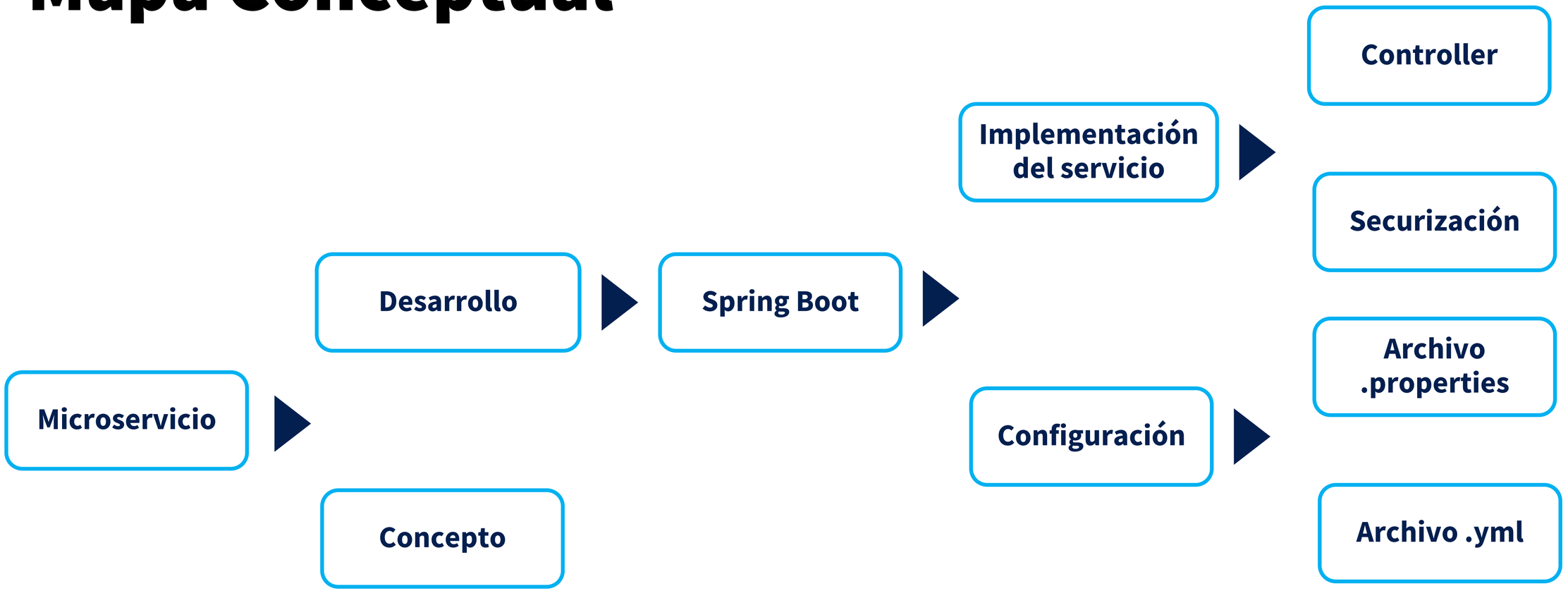
Identificar las características de un microservicio.

Comprender las ventajas del uso de Spring Boot en la creación de microservicios.

Aplicar las anotaciones y elementos de configuración para la creación de microservicios Spring Boot.

Securizar el acceso a microservicios en Spring boot.

Mapa Conceptual



Microservicios

Concepto

Durante el capítulo anterior, hemos estado estudiando como crear servicios REST utilizando el framework Spring. Actualmente, se está hablando bastante del termino microservicio, más que de servicio Web, pero, ¿Qué es realmente un microservicio?

Un microservicio no es más que un servicio Web, pero que posee algunas características peculiares que lo diferencian de un servicio clásico:

• **Realiza una tarea muy concreta**. Los microservicios son piezas de software que se encargan de realizar una tarea muy concreta y específica

• **Autocontenidos**. Quizá sea la característica más significativa de un microservicio, y es que un incluye todo lo necesario para realizar su despliegue en una máquina, sin necesidad de que exista ningún entorno de ejecución adicional. Los servicios clásicos se despliegan sobre un servidor de aplicaciones, en cambio, **los microservicios no necesitan de la existencia de dicho software**, ya que incorporan todo lo necesario para su ejecución.

• **Autodesplegables**. Al incorporar todo lo necesario para su ejecución, el microservicio **no requiere de un proceso de despliegue** en la máquina donde va a se alojado, basta con arrancarlo directamente.

Microservicios

Creación

Desde el punto de vista de la implementación del microservicio, no existe ninguna diferencia con los servicios REST tradicionales, es decir, en nuestro caso concreto utilizaremos las anotaciones y librerías proporcionadas por Spring para su desarrollo.

Sin embargo, el hecho de que el servicio incorpore el entorno de ejecución, nos lleva a utilizar un elemento del framework Spring que está adquiriendo una gran popularidad en los últimos años, que es **Spring Boot**.

Spring Boot

Fundamentos

Spring boot es una parte del framework Spring, cuyo principal objetivo es simplificar el desarrollo de aplicaciones con Spring. Más en concreto, Spring boot nos proporciona tres principales beneficios a la hora de construir aplicaciones:

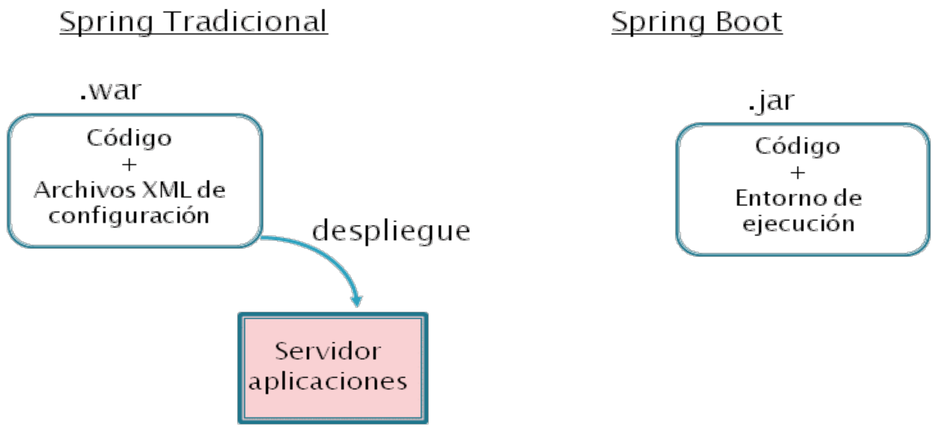
- **Autoconfigurable**. Cuando creamos una aplicación con Spring boot, se asumen una serie de configuraciones por defecto, de modo que el desarrollador **no tiene que tratar con los complejos archivos XML** del Spring tradicional. Esto simplifica enormemente las tareas de configuración, algo que siempre ha resultado ser una tarea pesada en Spring.
- **Autoejecutable**. Las aplicaciones **Spring boot se inician solas**, no necesitan de ningún otro componente externo que se encargue de iniciar el framework. De hecho, una aplicación Spring Boot es una aplicación Java de consola.
- **Autodesplegable**. En el caso de que vayamos a construir una aplicación Web con Spring Boot se **incluirá automáticamente el entorno de ejecución de la misma**, es decir, no necesitamos de un servidor de aplicaciones para realizar el despliegue de la aplicación. Esto hace que Spring Boot sea **especialmente interesante para la creación de microservicios**.
- **Simplificación de dependencias**. La inclusión de dependencias vía Maven en un proyecto Spring Boot es una terea sumamente simple gracias a los starters, que llevan asociados implícitamente un conjunto de dependencias maven para desarrollar un tipo de aplicación.

Spring Boot

Microservicios

Como hemos visto, todas estas características que tiene Spring Boot lo hacen especialmente interesante para la creación de microservicios, donde la propia aplicación debe incluir todo lo necesario para su ejecución.

En la siguiente imagen vemos la diferencia entre lo que sería un servicio tradicional y un servicio creado con Spring Boot.



Un microservicio creado con Spring Boot es una aplicación Java estándar que no necesita ser desplegada en ningún servidor. Basta con ejecutar la clase main de la aplicación para que el servicio se autodespliegue y esté listo para ser utilizado.

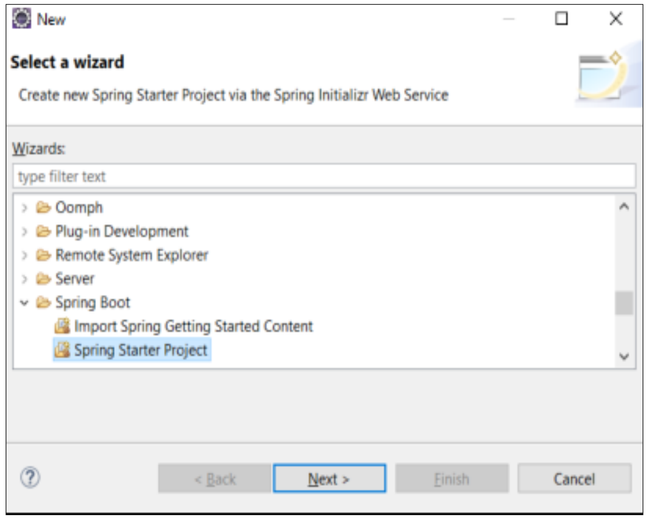
Creación de un servicio con Spring Boot

Creación del proyecto

Si tenemos instalado en eclipse el plugin Spring Tool Suite 4, dispondremos de una plantilla de proyecto especial para la creación de aplicaciones Spring Boot.

A continuación explicaremos el proceso de creación de un microservicio sencillo que devuelva un mensaje de saludo ante una petición GET. Este microservicio lo crearemos como una aplicación Spring Boot.

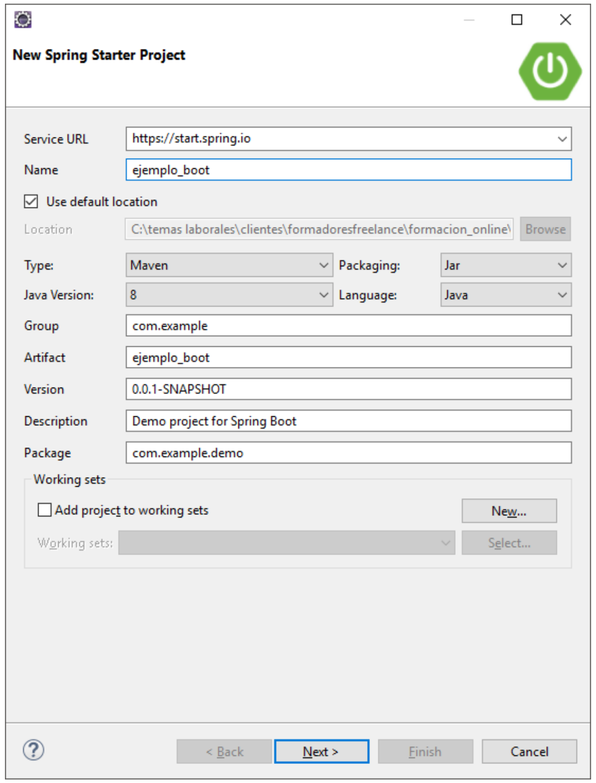
Lo primero será crear el proyecto, para ello nos vamos a la opción de menú File->New->Other y en el cuadro de diálogo que aparece a continuación buscaremos la categoría Spring Boot y dentro de la categoría Spring Boot, elegiremos "Spring Stared Project“:



Creación de un servicio con Spring Boot

Creación del proyecto

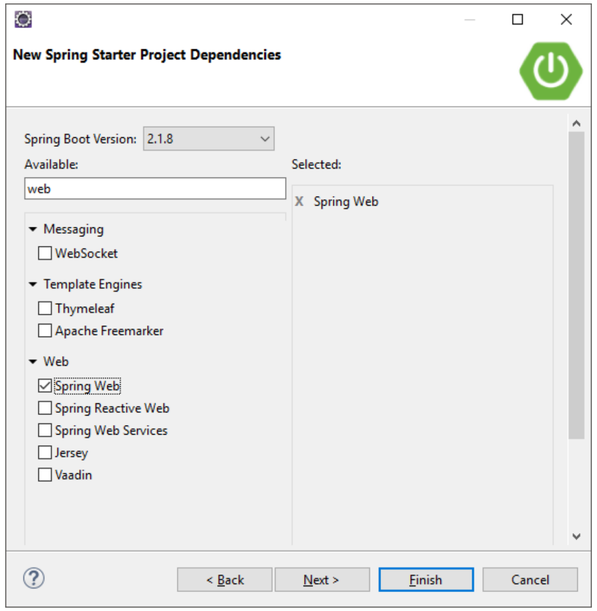
En el siguiente cuadro de diálogo, le daremos un nombre a nuestro proyecto (por ejemplo “ejemplo_boot") y también al paquete principal. El resto de opciones las dejaremos con los valores por defecto:



Creación de un servicio con Spring Boot

Creación del proyecto

Lo que viene a continuación es muy interesante, ya que el asistente incorpora **una opción para poder añadir de forma sencilla las dependencias de nuestro proyecto**. Por ejemplo, dado que se trata de una aplicación Web, introducimos la palabra “Web” en el buscador y elegiremos la opción “Spring Web” en los resultados de búsqueda. Con ello, ya se incluyen las dependencias necesarias para construir una aplicación Web con Spring.



A continuación, pulsaremos el botón "Finish" y se habrá completado la creación del proyecto con el asistente.

Creación de un servicio con Spring Boot

Starters Maven

Si nos fijamos en el archivo pom.xml, vemos que son muy pocas las dependencias que necesitamos en comparación con una aplicación Spring tradicional. En lugar de las referencias a las librerías clásicas de Spring, en **Spring Boot se utilizan los straters.**

```
<dependencies>

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

	</dependencies>
```

Cada starter lleva asociados una serie de dependencias a librerías Spring, lo cual hace que no tengamos que preocuparnos de incluir todas y cada una de estas librerías y simplifica enormemente la creación del pom.xml.

Creación de un servicio con Spring Boot

La clase main

Una vez creado el proyecto, dentro de `src/main/java`, encontramos la clase principal llamada EjemploBootApplication, dentro del paquete indicado en el asistente. El aspecto de la clase se muestra en el siguiente listado:

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

Podemos observar como la clase se encuentra anotada con **@SpringBootConfiguration**, que es lo único que necesitamos para configurar nuestra aplicación Spring Boot. Esta anotación por si sola proporciona la funcionalidad de otras tres anotaciones:

- **@Configuration**. Indica que la clase puede incluir configuración de objetos Spring.
- **@EnableAutoConfiguration**. Asume una serie de configuraciones por defecto en el contexto de aplicación de Spring.
- **@ComponentScan**. Permite que Spring pueda crear instancias de clases localizadas en ciertos paquetes. De forma predeterminada, **se localizan y se instancian todas las clases localizadas dentro del paquete principal**. Si queremos que spring instancie clases de otros paquetes, habría que incluir explícitamente esta anotación e indicar en el atributo *basePackages* los paquetes donde se encuentran estas clases.

Ahora, solamente tendremos que preocuparnos por la implementación de nuestro servicio.

Creación de un servicio con Spring Boot

Implementación del servicio

De cara a lo que es la implementación de un servicio o microservicio REST con Spring Boot, no tenemos que hacer **nada diferente a lo que hemos hecho anteriormente con los servicios REST** creados en una aplicación Web. Es decir, implementaremos la lógica de negocio de la aplicación y el controlador Rest con las anotaciones para exponer los recursos correspondientes.

En este ejemplo solo queremos exponer un recurso que genere un mensaje de saludo ante una petición GET, por tanto, implementaremos **una clase controladora en el mismo paquete donde se encuentra la clase main**. El código de esta clase será el que se indica en este listado:

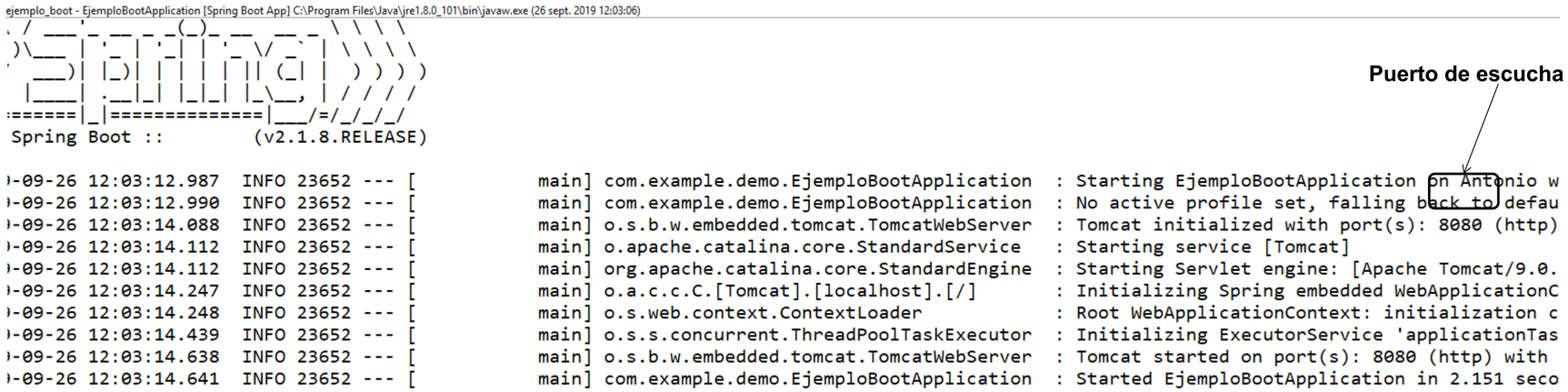
```
@RestController
public class SaludoBoot {
   @GetMapping(value="saludo",produces=MediaType.TEXT_PLAIN_VALUE)
   public String saludar() {
      return "Bienvenido a mi servicio Spring Boot";
   }
}
```

Creación de un servicio con Spring Boot

Ejecución y testing

Una vez completado el servicio, lo autodesplegaremos ejecutando la clase principal del proyecto. Con el ratón nos situamos sobre la clase main y en el menú que aparece al pulsar el botón derecho del ratón elegimos *Run As -> Spring Boot App*.

En la consola podemos ver como se ejecuta la aplicación y se nos informa que el Tomcat está iniciado en el puerto 8080. Y es que **la aplicación Spring Boot incluye una versión ligera de Tomcat** en la que despliega el servicio al ejecutar la aplicación:



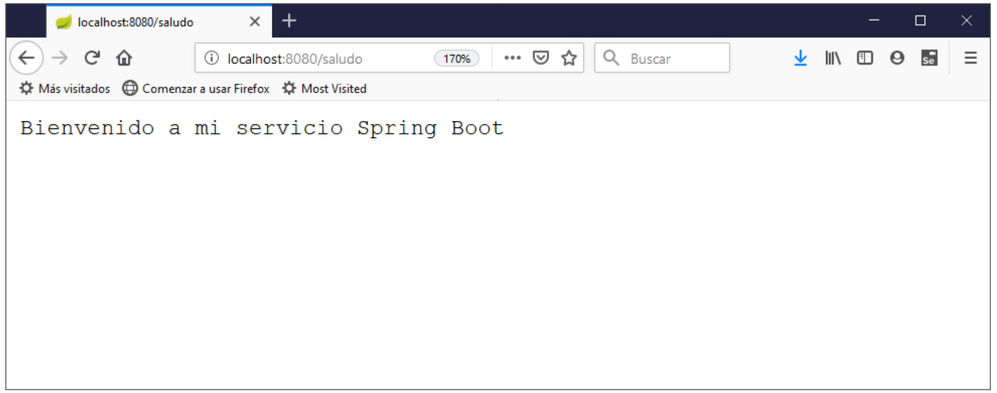
Creación de un servicio con Spring Boot

Ejecución y testing

Y ya está todo, ahora probaremos el servicio lanzando una petición GET del recurso correspondiente. Si escribimos la siguiente dirección en un navegador:

`http://localhost:8080/saludo`

Veremos como resultado:



Por supuesto, podemos utilizar Postman para testear también microservicios.

Como vemos, el proceso de creación de un microservicio con Spring Boot es tremendamente simple, ya que permite al programador **centrarse en lo que es la implementación del servicio** sin tener que preocuparse por detalles de configuración y despliegue del mismo.

Archivo de propiedades

application.properties

Spring Boot asume una serie de configuraciones por defecto y **elimina el uso de pesados archivos XML** para la configuración de aplicaciones. Sin embargo, en algunos casos es necesario indicarle a la aplicación ciertas propiedades que debe tener en cuenta de cara a configurar el servicio de manera diferente a como lo hace por defecto.

Para ello, utilizaremos **el archivo application.properties**, basado en la utilización de **parejas *clave=valor*** para el establecimiento de las propiedades de la aplicación

Por ejemplo, si queremos que el servidor tomcat embebido en el servicio se inicie en otro puerto diferente al por defecto, por ejemplo, en el 9000, deberíamos incluir la siguiente línea en application.properties, que podemos encontrarlo en la carpeta de proyecto src/main/resources:

```
server.port=9000
```

En este caso, si paramos la aplicación y volvemos a ejecutarla, para acceder al servicio sería mediante la url:

```
http://localhost:9000/saludo
```

Otra propiedad es server.servlet.context-path, mediante la cual **podemos establecer un context-root** o dirección base a la aplicación Web. Por ejemplo, si indicamos:

```
server.servlet.context-path=/service
```

Para acceder al servicio de ejemplo tendríamos que utilizar la url:

```
http://localhost:9000/service/saludo
```

Archivo de propiedades

application.properties

Existen una gran cantidad de propiedades que podemos configurar sobre distintos aspectos de la aplicación y de su entorno de ejecución.

En la siguiente dirección de la página oficial de spring.io, podemos encontrar muchas de esas propiedades, clasificadas por categorías, en función de los aspectos tratados por las mismas:

https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html

Archivo de propiedades

application.yml

Otra alternativa a la hora de definir propiedades en un microservicio Spring Boot es **utilizar archivos .yml**, que permiten estructurar la información al estilo XML pero de forma más clara y directa.

En caso de optar por este tipo de configuración, el archivo tendría que llamarse application.yml. A continuación vemos como quedaría un application.yml en el que definimos el puerto del servidor:

```
server:
    port:9000
    servlet:
        context-path:/service
```

Ejercicio Práctico

Implementar el servicio de cursos desarrollado en el capítulo anterior como un microservicio. En este caso, no deberán existir archivos de configuración .xml y toda la información deberá suministrarse a través de archivos de propiedades y objetos de configuración.

Solución ejercicio

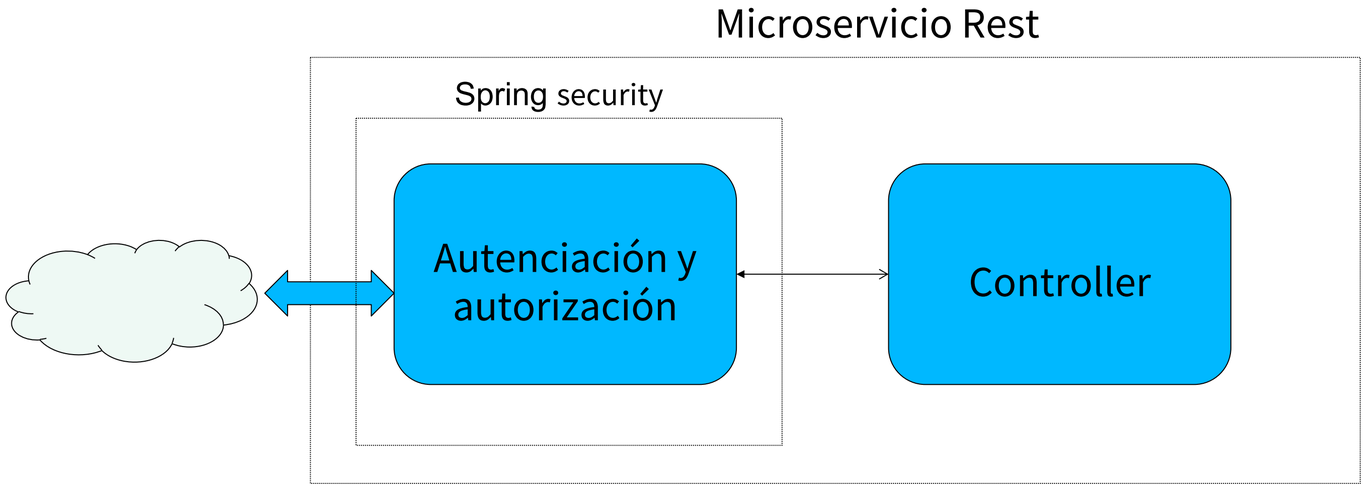
- En el siguiente video tutorial podrás ver la solución al ejercicio propuesto, con la explicación detallada de los componentes desarrollados.

VER TUTORIAL

Securización de microservicios

Fundamentos

En muchas ocasiones, el acceso a los recursos de un microservicio o servicio Rest debe ser securizado, a fin de que solamente pueden hacer uso de dicho recurso los usuarios que previamente hayan sido autenticados y estén autorizados a poder utilizar dicho recurso. Esto requiere aplicar un **mecanismo de autenticación y autorización** en el acceso al microservicio.



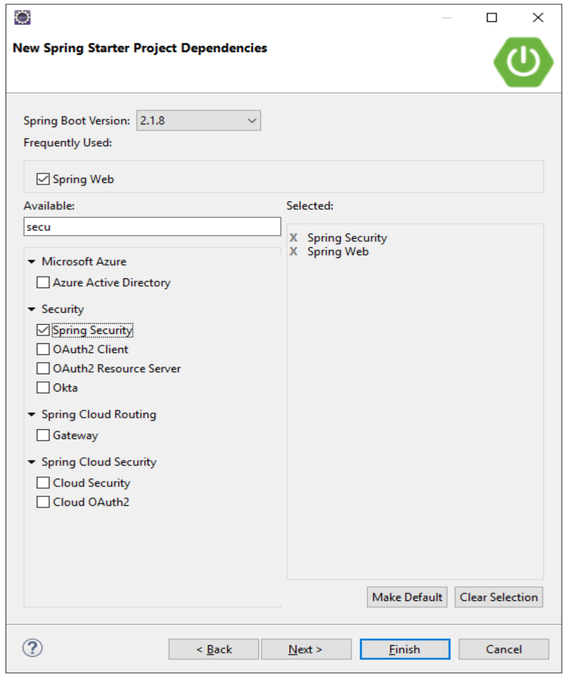
Este proceso de autenticación y autorización de usuarios es llevado a cabo directamente por el módulo Spring security que deberá ser incorporado en el proyecto. Como desarrolladores del servicio, debemos preocuparnos por indicarle a Spring el **mecanismo de autenticación utilizado y las políticas de autorización en el acceso a los recursos** que se deben seguir. Este proceso se realizará en una clase de configuración independiente.

Securización de microservicios

Configuración del proyecto

Vamos a crear un nuevo proyecto en el que vamos a exponer dos recursos, uno deberá estar securizado y el otro no. El proyecto lo vamos a llamar servicio_empleados, y lo crearemos como tipo Spring Boot (Spring Started Project).

A la hora de añadir las dependencias, incluiremos, además de Spring Web, la dependencia Spring security:



Securización de microservicios

Configuración del proyecto

Al finalizar el asistente, veremos como se añadido un nuevo starter en el pom.xml para la inclusión de las dependencias de seguridad:

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

Securización de microservicios

Código de la aplicación

En este nuevo microservicio, expondremos **dos recursos vía GET**, uno de ellos devolverá la lista de empleados existentes y el otro el empleado cuyo identificador se reciba como variable en la URL. Al primero podrán acceder todos los usuarios que estén autenticados, mientras que al segundo solo podrán acceder los usuarios pertenecientes al rol “ADMIN”.

Lo primer, será crear el código del servicio, dentro del cual no se tendrá que escribir ninguna instrucción ni anotación para la securización del mismo.

En primer lugar, vemos la clase Javabean que será incluida en el paquete *model*:

```
package model;
public class Empleado {
    private int idEmpleado;
    private String nombre;
    private double salario;
    public Empleado() {
        super();
    }
```

```
    public Empleado(int idEmpleado, String nombre, double salario) {
        super();
        this.idEmpleado = idEmpleado;
        this.nombre = nombre;
        this.salario = salario;
    }
      public int getIdEmpleado() {
        return idEmpleado;
    }
    public void setIdEmpleado(int idEmpleado) {
        this.idEmpleado = idEmpleado;
    }
       …
}
```

Securización de microservicios

Código de la aplicación

En el siguiente listado tenemos la clase controladora del servicio que definiremos en el paquete *controllers*:

```
@RestController
public class EmpleadosController {
   private List<Empleado> empleados;
   @PostConstruct
   public void init() {
      empleados=new ArrayList<>();
      empleados.add(new Empleado(2003,"Javier López",1780));
      empleados.add(new Empleado(1000,"María Sánchez",1900));
      empleados.add(new Empleado(4000,"David Martín",1600));
   }
   @GetMapping(value="empleados",produces=MediaType.APPLICATION_JSON_VALUE)
   public List<Empleado> recuperarEmpleados(){
      return empleados;
   }
   @GetMapping(value="empleado/{id}",produces=MediaType.APPLICATION_JSON_VALUE)
   public Empleado buscarContacto(@PathVariable("id") int id ){
      for(Empleado emp:empleados) {
         if(emp.getIdEmpleado()==id) {
            return emp;
         }
      }
      return null;
   }
}
```

Securización de microservicios

Código de la aplicación

Por otro lado, la clase main será prácticamente la misma que en el ejemplo anterior, solo que debemos indicar a través de la anotación **@ComponentScan** los **paquetes en donde se encuentran las clases que Spring debe instanciar**. En nuestro caso el paquete controllers y config, en este último será donde definamos la clase de configuración de seguridad:

```
@ComponentScan(basePackages= {"controllers","config"})
@SpringBootApplication
public class Application {
   public static void main(String[] args) {
      SpringApplication.run(Application.class, args);
   }
}
```

Securización de microservicios

Configuración de seguridad

La configuración de seguridad la realizaremos en una clase que deberá heredar WebSecurityConfigurerAdapter, y en la que sobrescribiremos dos versiones del método configure heredado de ésta, en uno de ellos realizaremos la **definición de usuarios y roles para el proceso de autenticación** y en el otro **las políticas de seguridad de acceso a los recursos para el proceso de autorización.**

El esquelto de la clase se muestra en el siguiente listado:

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter{
   @Bean
   public AuthenticationManager authMg() throws Exception {
      return super.authenticationManagerBean();
   }

   //definición roles y usuarios
   @Override
   protected void configure(AuthenticationManagerBuilder auth) throws Exception {

     }
   //políticas de acceso a recursos y mecanismo de autenticación
   @Override
   protected void configure(HttpSecurity http) throws Exception {

   }
}
```

Securización de microservicios

Configuración de seguridad

Lo primero que vemos en la clase anterior, es que la clase está anotada con @Configuration, lo que le permite a Spring **considerar como beans de Spring** todos aquellos objetos devueltos por métodos anotados con @Bean.

Uno de esos objetos es el devuelto por el método *authMg()* , y que será **inyectado como parámetro** en el primero método *configure()* definido en la clase.

A continuación, veremos la implementación de los dos métodos configure().

Securización de microservicios

Configuración de seguridad

La implementación del primero de los métodos *configure()* será como se indica en el siguiente listado:

```
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
     auth
         .inMemoryAuthentication()
         .withUser("user1")
           .password("{noop}user1") //lo de {noop} se pone para no obligar a usar mecanismo de
encriptación
           .roles("USER")
           .and()
         .withUser("admin")
           .password("{noop}admin")
           .roles("USER", "ADMIN");
}
```

A través del método *inMemoryAuthentication()* de AuthenticationManagerBuilder indicamos que la deinición de usuarios se realiza en memoria (hay otros métodos, como base de datos o LDAP). Sobre el objeto UserDetailsBuilder devuelto, se van realizando llamadas *a withUser(), password() y roles()* para indicar los **nombres de usuario a crear, su contraseña y el rol al que pertenecen**, respectivamente. En este ejemplo creamos dos usuarios (“user1” y “user2”), cada funo formando parte de un rol.

Securización de microservicios

Configuración de seguridad

En cuanto al segundo de los métodos, quedaría como se indica a continuación:

```
@Override
  protected void configure(HttpSecurity http) throws Exception {
    http.csrf().disable()
    .authorizeRequests()
    .antMatchers("/empleados").authenticated()
    .antMatchers("/empleados/*").hasRole("ADMIN")
    .and()
    .httpBasic();
  }
```

En primer lugar, lo que hacemos en este método es deshabilitar la protección contra ataques maliciosos csrf:

`http.csrf().disable()`

Esto se hace normalmente en servicios REST, para evitar que el cliente tenga que enviar un token de seguridad.

Después, a través de llamadas al método i*antMatchers() i***ndicamos los recursos a proteger** y qué tipo de usuarios pueden acceder a los mismos. En nuestro ejemplo indicamos que a la dirección “/empleados" puedan acceder todos los usuarios autenticados, mientras que a las direcciones que incluyan algo más en la url únicamente podrán acceder los miembros del rol admin.

Securización de microservicios

Configuración de seguridad

Es posible indicar también la **protección para un determinado método HTTP**. Por ejemplo, en el siguiente caso, solo protegemos el acceso al recurso en el caso de peticiones GET:

```
.antMatchers(HttpMethod.GET,"/empleados/*").hasRole("ADMIN")
```

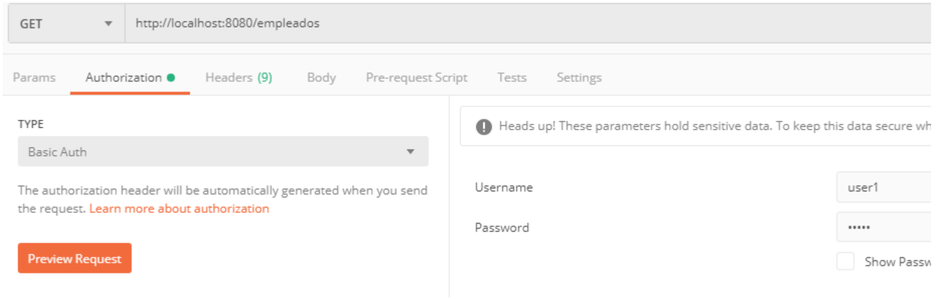
En la última instrucción, al objeto HttpSecurity devuelto por la última llamada a los métodos *antMatchers()*, le aplicamos el método *httpBasic()* para indicar que **el mecanismo de autenticación será de tipo básico**, es decir, el propio cliente utilizará su sistema predeterminado para solicitud de los credenciales.

Securización de microservicios

Testing

Si, una vez desplegado el microservicio, lanzamos la petición de uno de los recursos desde postman, recibiremos como respuesta un **error 401**, que significa que el **usuario no puede ser autenticado**. Y es que debemos enviar las creadenciales para que Spring nos pueda autenticar.

Para enviar el usuario y contraseña desde postman en una autenticación básica, nos vamos a la pestaña “Authorization” y elegimos el tipo “Basica Auth”. Al hacerlo nos aparecerán dos campos de texto para la introducción del usuario y contraseña, que serán enviados en la cabecera de la petición.



Si lanzamos la petición *http://localhost:8080/empleados*, obtendremos la lista de empleados, puesto que pueden acceder a ese recurso todos los usuarios autenticados, pero si accedemos a *http://localhost:8080/empleados/1000*, obtendremos un *error 403*, usuario no autorizado, ya que a este recurso solo pueden acceder los miembros del rol ADMIN. Si proporcionamos los credenciales admin/admin, entonces si veremos el resultado de la petición.

Securización de microservicios

Otras fuentes de usuarios

En el ejemplo que hemos analizado hemos definido los usuarios y roles en memoria. Sin embargo, en una aplicación real, eso no suele ser lo habitual. Los **usuarios suelen estar definidos en una base de datos o en un LDAP**.

En el siguiente ejemplo vemos como tendríamos que definir el método configure() para indicar que los usuarios se encuentran en una base de datos:

```
@Autowired
DataSource dataSource;
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
     auth.jdbcAuthentication().dataSource(dataSource)
          .usersByUsernameQuery("select username, password, enabled"
               + " from users where username=?")
          .authoritiesByUsernameQuery("select username, authority "
               + "from authorities where username=?");
}
```

La variable dataSource apunta a un **objeto DataSource definido mediante Spring** y que sería inyectado mediante @Autowired. A través de sendas consultas SQL, indicamos a Spring como localizar los usuarios y los roles.

Ejercicio Práctico

Vamos a securizar el acceso a nuestro microservicio de cursos, de modo que todos los usuarios del rol “COMUN” puedan realizar la recuperación y búsqueda de cursos, pero solo los del rol “PROF” puedan hacer las eliminaciones, altas y modificaciones.

Solución ejercicio

- En el siguiente video tutorial podrás ver la solución al ejercicio propuesto, con la explicación detallada de los componentes desarrollados.

VER TUTORIAL

Recuerda

- Un microservicio no es más que un servicio Web Rest, que ademas de la implementación del propio servicio incluye el entorno de ejecución del mismo para que pueda ser autodesplegado y autoejecutado.
- Spring Boot es una parte del framework Spring, ideal para la creación de microservicios ya que, además de simplificar el proceso de configuración asumiendo una serie de configuraciones por defecto, incluye en las aplicaciones generadas con el mismo una versión ligera del servidor de aplicaciones para que el servicio pueda ser ejecutado de manera independiente.
- Las aplicaciones Spring Boot no utilizan archivos .xml para su configuración, en lugar de ello, emplean ficheros .properties para proporcionar ciertas propiedades a la aplicación utilizando parejas clave=valor. También soporta la utilización de archivos .yml
- Mediante la anotación @SpringBootApplication sobre la clase principal de la aplicación Spring Boot, el framework asume una serie de configuraciones y comportamiento por defecto para la aplicación.
- Para securizar el acceso a un microservicio en Spring Boot, debemos crear una clase que herede WebSecurityConfigurerAdapter y sobrescribir en ella los métodos *configure()* en los que le indicaremos a Spring, por un lado, el repositorio de usuarios y mecanismo de autenticación a utilizar y, por otro, las políticas de autorización en el acceso a los recursos del servicio.