

Spring Boot

Creación de un app web

Sprint Boot: creación de un app web

ÍNDICE:

- 1. Creación de app web simple**
- 2. Incorporando Spring MVC + Thymeleaf**
- 3. Inyección de dependencias**
- 4. Configurando para trabajar con distintas BBDD**
- 5. Spring Data JPA**
- 6. Incorporando servicios REST**
- 7. Manejo de caché (@Cacheable)**
Actividad
- 8. Securizar con Spring Security**
- 9. Repaso de conceptos sobre servicios REST**
- 10. Creación de API REST con Spring Boot**
- 11. Definición de test unitarios**



1

Creación de
un app web
simple

1. Creación de app web simple

Qué es Spring Boot

Spring Boot es una de las tecnologías dentro del mundo de Spring de las que más se está hablando. **¿Qué es y cómo funciona Spring Boot?** Para entender el concepto primero debemos reflexionar sobre cómo construimos aplicaciones con **Spring Framework**.

Fundamentalmente existen tres pasos a realizar. El primero es crear un proyecto Maven/Gradle y descargar las dependencias necesarias. En segundo lugar desarrollamos la aplicación y en tercer lugar la desplegamos en un servidor. Si nos ponemos a pensar un poco a detalle en el tema, **únicamente el paso dos es una tarea de desarrollo**. Los otros pasos están más orientados a infraestructura.

- 1 seleccionar jars con maven
- 2 crear la aplicación
- 3 desplegar en servidor

1. Creación de app web simple

Creación de app web

Vamos a empezar creándonos una aplicación web básica que iremos ampliando durante el resto de la formación.

Primeramente abriremos **STS (Spring Tool Suite)**, el cuál nos podemos descargar de <https://spring.io/tools>. Y seleccionaremos la creación de un '**Spring Started Project**'.

Rellenaremos la siguiente información:

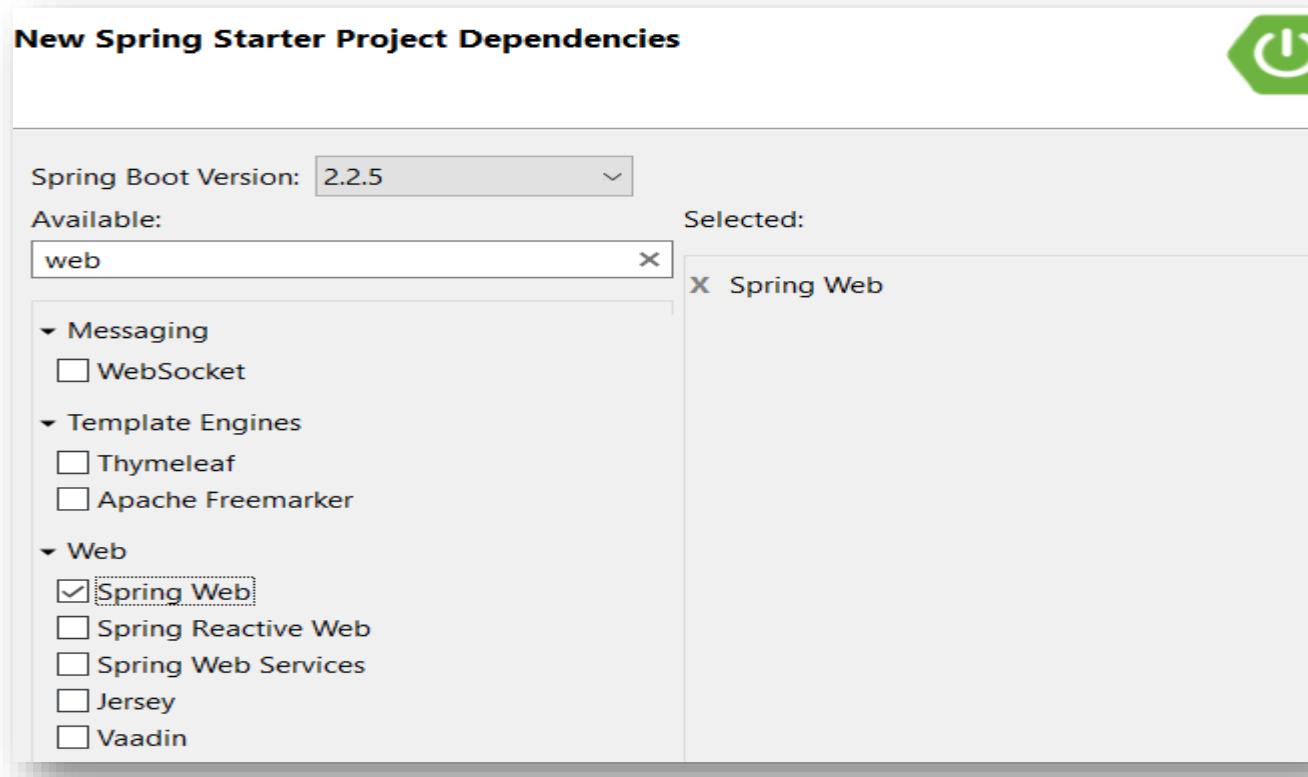
New Spring Starter Project

Service URL	<input type="text" value="https://start.spring.io"/>		
Name	<input type="text" value="DemoWeb"/>		
<input checked="" type="checkbox"/> Use default location			
Location	<input type="text" value="C:\Dev\workspace\jhernand\DemoWeb"/> <input type="button" value="Browse"/>		
Type:	<input type="text" value="Maven"/>	Packaging:	<input type="text" value="Jar"/>
Java Version:	<input type="text" value="8"/>	Language:	<input type="text" value="Java"/>
Group	<input type="text" value="everFuture"/>		
Artifact	<input type="text" value="DemoWeb"/>		
Version	<input type="text" value="0.0.1-SNAPSHOT"/>		
Description	<input type="text" value="Demo project for Spring Boot"/>		
Package	<input type="text" value="com.everis"/>		

1. Creación de app web simple

Creación de app web

De primeras vamos a seleccionar únicamente la dependencia de '**Spring Web**':

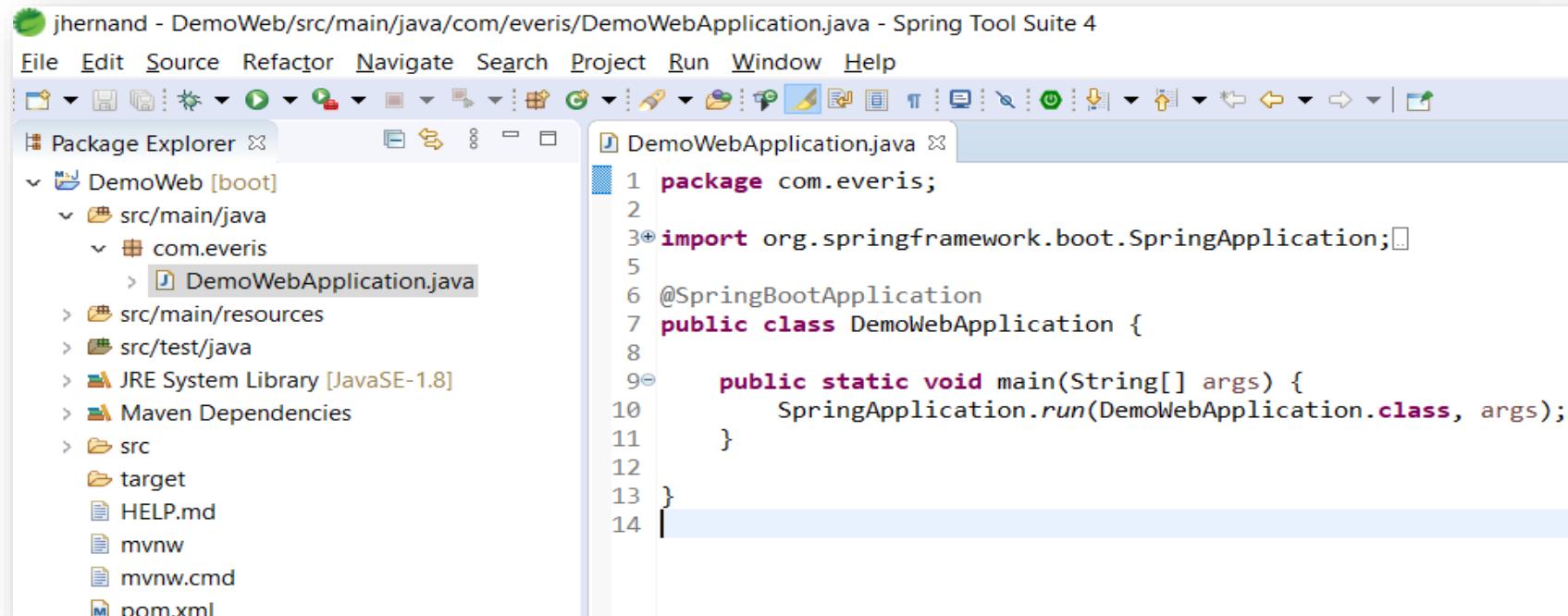


Y tras seleccionarla pulsaremos '**Finish**' para que nos cree el proyecto.

1. Creación de app web simple

Creación de app web

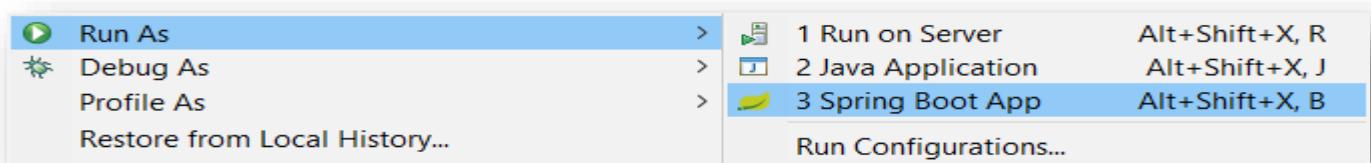
Revisamos el método 'main' creado:



The screenshot shows the Spring Tool Suite interface. On the left, the Package Explorer view displays a project named 'DemoWeb [boot]' with its structure: src/main/java/com/everis/DemoWebApplication.java, src/main/resources, src/test/java, JRE System Library [JavaSE-1.8], Maven Dependencies, src, target, HELP.md, mvnw, mvnw.cmd, and pom.xml. The right side shows the code editor for 'DemoWebApplication.java' with the following content:

```
1 package com.everis;
2
3 import org.springframework.boot.SpringApplication;
4
5 @SpringBootApplication
6 public class DemoWebApplication {
7
8     public static void main(String[] args) {
9         SpringApplication.run(DemoWebApplication.class, args);
10    }
11 }
12
13 }
14 }
```

Y la ejecutamos la aplicación como 'Spring Boot App':

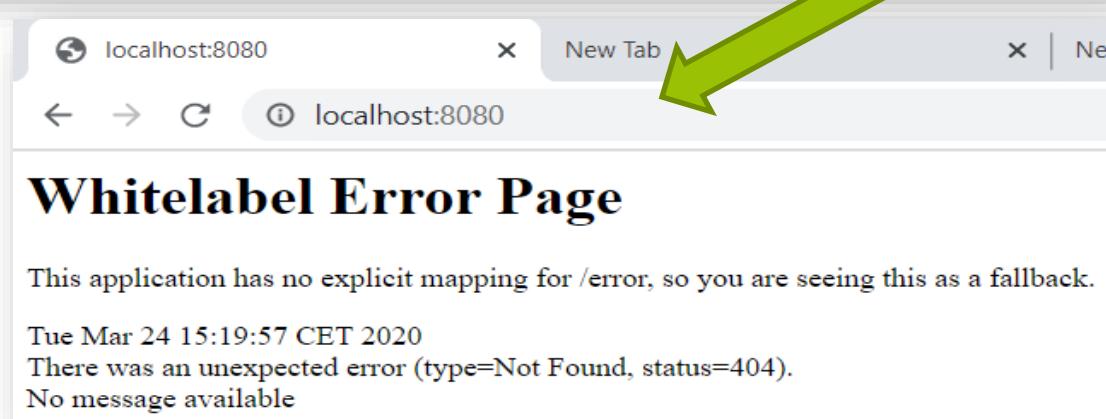


1. Creación de app web simple

Creación de app web

```
\\ / --'--( )--_ _\ \ \ \ \ 
(( )\ \ | | | | | | | \ \ \ \
\ \ \ \ | | | | | | | | | ) ) ) )
' | | | | | | | | | | | | | |
=====| | ======| | | | | | | | | |
:: Spring Boot ::      (v2.2.5.RELEASE)
```

```
2020-03-24 15:16:11.029 INFO 12860 --- [           main] com.everis.DemoWebApplication      : Starting DemoWebApplication on MUR-3W52SF2 with PID 12860 (C:\Dev\
2020-03-24 15:16:11.033 INFO 12860 --- [           main] com.everis.DemoWebApplication      : No active profile set, falling back to default profiles: default
2020-03-24 15:16:14.192 INFO 12860 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2020-03-24 15:16:14.205 INFO 12860 --- [           main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2020-03-24 15:16:14.205 INFO 12860 --- [           main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.31]
2020-03-24 15:16:15.141 INFO 12860 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]    : Initializing Spring embedded WebApplicationContext
2020-03-24 15:16:15.142 INFO 12860 --- [           main] o.s.web.context.ContextLoader       : Root WebApplicationContext: initialization completed in 4051 ms
2020-03-24 15:16:15.603 INFO 12860 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor: Initializing ExecutorService 'applicationTaskExecutor'
2020-03-24 15:16:16.023 INFO 12860 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2020-03-24 15:16:16.027 INFO 12860 --- [           main] com.everis.DemoWebApplication      : Started DemoWebApplication in 5.929 seconds (JVM running for 9.91)
```



Esta es la página por defecto, no quiere decir que hayamos hecho algo mal

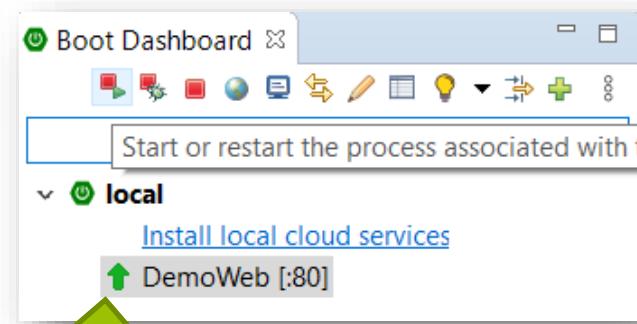
1. Creación de app web simple

Creación de app web

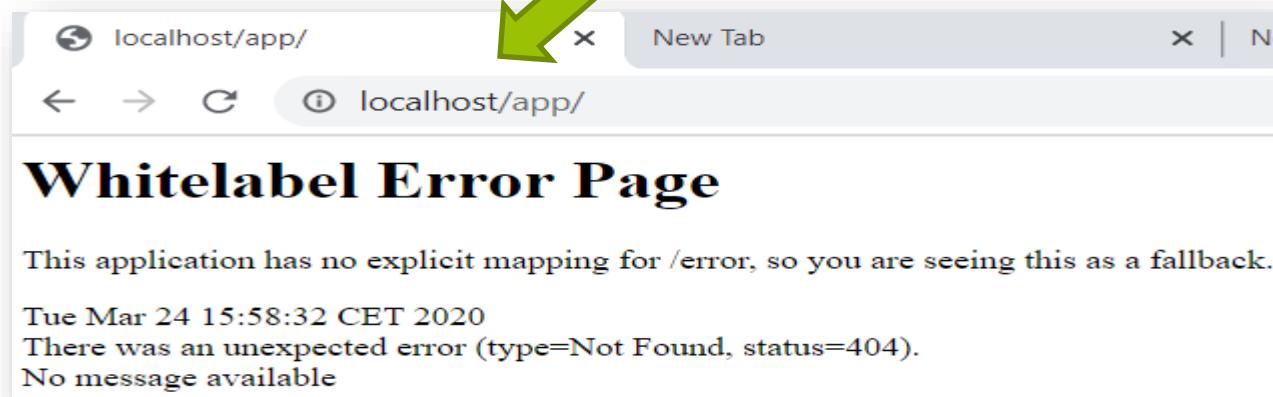
Vamos a cambiar el puerto y el contexto para que arranque en localhost/app, así modificaremos el fichero **application.properties** de la ruta **src/main/resources**:



```
application.properties
1 server.port=80
2 server.servlet.context-path=/app
```



Tras ello rearrancamos la aplicación:



A large green arrow points from the Boot Dashboard window above down to the browser window below. The browser window displays the Whitelabel Error Page with the message: "This application has no explicit mapping for /error, so you are seeing this as a fallback." Below this, it shows the timestamp "Tue Mar 24 15:58:32 CET 2020" and the error details: "There was an unexpected error (type=Not Found, status=404). No message available".



1. Creación de app web simple

Creación de app web

Pero, ¿a cada cambio hay que estar parando y arrancando el servidor?

Si no queremos no.

Vamos a añadir en el pom.xml el siguiente código y rearrancaremos el proyecto:

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

Nota: cada vez que incorporemos una librería, si no la hemos usado antes deberemos decirle a Maven que se la descargue → Botón derecho sobre mi proyecto: Maven → ‘Update Project...’ → ‘OK’

Una vez arrancado hacemos cualquier modificación (cambiamos el puerto o en cualquier clase java metemos por ejemplo un salto de línea) y vemos como la aplicación redespliega automáticamente:

```
2020-03-24 15:41:35.638 INFO 11056 --- [ restartedMain] com.everis.DemoWebApplication      : Starting DemoWebApplication on MUR-3W52SF2 with PID 11056 (C:\Dev\)
2020-03-24 15:41:35.638 INFO 11056 --- [ restartedMain] com.everis.DemoWebApplication      : No active profile set, falling back to default profiles: default
2020-03-24 15:41:35.918 INFO 11056 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 80 (http)
2020-03-24 15:41:35.918 INFO 11056 --- [ restartedMain] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2020-03-24 15:41:35.919 INFO 11056 --- [ restartedMain] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.31]
2020-03-24 15:41:35.997 INFO 11056 --- [ restartedMain] o.a.c.c.C.[Tomcat].[localhost].[/]    : Initializing Spring embedded WebApplicationContext
2020-03-24 15:41:35.997 INFO 11056 --- [ restartedMain] o.s.web.context.ContextLoader       : Root WebApplicationContext: initialization completed in 356 ms
2020-03-24 15:41:36.079 INFO 11056 --- [ restartedMain] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2020-03-24 15:41:36.115 INFO 11056 --- [ restartedMain] o.s.b.d.a.OptionalLiveReloadServer     : LiveReload server is running on port 35729
2020-03-24 15:41:36.131 INFO 11056 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 80 (http) with context path ''
2020-03-24 15:41:36.132 INFO 11056 --- [ restartedMain] com.everis.DemoWebApplication      : Started DemoWebApplication in 0.52 seconds (JVM running for 20.463)
2020-03-24 15:41:36.135 INFO 11056 --- [ restartedMain] .ConditionEvaluationDeltaLoggingListener : Condition evaluation unchanged
```



2

Incorporando Spring MVC y Thymeleaf

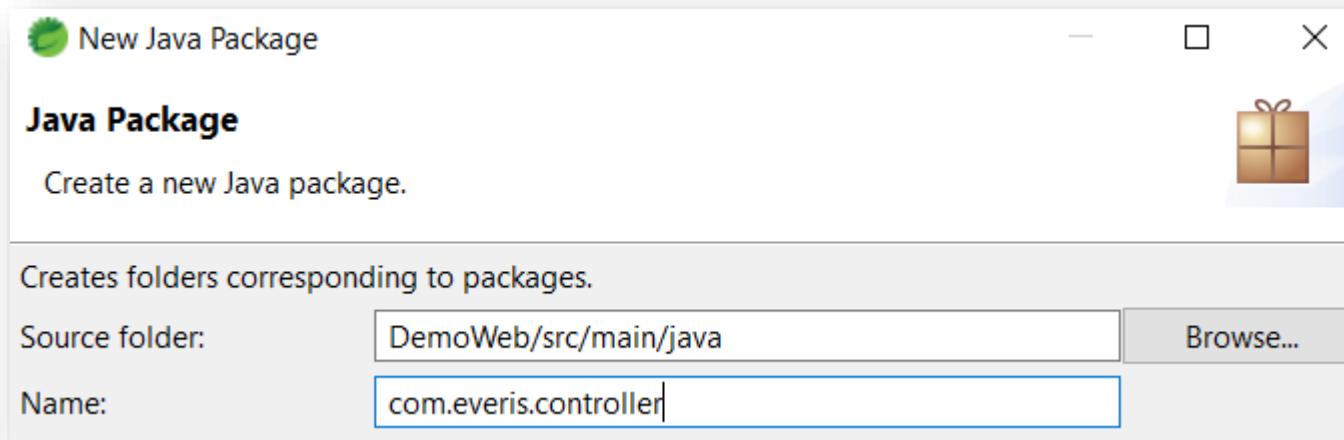
2. Incorporando Spring MVC + Thymeleaf

Thymeleaf

Vamos a incorporar el motor de plantillas **Thymeleaf** <https://www.thymeleaf.org>, más práctico que el tradicional JSP. al proyecto para desarrollar más fácil nuestras páginas web. Para ello incorporaremos al **pom.xml**

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

Tras ello crearemos el paquete **com.everis.controller**:

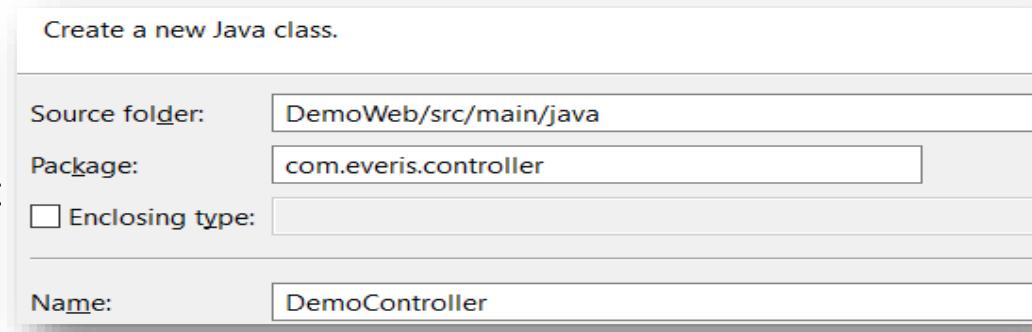


2. Incorporando Spring MVC + Thymeleaf

Spring MVC

Dentro crearemos la clase **DemoController**:

Que va ser el controlador de Spring MVC encargado de atender las peticiones a **/saludo**



Ahora escribimos el siguiente código en nuestra clase **DemoController.java**:

```
package com.everis.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class DemoController {

    @GetMapping("/saludo")
    public String greeting(@RequestParam(name="name", required=false, defaultValue="Mundo") String name, Model model) {
        model.addAttribute("name", name);
        return "hola";
    }
}
```

The code is annotated with Spring MVC annotations: '@Controller' on the class, '@GetMapping' with the path '/saludo' on the method, and '@RequestParam' with the name 'name' and default value 'Mundo' on the parameter.

2. Incorporando Spring MVC + Thymeleaf

Thymeleaf

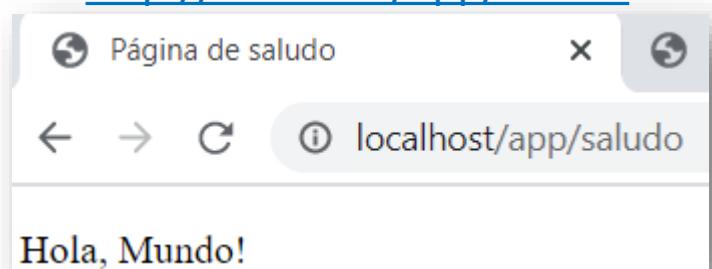
Tras ello crearemos en la carpeta **src/main/resources/templates** un documento html llamado **hola.html** con el siguiente contenido:

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org"> ←
<head>
    <title>Página de saludo</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
<body>
    <p th:text="'Hola, ' + ${name} + '!'>
</body>
</html>
```

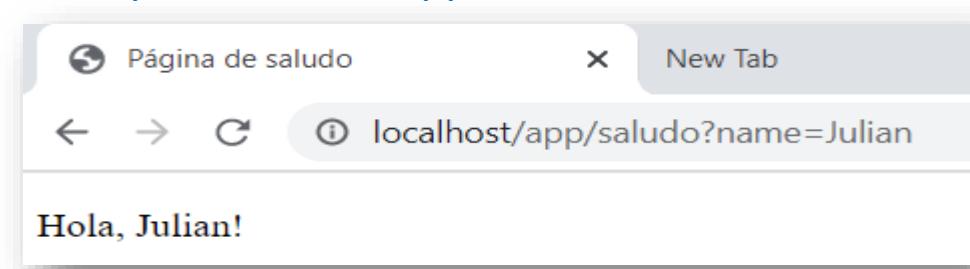


Si accedemos ahora a:

<http://localhost/app/saludo>



<http://localhost/app/saludo?name=Julian>



2. Incorporando Spring MVC + Thymeleaf

Spring MVC Thymeleaf

Vamos a **personalizar ahora la página de error** de nuestro aplicativo.

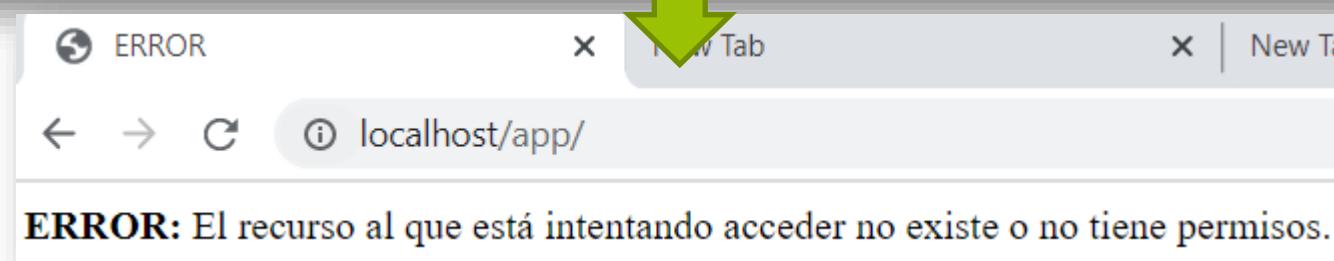
Para ello:

- 1) Añadimos el siguiente mapeo en el **DemoController**:

```
@GetMapping("/error")
public String error_page() {
    return "error";
}
```

- 2) Implementaremos la página src/main/resources/templates/error.html:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>ERROR</title>
</head>
<body>
    <b>ERROR:</b> El recurso al que está intentando acceder no existe o no tiene permisos.
</body>
</html>
```



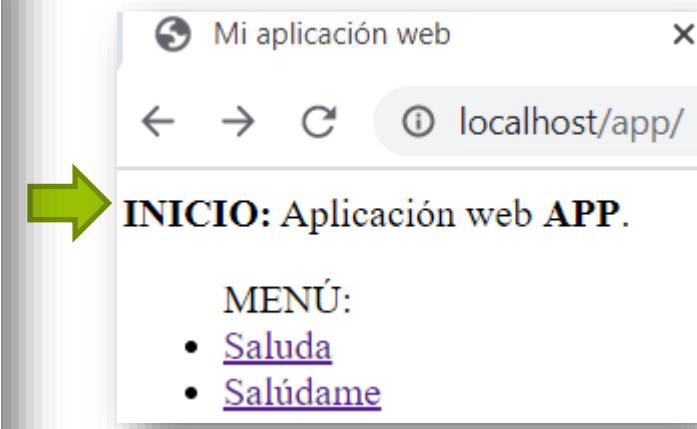
2. Incorporando Spring MVC + Thymeleaf

Spring MVC

Ahora nos quedará añadir una página de inicio para nuestra aplicación web.

Crearemos la página src/main/resources/templates/index.html y al ser una página por defecto no vamos a necesitar crear redirección en el controller:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Mi aplicación web</title>
</head>
<body>
    <b>INICIO:</b> Aplicación web <b>APP</b>.
    <ul> MENÚ:
        <li> <a href="/app/saludo">Saluda</a> </li>
        <li> <a href="/app/saludo?name=Julian">Salúdame</a> </li>
    </ul>
</body>
</html>
```





3

Inyección de dependencias

3. Inyección de dependencias

Contexto

Hasta ahora hemos creado un sitio web y hemos visto cómo crear una página de inicio, una de error y páginas que recojan parámetros (gestionadas en un **controller de Spring MVC**).

Ahora vamos a enlazar la lógica de negocio y los objetos java para poder utilizarlos en nuestras páginas, para ello aprovecharemos para mostrar la inyección de dependencias de Spring.

La funcionalidad va a consistir en tener un método ‘registrar’ que se va a encargar de mostrar a qué empleados se está saludando al llamar a nuestro método ‘greeting’ de nuestro

DemoController:

```
@GetMapping("/saludo")
public String greeting(@RequestParam(name="name"
    model.addAttribute("name", name);
    empleadoService.registrar(name);
    return "hola";
}
```

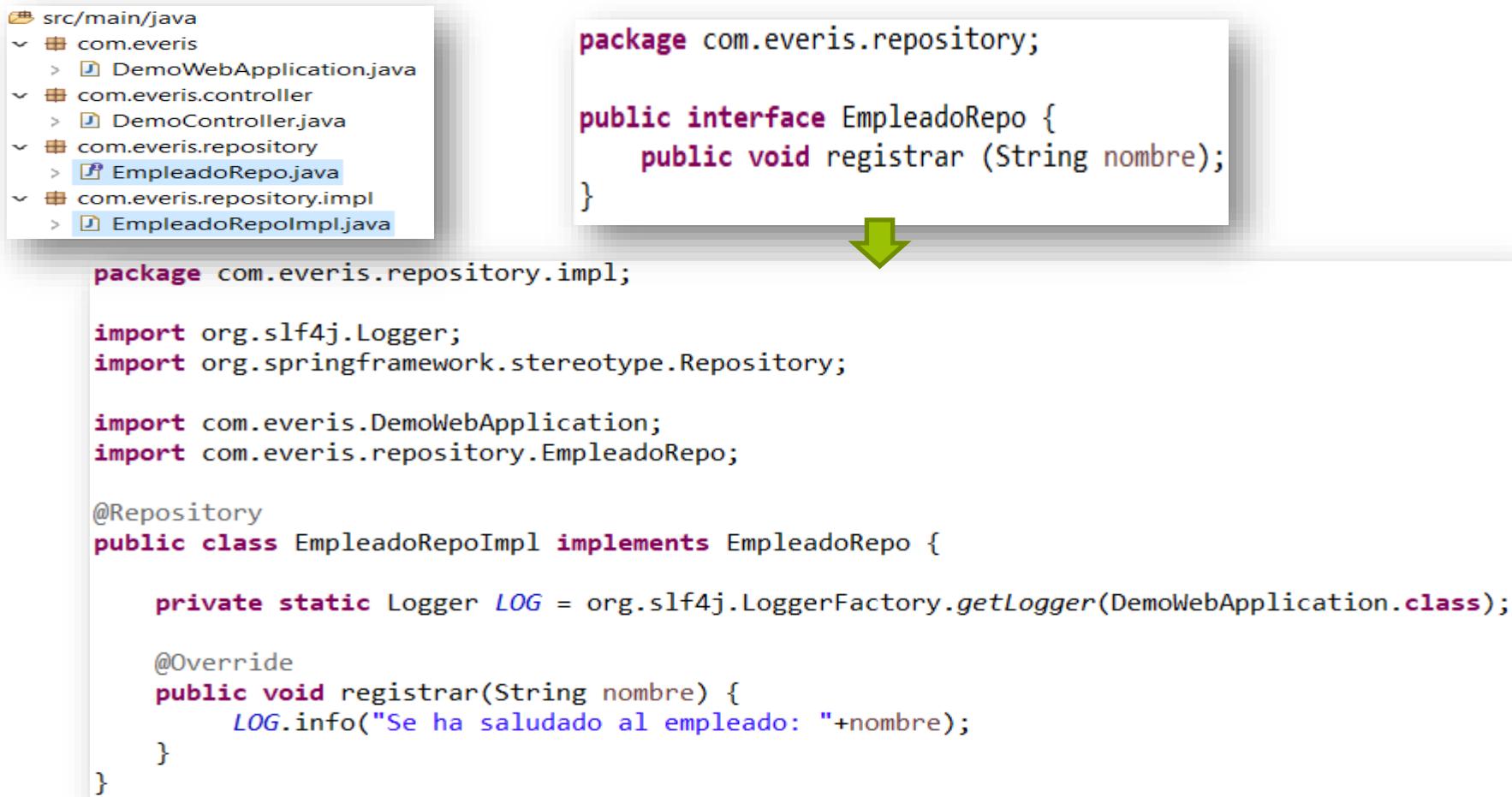


```
INFO 22332 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 80 (http) with context path '/app'
INFO 22332 --- [ restartedMain] com.everis.DemoWebApplication      : Started DemoWebApplication in 0.33 seconds (JVM running for 7402.001)
INFO 22332 --- [ restartedMain] .ConditionEvaluationDeltaLoggingListener : Condition evaluation unchanged
INFO 22332 --- [p-nio-80-exec-1] o.a.c.c.C.[Tomcat-2].[localhost].[/app]   : Initializing Spring DispatcherServlet 'dispatcherServlet'
INFO 22332 --- [p-nio-80-exec-1] o.s.web.servlet.DispatcherServlet       : Initializing Servlet 'dispatcherServlet'
INFO 22332 --- [p-nio-80-exec-1] o.s.web.servlet.DispatcherServlet       : Completed initialization in 7 ms
INFO 22332 --- [p-nio-80-exec-1] com.everis.DemoWebApplication      : Se ha saludado al empleado: Mundo
INFO 22332 --- [p-nio-80-exec-3] com.everis.DemoWebApplication      : Se ha saludado al empleado: Julian
```

3. Inyección de dependencias

@Repository

Primero vamos a crear la clase e interfaz que van a simular la capa 'repository' y que tendrán el método 'registrar' que escriba por log a quién se ha saludado:



```
src/main/java
  com.everis
    DemoWebApplication.java
  com.everis.controller
    DemoController.java
  com.everis.repository
    EmpleadoRepo.java
  com.everis.repository.impl
    EmpleadoRepoImpl.java
```

```
package com.everis.repository;

public interface EmpleadoRepo {
    public void registrar (String nombre);
}
```

```
package com.everis.repository.impl;

import org.slf4j.Logger;
import org.springframework.stereotype.Repository;

import com.everis.DemoWebApplication;
import com.everis.repository.EmpleadoRepo;

@Repository
public class EmpleadoRepoImpl implements EmpleadoRepo {

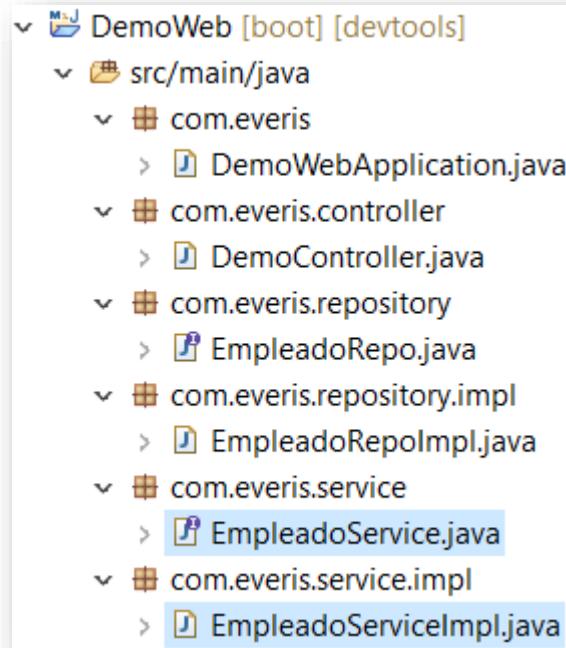
    private static Logger LOG = org.slf4j.LoggerFactory.getLogger(DemoWebApplication.class);

    @Override
    public void registrar(String nombre) {
        LOG.info("Se ha saludado al empleado: "+nombre);
    }
}
```

3. Inyección de dependencias

@Service

Una vez que tenemos la capa de acceso a datos (Repository), vamos a implementar la capa donde va la lógica de negocio (Service):



```
package com.everis.service;

public interface EmpleadoService {
    public void registrar (String name);
}
```

```
package com.everis.service.impl;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.everis.repository.EmpleadoRepo;
import com.everis.service.EmpleadoService;

@Service
public class EmpleadoServiceImpl implements EmpleadoService{

    @Autowired
    EmpleadoRepo empleadoRepo;

    @Override
    public void registrar(String name) {
        empleadoRepo.registrar(name);
    }
}
```

3. Inyección de dependencias

@Autowired

Y tras ello sólo nos quedará llamar a la funcionalidad desde el **controller**:

```
@Controller
public class DemoController {

    @Autowired
    EmpleadoService empleadoService;

    @GetMapping("/saludo")
    public String greeting(@RequestParam(name="name", required=false, defaultValue="Mundo") String name, Model model) {
        model.addAttribute("name", name);
        empleadoService.registrar(name);
        return "hola";
    }

    @GetMapping("/error")
    public String error_page() {
        return "error";
    }
}
```

<http://localhost/app/saludo?name=Julian>

[p-nio-80-exec-1] o.a.c.c.C.[Tomcat-2].[localhost].[/app] : Initializing Spring DispatcherServlet 'dispatcherServlet'
[p-nio-80-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
[p-nio-80-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 4 ms
[p-nio-80-exec-1] com.everis.DemoWebApplication : Se ha saludado al empleado: Julian



4

Configurando
para trabajar
con distintas
BBDD

4. Conectando con BBDD

Vamos a habilitar nuestra aplicación para conectarse con bases de datos, pero lo vamos a hacer buscando una configuración práctica que pueda sernos útil en algunos proyectos.

Para empezar vamos a imaginar que en los entornos de clientes tenemos una BBDD MySQL, lo que tendríamos que añadir a nuestro código para trabajar con MySQL sería lo siguiente:

En el fichero **pom.xml**:

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
```

En el fichero **src/main/resources/application.properties**:

```
spring.datasource.url=jdbc:mysql://${MYSQL_HOST}:localhost:3306/db_example
spring.datasource.username=user_bbdd
spring.datasource.password=password_bbdd
```

4. Conectando con BBDD

Hay ocasiones en que nos gustaría tener la BBDD en local para realizar ciertas operaciones, pero es costoso instalarse un servidor en local (MySQL en nuestro caso) y luego crear toda la estructura de tablas y relaciones.

Nosotros lo que vamos a hacer es permitir que nuestra aplicación se conecte a una BBDD en memoria que vamos a tener en local, para poder realizar cierto tipo de pruebas:

En el fichero **pom.xml**:

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

En el fichero **src/main/resources/application.properties**:

```
spring.datasource.url=jdbc:h2:file:C:/h2/springboot2
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=us
spring.datasource.password=pa
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.h2.console.path=/h2-console
spring.jpa.hibernate.ddl-auto=update
spring.h2.console.enabled=true
```

Nota:
Necesitamos que exista
el directorio **c:/h2**

4. Conectando con BBDD

En el siguiente capítulo vamos a ver JPA, pero antes aquí vamos a añadírselo al proyecto para ver parte de la ‘magia’ que nos va a aportar.

- 1) Primero añadimos la dependencia al **pom.xml**:

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

- 2) Y por último nos crearemos la siguiente clase
Empleado.java:
(a la que faltarán añadirle los getter/setters)

```
package com.everis.repository.entity;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table
public class Empleado {
    @Id
    @Column
    private Integer id;

    @Column(nullable = false, length=30)
    private String nombre;

    @Column
    private String apellidos;
}
```

4. Conectando con BBDD

Si nos fijamos, al arrancar la aplicación ahora nos indica:

```
INFO 3364 --- [ restartedMain] o.s.web.context.ContextLoader      : Root WebApplicationContext: initialization completed in 1281 ms
INFO 3364 --- [ restartedMain] com.zaxxer.hikari.HikariDataSource   : HikariPool-1 - Starting...
INFO 3364 --- [ restartedMain] com.zaxxer.hikari.HikariDataSource   : HikariPool-1 - Start completed.
INFO 3364 --- [ restartedMain] o.s.b.a.h2.H2ConsoleAutoConfiguration : H2 console available at '/h2-console'. Database available at 'jdbc:
INFO 3364 --- [ restartedMain] o.hibernate.jpa.internal.util.LogHelper  : HHH000204: Processing PersistenceUnitInfo [name: default]
INFO 3364 --- [ restartedMain] org.hibernate.Version                  : HHH000412: Hibernate ORM core version 5.4.12.Final
INFO 3364 --- [ restartedMain] o.hibernate.annotations.common.Version : HCANN000001: Hibernate Commons Annotations {5.1.0.Final}
INFO 3364 --- [ restartedMain] org.hibernate.dialect.Dialect        : HHH000400: Using dialect: org.hibernate.dialect.H2Dialect
INFO 3364 --- [ restartedMain] o.h.e.t.j.p.i.JtaPlatformInitiator    : HHH000490: Using JtaPlatform implementation: [org.hibernate.engine.jta.platform.internal.NoJtaPlatform]
INFO 3364 --- [ restartedMain] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
INFO 3364 --- [ restartedMain] o.s.b.d.a.OptionalLiveReloadServer   : LiveReload server is running on port 35729
WARN 3364 --- [ restartedMain] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries will be executed via JPA, even if a JSTL-based view is rendered (e.g. Spring MVC)
INFO 3364 --- [ restartedMain] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
INFO 3364 --- [ restartedMain] o.s.b.a.w.s.WelcomePageHandlerMapping  : Adding welcome page template: index
INFO 3364 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 80 (http) with context path '/app'
INFO 3364 --- [ restartedMain] com-everis-DemoWebApplication       : Started DemoWebApplication in 3.358 seconds (JVM running for 4.306)
```

Esto es porque se nos ha habilitado una consola web para poder acceder nuestra BBDD h2.

Desde ella vamos a poder tener una especie de gestor web de BBDD donde vamos a poder ver las tablas, registro, etc e incluso crear y modificar datos, así como tablas.

4. Conectando con BBDD

Si accedemos a la url <http://localhost/app/h2-console> con los datos de nuestra BBDD:

localhost/app/h2-console/

Español Preferencias Tools Ayuda

Registrar

Configuraciones guardadas: Generic H2 (Embedded) REST

Nombre de la configuración: Generic H2 (Embedded) REST Guardar Eliminar

Controlador: org.h2.Driver

URL JDBC: jdbc:h2:file:C:/h2/springboot2

Nombre de usuario: us

Contraseña: ..

Conectar Probar la conexión

Contraseña = pa

4. Conectando con BBDD

Al entrar nos encontramos que JPA además de habilitarnos la consola web, nos ha creado ya la tabla 'empleado' en base a las características que le hemos indicado en la clase java **com.everis.repository.entity.Empleado**:

The screenshot shows the H2 Database Console interface. On the left, the database tree displays 'jdbc:h2:file:C:/h2/springboot2' as the current connection, with 'EMPLEADO' expanded to show columns 'ID', 'APELLODOS', and 'NOMBRE'. Other nodes include 'INFORMATION_SCHEMA' and 'Users'. At the bottom, a version message 'H2 1.4.200 (2019-10-14)' is visible.

The main area contains a SQL editor with the following content:

```
Run Selected Auto complete Clear SQL statement:  
SELECT * FROM EMPLEADO|
```

Below the editor, the results of the executed query are shown:

```
SELECT * FROM EMPLEADO;  
ID APELLIDOS NOMBRE  
(no rows, 5 ms)
```

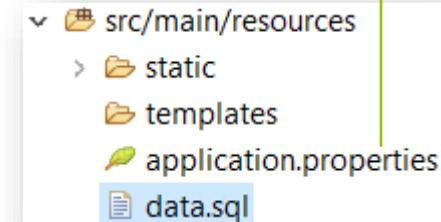
An 'Edit' button is located at the bottom of the results panel.

4. Conectando con BBDD

Ahora creamos el fichero /src/main/resources/**data.sql** añadimos las siguientes líneas:

```
insert into empleado (id, nombre, apellidos)
select 1, 'Rocío', 'De la O' from dual where not exists (select 1 from empleado where id = 1);

insert into empleado (id, nombre, apellidos)
select 2, 'Alberto', 'Del Monte' from dual where not exists (select 1 from empleado where id = 2);
```



Y al entrar de nuevo en la consola web veremos los registros creados:

The screenshot shows the H2 Database Console interface. On the left, the database schema is visible with tables 'EMPLEADO', 'INFORMATION_SCHEMA', and 'Users'. The central area contains a SQL editor with the following content:

```
jdbc:h2:file:C:/h2/springboot2
EMPLEADO
INFORMATION_SCHEMA
Users
H2 1.4.200 (2019-10-14)

Run Run Selected Auto complete Clear SQL statement:
SELECT * FROM EMPLEADO
```

Below the editor, the results of the query are displayed in a table:

ID	APELLIDOS	NOMBRE
1	De la O	Rocío
2	Del Monte	Alberto

A green arrow points from the bottom table back up towards the 'data.sql' file in the file tree on the right.



5

Spring Data JPA

5. Spring Data JPA

Ya hemos visto algunas ventajas que nos ha aportado JPA.

En el apartado anterior hemos visto cómo ha ayudado a levantar una consola web para nuestra base de datos h2 y luego cómo ha creado las tablas correspondientes a nuestras clases java que hemos marcado con las anotaciones correspondientes. En nuestro caso ha sido la **tabla Empleado**:

```
@Entity  
@Table  
public class Empleado {
```

Todo esto ha sido gracias a la configuración que hemos definido en el fichero **application.properties**:

```
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect  
spring.h2.console.enabled=true  
spring.h2.console.path=/h2-console  
spring.jpa.hibernate.ddl-auto=update
```

5. Spring Data JPA

Ahora vamos a ampliar nuestro ejemplo para permitirnos el acceso a la base de datos.

Vamos a implementar una página que nos muestre un listado de los empleados que tenemos en base de datos.

Para ello el primer paso va a ser crearnos un `@Repository` que nos va a permitir acceder a los métodos JPA de acceso a BBDD:

```
package com.everis.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import com.everis.repository.entity.Empleado;

@Repository
public interface EmpleadoRepoJPA extends JpaRepository<Empleado, Integer>{}
```

Sólo con haber creado esta interface, ahora vamos a poder consultar, insertar, etc...

5. Spring Data JPA

Como siguiente paso vamos a implementar el método correspondiente en la capa de servicios.

Para ello primeros daremos de alta un método listar en la **interface**:

```
package com.everis.service;

import java.util.List;

import com.everis.repository.entity.Empleado;

public interface EmpleadoService {
    public void registrar (String name);
    public List<Empleado> listar ();
}
```

Y en la implementación del servicio modificamos para que apunte al **@Repository** último creado. Como nos fallará la llamada al método ‘**registrar**’ comentamos esa línea y ya la arreglaremos más adelante:

```
@Autowired
EmpleadoRepoJPA empleadoRepo;
```

5. Spring Data JPA

Una vez creado, implementamos en el **@Service** nuestro método listar:

```
package com.everis.service.impl;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.everis.repository.EmpleadoRepoJPA;
import com.everis.repository.entity.Empleado;
import com.everis.service.EmpleadoService;

@Service
public class EmpleadoServiceImpl implements EmpleadoService{

    @Autowired
    EmpleadoRepoJPA empleadoRepo;

    @Override
    public void registrar(String name) {
        //empleadoRepo.registrar(name);
    }

    @Override
    public List<Empleado> listar() {
        return empleadoRepo.findAll();
    }
}
```

5. Spring Data JPA

Tras esto crearemos un nuevo mapeo en **DemoController**:

```
@GetMapping("/listarEmpleados")
public String listarEmp(Model model) {
    model.addAttribute ("listaEmp", empleadoService.listar() );
    return "listarDeEmpleados";
}
```

Y por último la página web **./templates/listarDeEmpleados.html** que mostrará el listado y el link a la misma en la página **index.html**:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Mi aplicación web</title>
</head>
<body>
    <b>INICIO:</b> Aplicación web <b>APP</b>.
    <ul> MENU:
        <li> <a href="/app/saludo">Saluda</a> </li>
        <li> <a href="/app/saludo?name=Julian">Salúdame</a> </li>
        <li> <a href="/app/listarEmpleados">Lista de empleados</a> </li>
    </ul>
</body>
</html>
```

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Lista de empleados</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
<body>
Lista de empleados:

```

Nota: separar espacios de ... th:

5. Spring Data JPA

Al acceder ahora a <http://localhost/app> obtendremos:

INICIO: Aplicación web APP.

MENÚ:

- [Saluda](#)
- [Salúdame](#)
- [Lista de empleados](#)

Lista de empleados

Lista de empleados:

1 Rocío De la O
2 Alberto Del Monte

5. Spring Data JPA

Gracias a **EmpleadoRepoJPA** ya tenemos implementadas las funciones de inserción, consultas, modificación y borrado de nuestra entidad (POJO) Empleado en BBDD.

Pero ¿qué sucede si quiero personalizar alguna función de bbdd, cómo por ejemplo implementar un método que me devuelva sólo los Empleados cuyo nombre contenga un texto?

Pues para ello tenemos primero que dar de alta dicho método en la interfaz **EmpleadoRepo**:

```
package com.everis.repository;

import java.util.List;
import com.everis.repository.entity.Empleado;

public interface EmpleadoRepo {
    public void registrar (String nombre);
    public List<Empleado> listarCuyoNombreContiene(String texto_nombre);
}
```

Y tras ello hacemos que nuestra interfaz **EmpleadoRepoJPA** herede también de **EmpleadoRepo**:

```
@Repository
public interface EmpleadoRepoJPA extends JpaRepository<Empleado, Integer>, EmpleadoRepo{}
```

5. Spring Data JPA

Ello nos va a permitir implementar cualquier método personalizado, así como también hacer uso del contexto **EntityManager** para ejecutar las operaciones de BBDD que necesitemos:

```
@Repository
public class EmpleadoRepoImpl implements EmpleadoRepo {

    private static Logger LOG = org.slf4j.LoggerFactory.getLogger(DemoWebApplication.class);

    @PersistenceContext
    EntityManager entityManager;

    @Override
    public void registrar(String nombre) {
        LOG.info("Se ha saludado al empleado: " + nombre);
    }

    @Override
    public List<Empleado> listarCuyoNombreContiene(String texto_nombre) {
        Query query = entityManager.createNativeQuery("SELECT * FROM empleado " +
            "WHERE nombre LIKE ?", Empleado.class);
        query.setParameter(1, "%" + texto_nombre + "%");
        return query.getResultList();
    }
}
```

5. Spring Data JPA

Ahora ya en el **Service** podemos utilizar la llamada al método **registrar** del Repositorio y crearnos la llamada al nuevo método **listarCuyoNombreContiene**:

```
public interface EmpleadoService {  
    public void registrar (String name);  
    public List<Empleado> listar ();  
    public List<Empleado> listarFiltroNombre(String cad);
```

```
@Service  
public class EmpleadoServiceImpl implements EmpleadoService{  
  
    @Autowired  
    EmpleadoRepoJPA empleadoRepo;  
  
    @Override  
    public void registrar(String name) {  
        empleadoRepo.registrar(name);  
    }  
  
    @Override  
    public List<Empleado> listar() {  
        return empleadoRepo.findAll();  
    }  
  
    @Override  
    public List<Empleado> listarFiltroNombre(String cad) {  
        return empleadoRepo.listarCuyoNombreContiene(cad);  
    }  
}
```

5. Spring Data JPA

Tras ello modificaremos el método **listarEmp** del **controller** incluyendo la nueva llamada:

```
@GetMapping("/listarEmpleados")
public String listarEmp(Model model) {
    model.addAttribute ("listaEmp", empleadoService.listar() );
    model.addAttribute ("listaEmpConE", empleadoService.listarFiltroNombre("e") );
    return "listarDeEmpleados";
}
```

Y **listarDeEmpleados.html**:

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Lista de empleados</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
<body>
    Lista de empleados:
    <table>
        <th:block th:each="per : ${listaEmp}">
            <tr>
                <td th:text="${per.id}"></td>
                <td th:text="${per.nombre}"></td>
                <td th:text="${per.apellidos}"></td>
            </tr>
        </th:block>
    </table> <br/>
    Lista de empleados que contenga una 'e' en su nombre:
    <table>
        <th:block th:each="per : ${listaEmpConE}">
            <tr>
                <td th:text="${per.id}"></td>
                <td th:text="${per.nombre}"></td>
                <td th:text="${per.apellidos}"></td>
            </tr>
        </th:block>
    </table>
</body>
</html>
```

5. Spring Data JPA

Volvemos a acceder a <http://localhost/app> obtendremos:

The image shows two browser tabs side-by-side. The left tab, titled 'Mi aplicación web' with the URL 'localhost/app', displays the application's homepage with a menu containing links to 'Saluda', 'Salúdame', and 'Lista de empleados'. A large green arrow points from this tab to the right tab. The right tab, titled 'Lista de empleados' with the URL 'localhost/app/listarEmpleados', displays a list of employees: '1 Rocío De la O' and '2 Alberto Del Monte'. Below this, it shows a filtered list: 'Lista de empleados que contenga una 'e' en su nombre:' followed by '2 Alberto Del Monte'.

INICIO: Aplicación web APP.

MENÚ:

- [Saluda](#)
- [Salúdame](#)
- [Lista de empleados](#)

Lista de empleados:

1 Rocío De la O
2 Alberto Del Monte

Lista de empleados que contenga una 'e' en su nombre:

2 Alberto Del Monte

5. Spring Data JPA

Pero recurrir a una query nativa debe ser cuando necesitemos una consulta muy específica, ya que si queremos realizar una consulta sobre los campos de nuestra tabla, vamos a poder decírselo directamente a JPA creando un método cuyo nombre indique la consulta que se quiere realizar:

Keyword	Sample	JPQL snippet
And	findByLastnameAndFirstname	... where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	... where x.lastname = ?1 or x.firstname = ?2
Between	findByStartDateBetween	... where x.startDate between ?1 and ?2
LessThan	findByAgeLessThan	... where x.age < ?1
GreaterThan	findByAgeGreaterThan	... where x.age > ?1
IsNull	findByAgeIsNull	... where x.age is null
IsNotNull, NotNull	findByAge(Is)NotNull	... where x.age not null
Like	findByFirstnameLike	... where x.firstname like ?1
NotLike	findByFirstnameNotLike	... where x.firstname not like ?1
OrderBy	findByAgeOrderByLastnameDesc	... where x.age = ?1 order by x.lastname desc
Not	findByLastnameNot	... where x.lastname <> ?1
In	findByAgeIn(Collection<Age> ages)	... where x.age in ?1
NotIn	findByAgeNotIn(Collection<Age> age)	... where x.age not in ?1

```
public interface UserRepository extends JpaRepository<User, Long> {  
    List<User> findByLastname(String lastname);  
}
```

5. Spring Data JPA

Métodos de consulta a partir de su nomenclatura

Es posible generar métodos de consulta para las propiedades añadiéndolos en el repositorio.

Se basa en el siguiente algoritmo

- La nomenclatura del método de consulta debe ser
 - `findByPropiedad1AndPropiedad2...AndPropiedadN`
 - Ej: `List<Cliente> findByNombre(String nombre);`
`List<Cliente> findByNombreAndApellidos(String nombre, String apellidos);`
- En caso de realizar una ejecución sobre una propiedad que no existe se obtiene una excepción en tiempo de ejecución

`org.springframework.data.mapping.PropertyReferenceException: No property nombres found for type
com.everis.ejemploSpringJpaRepository.model.Cliente`

- Pueden realizarse consultas no case sensitive (añadiendo sufijo `IgnoreCase`) y por búsqueda parcial (`Like`).
 - Ej: `findByNombreIgnoreCase(String nombre);` o `findByNombreLike(String nombre);`

5. Spring Data JPA

Así vamos a definir ahora un método de consulta y vamos a dejar que JPA lo implemente. Primero vamos a insertar dos nuevos empleados en nuestra BBDD (desde el h2-console):

```
insert into empleado (id, nombre, apellidos) values (3, 'Lucía', 'Ricarti');  
insert into empleado (id, nombre, apellidos) values (4, 'Roberto', 'Sánchez');
```

SELECT * FROM EMPLEADO;		
ID	APELLIDOS	NOMBRE
1	De la O	Rocío
2	Del Monte	Alberto
3	Ricarti	Lucía
4	Sánchez	Roberto

Tras ello vamos a definir un método que nos devuelva los empleados cuyo ID sea mayor de 2 y que contengan una 'o' en el nombre.

Así en **EmpleadoRepoJPA** declararemos el método **findByIdGreaterThanOrNombreLike**:

```
@Repository  
public interface EmpleadoRepoJPA extends JpaRepository <Empleado, Integer>, EmpleadoRepo {  
  
    List<Empleado> findByIdGreaterThanOrNombreLike (Integer pId, String contiene);  
}
```

5. Spring Data JPA

El método que acabamos de declarar no vamos a tener que implementarlo, sino que lo implementará JPA cuando arranquemos nuestra aplicación, por eso sólo tendremos que llamarlo desde el **service**, donde crearemos el método **listarConJPA** para tal fin:

```
public interface EmpleadoService {  
    public void registrar (String name);  
    public List<Empleado> listar();  
    public List<Empleado> listarFiltroNombre(String cad);  
     public List<Empleado> listarConJPA(Integer pID, String contiene);  
}
```



```
@Override  
public List<Empleado> listarConJPA(Integer pID, String contiene) {  
    return empleadoDAO.findByIdGreaterThanOrNombreLike(pID, contiene);  
}
```

5. Spring Data JPA

Tras ello sólo nos quedará llamarlo desde el **controller** y pintarlo en la página **html**:

```
@GetMapping("/listarEmpleados")
public String listarEmp(Model model) {
    model.addAttribute ("listaEmp", empleadoService.listar() );
    model.addAttribute ("listaEmpConE", empleadoService.listarFiltroNombre("e") );
    model.addAttribute ("listaJPA", empleadoService.listarConJPA( 2, "%o%" ) );
    return "listarDeEmpleados";
}
```

```
</table><br/>
Lista de empleados consultados con JPA (id>2 & contenga 'o' en el nombre):
<table>
    <th:block th:each="per : ${listaJPA}"> 
        <tr>
            <td th:text="${per.id}"></td>
            <td th:text="${per.nombre}"></td>
            <td th:text="${per.apellidos}"></td>
        </tr>
    </th:block>
</table>
</body>
</html>
```

5. Spring Data JPA

INICIO: Aplicación web APP.

MENÚ:

- [Saluda](#)
- [Salúdame](#)
- [Lista de empleados](#)



Lista de empleados:

- 1 Rocío De la O
- 2 Alberto Del Monte
- 3 Lucía Ricarti
- 4 Roberto Sánchez

Lista de empleados que contenga una 'e' en su nombre:

- 2 Alberto Del Monte
- 4 Roberto Sánchez

➔ Lista de empleados consultados con JPA ($\text{id} > 2$ & contenga 'o' en el nombre):

- 4 Roberto Sánchez



6

Incorporando servicios REST

6. Incorporando servicios REST

Ahora vamos a incorporar a nuestra aplicación **una capa de servicios REST**, en esta formación vamos a tratar este tema muy por encima ya que hay otra donde se ve a más detalle toda esta parte (**Conociendo Spring Boot. Crear un API REST**).

Lo primero que vamos hacer es crearnos **un nuevo controller** que se encargue de gestionar las peticiones que lleguen a **/rest/empleados**, que es dónde se implementarían los servicios REST asociados al dominio de empleados.

```
package com.everis.rest;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping ("/rest/empleados")
public class EmpleadosRestController {

}
```

6. Incorporando servicios REST

Ahora implementamos el método **listarEmpleados** que se va a encargar de devolver el listado de empleados llamando al servicio:

```
package com.everis.rest;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import com.everis.repository.entity.Empleado;
import com.everis.service.EmpleadoService;

@RestController
@RequestMapping ("/rest/empleados")
public class EmpleadosRestController {

    @Autowired
    EmpleadoService empleadoService;

    @GetMapping
    public List<Empleado> listarEmpleados() {
        return empleadoService.listar();
    }
}
```

6. Incorporando servicios REST

Tras ello sólo nos quedará la llamada al mismo:

<http://localhost/app/rest/empleados>

A screenshot of a web browser window. The address bar shows 'localhost/app/rest/empleados'. The page content displays a JSON array of employee objects. Each object has an 'id' (1 or 2), 'nombre' ('Rocío' or 'Alberto'), and 'apellidos' ('De la O' or 'Del Monte').

```
[{"id": 1, "nombre": "Rocío", "apellidos": "De la O"}, {"id": 2, "nombre": "Alberto", "apellidos": "Del Monte"}]
```

Nota: se está usando la extensión jsonView de Chrome para verlo así



7

Manejo de caché (@Cacheable)

7. Manejo de caché

Habitualmente usamos Spring para crear Servicios y Repositorios que definen **la parte del Modelo de nuestra aplicación**. En bastantes casos nos encontramos con situaciones **en las que un Servicio siempre devuelve la misma información**, por ejemplo tablas de parámetros. En las que no tiene sentido estar continuamente realizando una consulta a la base de datos para devolver la misma información. Para estas situaciones Spring incorpora soluciones de Cache que **permiten almacenar en memoria datos devueltos por un método**.

En este capítulo vamos a ver como utilizar el manejo de la caché de Spring para mejorar el rendimiento de nuestras aplicaciones.

Para empezar incluiremos la siguiente dependencia en el **pom.xml**:

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

7. Manejo de caché

Pero antes de utilizar la caché vamos a meter un retardo de segundo y medio (1.500 milisegundos) al servicio REST que nos devuelve el listado de empleados:

```
package com.everis.rest;

import java.util.List;

@RestController
@RequestMapping ("/rest/empleados")
public class EmpleadosRestController {

    @Autowired
    EmpleadoService empleadoService;

    @GetMapping
    public List<Empleado> listarEmpleados() {
        try {
            Thread.sleep(1500);
        } catch (InterruptedException e) {}
        return empleadoService.listar();
    }
}
```

7. Manejo de caché

Si hacemos varias llamadas al servicio REST vemos que todas las llamadas tardan más de 1,5 seg:

The screenshot shows two API requests in Postman. Both requests are GET methods to `http://localhost/app/rest/empleados`. The first request has a status of 200 OK, time of 1899ms, and size of 317 B. The second request also has a status of 200 OK, time of 1531ms, and size of 317 B. In both requests, there is a large green arrow pointing downwards from the 'Description' column, likely indicating a cached response.

GET `http://localhost/app/rest/empleados` Send

Params Authorization Headers (7) Body Pre-request Script Tests Settings

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

Status: 200 OK Time: 1899ms Size: 317 B

Body Cookies Headers (5) Test Results

Pretty Raw Preview \ GET `http://localhost/app/rest/empleados`

1 [Params Authorization Headers (7) Body Pre-request Script Tests Settings

2 { Query Params

3 "id": 1, KEY

4 "nombre": "Ro VALUE

5 "apellidos": Description

6 }, Key

7 { Value Description

8 "id": 2, Body

9 "nombre": "Al Cookies Headers (5) Test Results

10 "apellidos": Status: 200 OK Time: 1531ms Size:

11 }

7. Manejo de caché

Tras ello habilitaremos la caché en nuestra aplicación con la siguiente anotación:

```
package com.everis;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cache.annotation.EnableCaching;

@EnableCaching
@SpringBootApplication
public class DemoWebApplication {

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

7. Manejo de caché

Y a continuación creamos una ‘caché’ asociada al recurso de ‘empleados’ que llamaremos por el mismo nombre:

```
package com.everis.rest;

import java.util.List;

@RestController
@RequestMapping ("/rest/empleados")
public class EmpleadosRestController {

    @Autowired
    EmpleadoService empleadoService;

    @GetMapping
    @Cacheable(value="empleados")
    public List<Empleado> listarEmpleados() {
        try {
            Thread.sleep(1500);
        } catch (InterruptedException e) {}
        return empleadoService.listar();
    }
}
```



7. Manejo de caché

Y volvemos a hacer dos llamadas al servicio REST. La primera capturará la información y la dejará cacheada, y así la segunda no tendrá que hacer acceso a datos y sólo leerá de la caché:

The screenshot shows two API requests in the Postman application.

Request 1 (Top):

- Method: GET
- URL: `http://localhost/app/rest/empleados`
- Status: 200 OK, Time: 1899ms, Size: 317 B
- Body tab is selected.
- Query Params table:

KEY	VALUE	DESCRIPTION
Key	Value	Description

Request 2 (Bottom):

- Method: GET
- URL: `http://localhost/app/rest/empleados`
- Status: 200 OK, Time: 9ms, Size: 266 B
- Body tab is selected.
- Query Params table:

KEY	VALUE	DESCRIPTION
Key	Value	Description

7. Manejo de caché

A partir de este momento siempre que se acceda a este servicio, la información estará cacheada. Pero, **¿y si cambian los datos?**

Vamos a crear un nuevo servicio REST que inserte un empleado, y le diremos que cuando se ejecute debe actualizarse la caché:

```
package com.everis.rest;

import java.util.List;

@RestController
@RequestMapping ("/rest/empleados")
public class EmpleadosRestController {

    @Autowired
    EmpleadoService empleadoService;

    @PostMapping
    @CacheEvict(value="empleados", allEntries = true)
    public void insertarEmpleado(@RequestBody Empleado emp) {
        empleadoService.inserta(emp);
    }
}
```

7. Manejo de caché

A continuación haremos una llamada (desde Postman) al servicio REST para que nos cachee los empleados, y posteriormente otra a nuestro nuevo servicio insertando un nuevo empleado:

The screenshot shows the Postman application interface. At the top, it displays a 'POST' method and the URL 'localhost/app/rest/empleados'. On the right, there is a blue 'Send' button. Below the URL, there are tabs for 'Params', 'Authorization', 'Headers (9)', 'Body' (which is highlighted with an orange underline), 'Pre-request Script', 'Tests', and 'Settings'. Under the 'Body' tab, there are several options: 'none', 'form-data', 'x-www-form-urlencoded', 'raw' (which is selected and highlighted with an orange circle), 'binary', 'GraphQL', and 'JSON' (with a dropdown arrow). The 'JSON' option is currently selected. In the main body area, there is a code editor containing the following JSON payload:

```
1 {  
2   "id": "3",  
3   "nombre": "Antonia",  
4   "apellidos": "Del Arte"  
5 }
```

7. Manejo de caché

Y si ahora volvemos a ejecutar dos peticiones seguidas al servicio REST que nos lista los empleados:

The screenshot shows the Postman interface with two requests to the same endpoint. The first request has a 'Description' column in its query parameters table, indicated by a green arrow pointing down at the 'Description' header. The second request shows a JSON response with three employee objects, indicated by a green arrow pointing up at the response body.

Request 1 (Top):

KEY	VALUE	DESCRIPTION
Key	Value	Description

Request 2 (Bottom):

KEY	VALUE	DESCRIPTION
Key	Value	Description

Response Body (Pretty):

```
1 [ { 2   "id": 1, 3     "nombre": "Rocío", 4     "apellidos": "De la O" 5 }, 6   { 7     "id": 2, 8     "nombre": "Alberto", 9     "apellidos": "Del Monte" 10 }, 11   { 12     "id": 3, 13     "nombre": "Antonia", 14     "apellidos": "Del Arte" 15 } ]
```

7. Manejo de caché

Esa misma caché podríamos utilizarla para el controlador asociado a la web que muestra el listado de empleados:

```
@GetMapping("/listarEmpleados")
@Cacheable(value="empleados") ←
public String listarEmp(Model model) {
    model.addAttribute ("listaEmp", empleadoService.listar() );
    model.addAttribute ("listaEmpConE", empleadoService.listarFiltroNombre("e") );
    return "listarDeEmpleados";
}
```



Actividad



- Crea la entidad.
- En el fichero 'data.sql' añádele dos registros.
- Implementa una nueva opción de tu web en 'index.html' que al pulsarla muestre el listado de asignaturas (tendrás que crear el repositorioJPA, el servicio y el método asociado al controller).
- Implementa un servicio REST (de tipo 'GET') que devuelva el listado de asignaturas.
- Crea una 'caché' asociada a dicho listado.

Actividad:

Nuestra aplicación web va a tener una nueva entidad 'asignaturas'.

Una asignatura tendrá los siguientes campos:

- Id: number (primary key)
- Nombre (string tamaño 20)
- Descripcion (string tamaño 50)
- Curso: numérico



8

Securizar con Spring Security

8. Segurizar con Spring Security

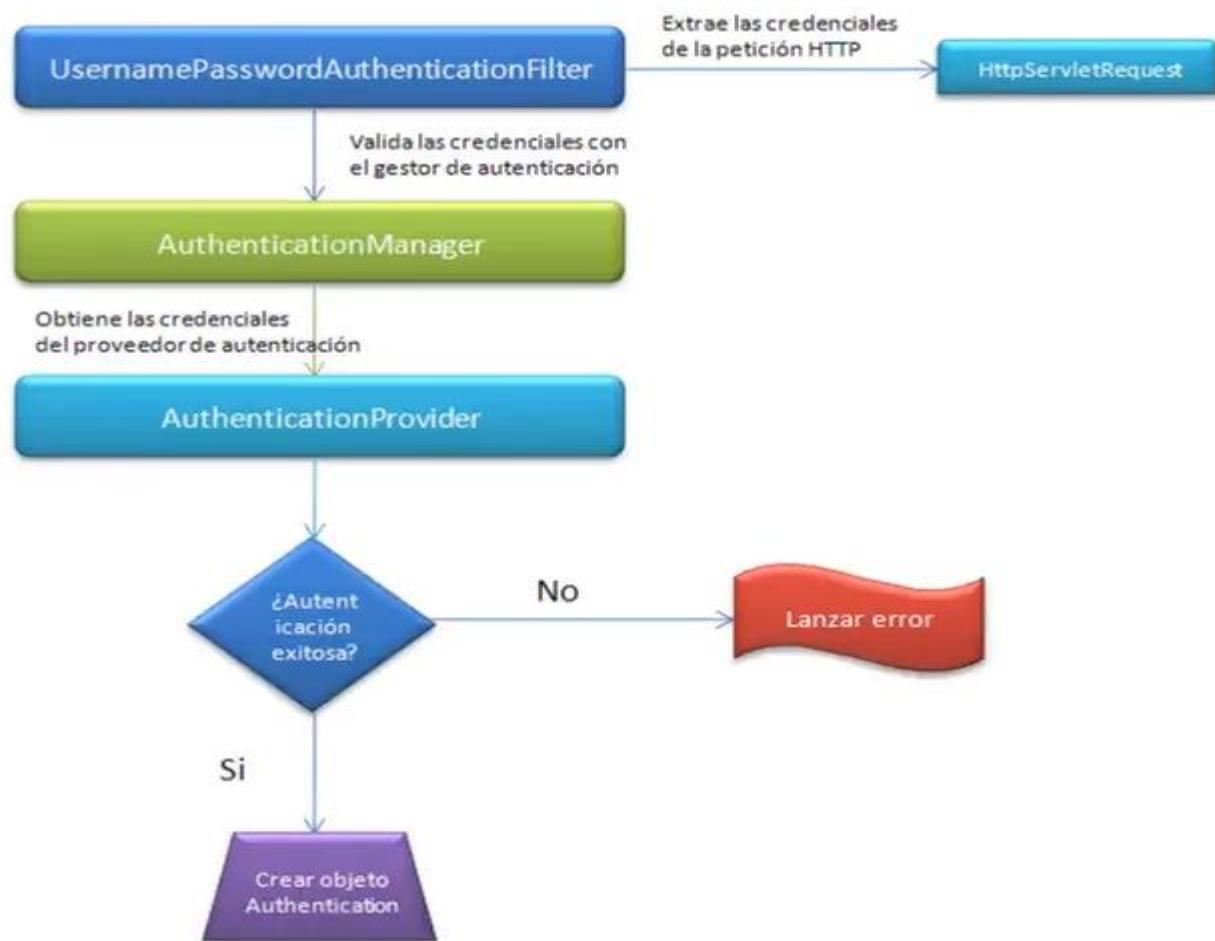
Spring Security es un módulo de Spring que nos permite manejar autenticaciones.

¿Cómo funciona?

Cuando hacemos una petición Spring Security extrae la información de autorización de la petición, que suele viajar en el header.

De ahí extrae las credenciales de la autorización, que las gestionará el **AuthenticationManager**.

Y si está correcto nos crea un '**AuthenticationProvider**' (sino lanzará un error).



8. Segurizar con Spring Security

Toda esta información de la petición vamos a poder verla de forma sencilla:

The screenshot shows the Network tab of the Chrome DevTools Network panel. The URL is `localhost/app`. The request has a duration of 10 ms. The Headers section shows the following response headers:

- Connection:** keep-alive
- Content-Language:** es-ES
- Content-Type:** text/html; charset=UTF-8
- Date:** Sat, 04 Apr 2020 12:27:18 GMT
- Keep-Alive:** timeout=60
- Transfer-Encoding:** chunked

The Request Headers section shows the following accept header:

- Accept:** text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/ap
on/signed-exchange;v=b3;q=0.9

At the bottom left, it says 1 requests | 581 B translat.

8. Segurizar con Spring Security

Toda esta información de la petición vamos a poder verla de forma sencilla:

The screenshot shows the Postman application interface. At the top, there is a search bar with the URL `http://localhost/app/rest/empleados`. Below the search bar, the method is set to `GET` and there are buttons for `Send` and `Save`. The `Headers` tab is currently selected, indicated by an orange underline. Other tabs include `Params`, `Authorization`, `Body`, `Pre-request Script`, `Tests`, and `Settings`. To the right of the tabs are buttons for `Cookies` and `Code`. Below the tabs, there is a table titled "Headers" with columns `KEY`, `VALUE`, and `DESCRIPTION`. A note indicates there are 7 hidden headers. The table shows the following data:

KEY	VALUE	DESCRIPTION
Key	Value	Description

At the bottom of the interface, there are tabs for `Body`, `Cookies`, `Headers (5)`, and `Test Results`. The `Headers (5)` tab is selected, indicated by an orange underline. To the right of these tabs, status information is displayed: `Status: 200 OK`, `Time: 37ms`, `Size: 317 B`, and a `Save Response` button.

`Content-Type`: application/json

`Transfer-Encoding`: chunked

`Date`: Sat, 04 Apr 2020 12:33:28 GMT

`Keep-Alive`: timeout=60

`Connection`: keep-alive

8. Segurizar con Spring Security

El primer paso será añadir la dependencia en el pom.xml para poder utilizar **Spring Security**:

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

Y tras ello, **reiniciaremos** nuestra aplicación web:

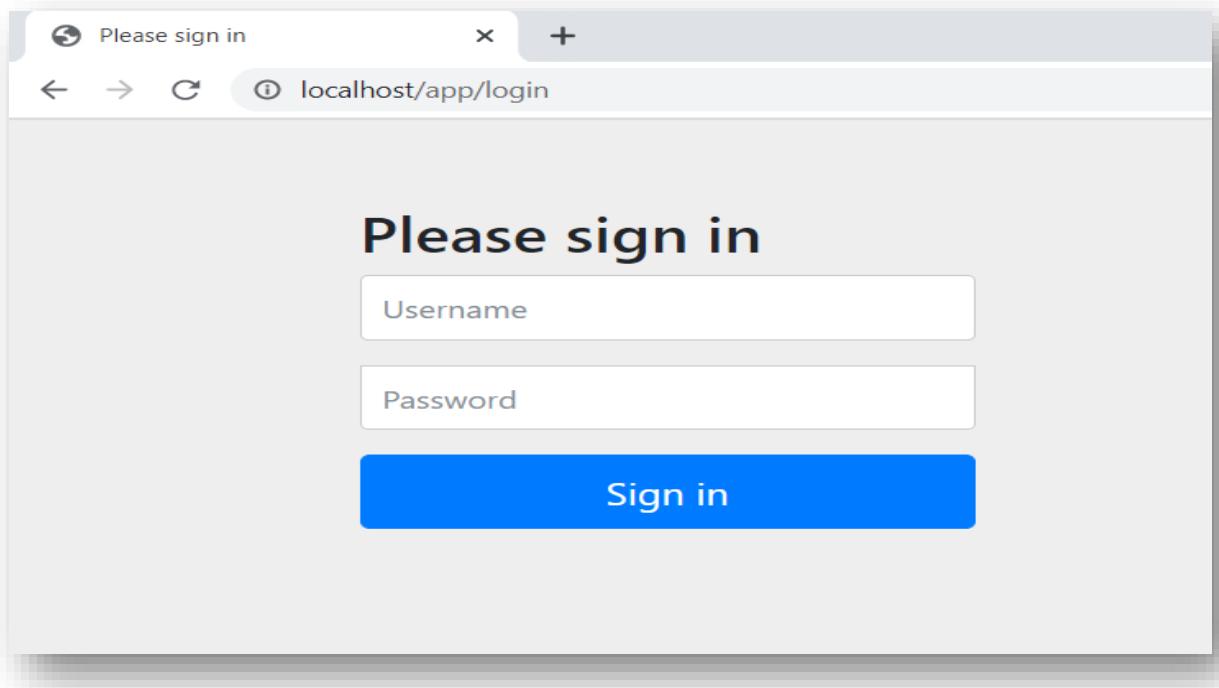
```
[ restartedMain] o.s.s.concurrent.ThreadPoolExecutor : Initializing ExecutorService 'applicationTaskExecutor'
[ restartedMain] o.s.b.a.w.s.WelcomePageHandlerMapping : Adding welcome page template: index
[ restartedMain] .s.s.UserDetailsServiceAutoConfiguration :

Using generated security password: b912ec67-907f-4023-9efe-95daef71bd11

[ restartedMain] o.s.s.web.DefaultSecurityFilterChain : Creating filter chain: any request, [org.springframework.security.web.
[ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 80 (http) with context path '/app'
[ restartedMain] com.everis.DemoWebApplication : Started DemoWebApplication in 4.322 seconds (JVM running for 5.255)
```

8. Segurizar con Spring Security

Si volvemos a acceder a nuestra web veremos que ahora nos pide autenticación:

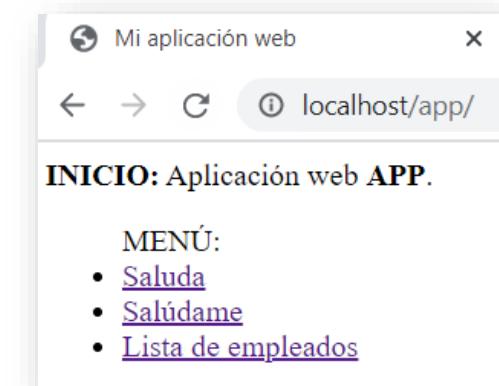


Y veremos que sólo nos dejará autenticarnos con:

Username: **user**

Password: **(la contraseña que os ha generado en cada caso)**

b912ec67-907f-4023-9efe-95daef71bd11 (en el mío)



8. Segurizar con Spring Security

Nosotros vamos a implementar toda la lógica necesaria para que nuestra aplicación tenga una validación de usuarios por BBDD.

Para ello vamos a necesitar en una primera instancia:

- Entity de Usuario
- DAO/Repo de Usuario
- Servicio de Usuario

Pero a su vez cada Usuario va a tener un rol ('ADMIN', 'ROL2', 'ROL3'...), con lo que necesitaremos también las entidad rol.

- Entity de Rol

Pero antes de empezar vamos a desactivar la seguridad porque sino no podremos acceder a la h2-console (más adelante habilitaremos la seguridad y configuraremos para que se pueda acceder a la misma):



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

8. Segurizar con Spring Security

Como primer paso vamos a implementar la entidad **Rol**, que tendrá los atributos **id** y **rol**. El primero será un identificador único y el segundo el rol en sí ('ADMIN', 'ROL2'...).

Para que sea un **rol válido para Spring Security** vamos a implementar la interfaz **GrantedAuthority**:

```
package com.everis.repository.entity;

@Entity
@Table
public class Rol /*implements GrantedAuthority*/ {
    @Id
    @Column
    private Integer id;

    @Column
    private String rol;

    //Override
    public String getAuthority() {
        return ("ROLE_"+this.rol).toUpperCase();
    }
}
```

Nota: faltan incluir los getters/setters

8. Segurizar con Spring Security

A continuación crearemos la entidad de **Usuario**. Tendrá como atributos un **username**, un **nombreYapellidos**, un **password** y un **rol**. Para que sea un **usuario válido para Spring Security** vamos a implementar la interfaz **UserDetails**:

```
package com.everis.repository.entity;

@Entity
@Table
public class Usuario /*implements UserDetails*/ {
    @Id
    @Column
    private String username;

    @Column(name="nombre", nullable = false, length=50)
    private String nombreYapellidos;

    @Column(nullable = false)
    private String password;

    @OneToOne(optional = false)
    private Rol rol;

}
```

Nota: faltan incluir los getters/setters

8. Segurizar con Spring Security

Una vez que ya tenemos las entidades, pasamos al **REPO/DAO** de **Usuario** y de **Rol**, aunque este último de primeras no lo vamos a usar:

```
package com.everis.repository;

import org.springframework.data.jpa.repository.JpaRepository;

import com.everis.repository.entity.Usuario;

public interface UsuarioRepoJPA extends JpaRepository<Usuario, String> {  
}  
  
package com.everis.repository;

import org.springframework.data.jpa.repository.JpaRepository;

import com.everis.repository.entity.Rol;

public interface RolRepoJPA extends JpaRepository<Rol, Integer>{  
}
```

8. Segurizar con Spring Security

Tras implementar toda la parte de datos, nos queda añadir la carga inicial de datos para poder tener un par de usuarios y roles. Así en el fichero `src/main/resources/data.sql` añadiremos:

```
insert into rol (id, rol)
select 1, 'ADMIN' from dual where not exists (select 1 from rol where id = 1);

insert into rol (id, rol)
select 2, 'GESTOR' from dual where not exists (select 1 from rol where id = 2);

/* pass = 1111 */
insert into usuario (username, nombre, password, rol_id)
select 'user1', 'Pon aquí tu nombre',
'$2a$10$5xOe75pbLcAjp0TbVWaluunrSshgYdH82YNwGd.b0Os4hAWbIEkry', 1 from dual where not
exists (select 1 from usuario where username = 'user1');

insert into usuario (username, nombre, password, rol_id)
select 'user2', 'Empleado de everis',
'$2a$10$5xOe75pbLcAjp0TbVWaluunrSshgYdH82YNwGd.b0Os4hAWbIEkry', 2 from dual where not
exists (select 1 from usuario where username = 'user2');
```

8. Segurizar con Spring Security

El servicio de **Usuario** contendrá un método para **listar** y otro para **buscar por username**:

```
package com.everis.service;

import java.util.List;

import com.everis.repository.entity.Usuario;

public interface UsuarioService {
    public List<Usuario> listar();
    Usuario buscarPorUsername(String username);
}
```

```
package com.everis.service.impl;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

import com.everis.repository.UsuarioRepoJPA;
import com.everis.repository.entity.Usuario;
import com.everis.service.UsuarioService;
```

En la implementación de dicho servicio, se implementará también la interfaz **UserDetailsService** para facilitar el trabajo con **Spring Security**:

```
@Service
public class UsuarioServiceImpl implements UsuarioService, UserDetailsService {

    @Autowired
    UsuarioRepoJPA usuarioDAO;

    @Override
    public List<Usuario> listar() {
        return usuarioDAO.findAll();
    }

    @Override
    public Usuario buscarPorUsername(String username) {
        Usuario u = usuarioDAO.findById(username).get();
        return usuarioDAO.findById(username).get();
    }

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        return buscarPorUsername(username);
    }
}
```

8. Segurizar con Spring Security

Nota:

Por seguridad no vamos a almacenar las contraseñas directamente, sino que las encriptaremos antes. Para ver qué valor tendría una contraseña encriptada podría usarse algo como lo siguiente:

```
@Autowired  
private BCryptPasswordEncoder passwordEncoder() {  
    return new BCryptPasswordEncoder();  
}
```



BCryptPasswordEncoder va a ser el encargado de encriptar nuestras contraseñas y así no almacenarlas en formato original y protegernos en caso de robo de las mismas.

```
private static Logger LOG = org.slf4j.LoggerFactory.getLogger(DemowebApplication.class);
```

```
@Override  
public Usuario buscarPorUsername(String username) {  
    Usuario u = usuarioDAO.findById(username).get();  
    LOG.info("UsuarioServiceImpl - " + u.getUsername() + ":" + u.getPassword() + ":" + passwordEncoder().encode( u.getPassword() ));  
    return usuarioDAO.findById(username).get();  
}
```



8. Segurizar con Spring Security

Tras ello añadimos en el índice un acceso al h2-console para facilitarnos el comprobar que se insertan bien ambos roles y usuarios:

```
<li> <a href="/app/h2-console/">Consola BBDD H2</a> </li>
```



SELECT * FROM USUARIO;

USERNAME	NOMBRE	PASSWORD	ROL_ID
user1	Julian Hernandez	\$2a\$10\$5xOe75pbLcAjp0TbVWaluunrSshgYdH82YNwGd.b0Os4hAWbIEkry	1
user2	Empleado de everis	\$2a\$10\$5xOe75pbLcAjp0TbVWaluunrSshgYdH82YNwGd.b0Os4hAWbIEkry	2

(2 filas, 2 ms)



SELECT * FROM ROL;

ID	ROL
1	ADMIN
2	GESTOR

(2 filas, 5 ms)

8. Segurizar con Spring Security

Una vez que hemos implementado la parte de BBDD que necesitábamos, vamos ahora a implementar las configuraciones y lógica necesaria para darle uso a estos usuarios y roles. Como primer paso volvemos a activar la seguridad:

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

Tras ello vamos a crear la clase que se va a encargar de nuestra configuración con **Spring Security**:

```
package com.everis.configuration;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

@Configuration
@EnableWebSecurity
public class WebSecurityConfiguration extends WebSecurityConfigurerAdapter {
```

8. Segurizar con Spring Security

En dicha clase lo primero que vamos a implementar es cómo va a recoger la autenticación para nuestras peticiones web. Y basándonos en el **BCryptPasswordEncoder** vamos a encriptar la contraseña que meta el usuario en pantalla para poder compararla con la de nuestra BBDD:

```
@Autowired  
private UserDetailsService serviceUsuario;  
  
public BCryptPasswordEncoder passwordEncoder() {  
    return new BCryptPasswordEncoder();  
}  
  
@Override  
protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
    auth.userDetailsService( serviceUsuario ).passwordEncoder(passwordEncoder());  
}
```

8. Segurizar con Spring Security

Tras ello en las tres siguientes transparencias vamos a completar las clases ya creadas utilizando algunas clases de la librería de **Spring Security**:

```
package com.everis.repository.entity;  
  
@Entity  
@Table  
public class Rol implements GrantedAuthority {  
    @Id  
    @Column  
    private Integer id;  
  
    @Column  
    private String rol;  
  
    @Override  
    public String getAuthority() {  
        return ("ROLE_"+this.rol).toUpperCase();  
    }  
}
```



Nota: faltan incluir los getters/setters

8. Segurizar con Spring Security

A continuación crearemos la entidad de **Usuario**. Tendrá como atributos un **username**, un **nombreYapellidos**, un **password** y un **rol**. Para que sea un **usuario válido para Spring Security** vamos a implementar la interfaz **UserDetails**:

```
package com.everis.repository.entity;

@Entity
@Table
public class Usuario implements UserDetails {
    @Id
    @Column
    private String username;
    @Column(name="nombre", nullable = false, length=50)
    private String nombreYapellidos;
    @Column(nullable = false)
    private String password;
    @OneToOne(optional = false)
    private Rol rol;
    @Override
    public String getAuthority() {
        return ("ROLE_"+this.rol).toUpperCase();
    }
}
```

```
...
@Override
public Collection<? extends GrantedAuthority> getAuthorities() {
    return Arrays.asList(rol);
}

@Override
public boolean isAccountNonExpired() {
    return true;
}

@Override
public boolean isAccountNonLocked() {
    return true;
}

@Override
public boolean isCredentialsNonExpired() {
    return true;
}

@Override
public boolean isEnabled() {
    return true;
}
}
```

Nota: faltan incluir los getters/setters

8. Segurizar con Spring Security

Como primer paso vamos a implementar la entidad **Rol**, que tendrá los atributos **id** y **rol**. El primero será un identificador único y el segundo el rol en sí ('ADMIN', 'ROL2'...).

Para que sea un **rol válido para Spring Security** vamos a implementar la interfaz **GrantedAuthority**:

```
package com.everis.repository.entity;

@Entity
@Table
public class Rol implements GrantedAuthority {
    @Id
    @Column
    private Integer id;

    @Column
    private String rol;

    @Override
    public String getAuthority() {
        return ("ROLE_"+this.rol).toUpperCase();
    }
}
```

Nota: faltan incluir los getters/setters

8. Segurizar con Spring Security

A continuación configuraremos el acceso a los distintos recursos:

```
String[] resources = new String[] { "/include/**", "/js/**", "/css/**"};  
  
 @Override  
protected void configure(HttpSecurity http) throws Exception {  
    http.authorizeRequests()  
        .antMatchers(resources).permitAll() // Se permite el acceso a los 'resources'  
        .and().authorizeRequests().antMatchers("/console/**").permitAll() // Permitir acceso a consola de H2  
        .and().authorizeRequests().anyRequest().authenticated() // El resto peticiones debe estar autenticadas  
        .and().httpBasic() // Permitir autenticación básica para los servicios rest  
        .and().formLogin() // Página de login de mi aplicación  
        .failureUrl("/login?error=true") // Si hay fallo dónde me direcciona  
        .defaultSuccessUrl("/") // Si todo va correcto me manda aquí  
        .and().logout().logoutSuccessUrl("/login?logout=true").permitAll()  
        .and().rememberMe().key("uniqueAndSecret"); // Para recordar autenticación (!)  
  
    http.csrf().disable();  
    http.headers().frameOptions().disable();  
}
```

8. Segurizar con Spring Security

Una vez realizado este paso en el **DemoController** vamos a pasar la información del usuario logueado a la página **index.html** (y logueándonos con user1 ó user2 pass '1111'):

```
@GetMapping("/")
public String index(Model model) {
    Usuario u = (Usuario) SecurityContextHolder.getContext().getAuthentication().getPrincipal();
    model.addAttribute ("usuario", u);
    return "index";
}
```

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Mi aplicación web</title>
</head>
<body>
    <h5 th:inline="text">Hola '[[${usuario.nombreYapellidos}]]' ([[${usuario.username}]])<br/>
    tu rol es [[${usuario.rol.rol}]]. </h5>
    <b>INICIO:</b> Aplicación web <b>APP</b>.
    <ul> MENÚ:

```

The screenshot shows a browser window titled 'Mi aplicación web' with the URL 'localhost/app/'. The page content is: 'Hola 'Julian Hernandez' (user1), tu rol es ADMIN.' Below this, there is a menu section with the heading 'INICIO: Aplicación web APP.' followed by a list of links: 'MENÚ:' with items 'Saluda', 'Salúdame', 'Lista de empleados', and 'Consola BBDD H2'.

```
MENÚ:
• Saluda
• Salúdame
• Lista de empleados
• Consola BBDD H2
```

8. Segurizar con Spring Security

Como siguiente paso vamos a restringir el acceso al listado de empleados para que sólo sea accesible para el rol 'ADMIN', para ello vamos a añadir dos anotaciones, tanto en el **WebSecurityConfiguration** como en nuestro **DemoController**:

```
@Configuration  
@EnableWebSecurity  
@EnableGlobalMethodSecurity(prePostEnabled = true)  
public class WebSecurityConfiguration extends WebSecurityConfigurerAdapter {
```

```
@PreAuthorize("hasRole('ROLE_ADMIN')")  
@GetMapping("/listarEmpleados")  
@Cacheable(value="empleados")  
public String listarEmp(Model model) {  
    model.addAttribute ("listaEmp", empleadoService.listar() );  
    model.addAttribute ("listaEmpConE", empleadoService.listarFiltroNombre("e") );  
    return "listarDeEmpleados";  
}
```

8. Segurizar con Spring Security

Ahora vamos a probar acceder con los distintos usuarios (user1 y user2):

The diagram illustrates the behavior of Spring Security for two users: user1 (ADMIN) and user2 (GESTOR). It shows two browser tabs and their respective content pages.

- User1 (ADMIN):** The top tab shows the URL `localhost/app/listarEmpleados`. The content page displays:
 - Lista de empleados:**
1 Rocío De la O
2 Alberto Del Monte
 - Lista de empleados que contenga una 'e' en su nombre:**
2 Alberto Del Monte
- User2 (GESTOR):** The bottom tab shows the URL `localhost/app/listarEmpleados`. The content page displays:

ERROR: El recurso al que está intentando acceder no existe o no tiene permisos.

8. Segurizar con Spring Security

Para terminar vamos a modificar la página **index.html** para que no se vea el link asociado al listado de empleados si no se tienen permisos:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"> ←
<head>
    <meta charset="UTF-8">
    <title>Mi aplicación web</title>
</head>
<body>
    <h5 th:inline="text">Hola '[[${usuario.nombreYapellidos}]]' ([[${usuario.username}]]) ,
    tu rol es [[${usuario.rol.rol}]]. </h5>

    <b>INICIO:</b> Aplicación web <b>APP</b>.
    <ul> MENÚ:
        <li> <a href="/app/saludo">Saluda</a> </li>
        <li> <a href="/app/saludo?name=Julian">Salúdame</a> </li>
    <li th:if="${usuario.rol.rol == 'ADMIN'}"> <a href="/app/listarEmpleados">Lista de empleados</a></li> →
        <li> <a href="/app/h2-console/">Consola BBDD H2</a> </li>
    </ul>

    </form>
</body>
</html>
```

8. Segurizar con Spring Security

Así según con qué usuario accedamos ahora se verá o no la opción de menú de listar empleados:

Mi aplicación web

localhost/app/

Hola 'Julian Hernandez' (user1), tu rol es ADMIN.

INICIO: Aplicación web APP.

MENÚ:

- [Saluda](#)
- [Salúdame](#)
- [Lista de empleados](#)
- [Consola BBDD H2](#)

A large green arrow points to the "Lista de empleados" menu item.

Mi aplicación web

localhost/app/

Hola 'Empleado de everis' (user2), tu rol es GESTOR.

INICIO: Aplicación web APP.

MENÚ:

- [Saluda](#)
- [Salúdame](#)
- [Consola BBDD H2](#)



9

Repaso de conceptos sobre servicios REST

9. Repaso de conceptos sobre servicios REST

Fundamentos servicios REST

REST: REpresentational State Transfer. Son un conjunto de restricciones que, aplicadas al diseño de un sistema, crean un estilo arquitectónico caracterizado por:

- Debe ser un sistema cliente-servidor
- Tiene que ser sin estado
- Tiene que soportar *cachés*
- Tiene que ser un sistema uniformemente accesible (a través de URIs)
- Tiene que ser un sistema por capas (escalabilidad)

RESTful web service o **RESTful api** son aplicaciones basadas en REST.

Estas restricciones no dictan qué tipo de tecnología utilizar.

Podemos utilizar las infraestructuras de red existentes.

- Un ejemplo de sistema **RESTful**: la web estática

PARIS *Literature* COME ENJOY THE ROMANCE, POETRY AND FINE ART OF PARIS

Y G Go Store

The world is your grapevine. **VISA**

Click here for great wines from all over the world.

The world is your grapevine. Click here for great wines!

[JOIN](#) | [CHAT](#) | [FORUM](#) | [AUD](#)

Paris: Literature

[Yahoo!Literature](#)

Paris 9261 - IRElingus- all about Ireland and its people.

Paris 3869 - Denise has some words and memories she wants to express.

Paris 7223 - Poetry about love, relationships, culture and society.

Paris 9007 - Swedish schoolgirl expresses her views, shares her poetry, and comments on life in Sweden.

Paris 5638 - Read Spanish commentaries about the world of cyberspace at Cartas A Mi Gato.

Paris/LeftBank 4988 - Read Brian's original fiction and prose, and stay updated on his weekly serial, "The Mowing of The Lawn".

Paris 8073 - The Upstairs Room has poetry, Robert Smith quotes, and an ever-expanding Danish poetry library.

Paris/Metro 8839 - Read about all kinds of romances from fairies to hosts!

Paris 3963 - Gloriana's Court is for a time when the unicorn romped and the gryphon guarded the cathedral.

Paris 9376 - The Mendicant shares her wares, poetry and prose

[Books about Literature](#)

from [amazon.com](#)

- [New Titles](#)
- [Bestsellers](#)

For the best computer deals on the net

SURPLUS DIRECT



[click here](#)

Apply for your Platinum Card

4.9% ADD

MARCA
DIGITAL

miniMARCA

NUEVO

- Base de datos de la [LIGA DE FUTBOL 98-99](#)

En juego...

FUTBOL

LIGA 98/99

- Primera: [13ª J.](#)
 - Segunda: [15ª J.](#)
 - [Quiniela](#)
- EUROPA**
- [1/8 Copa de la UEFA](#)
 - [Liga de Campeones](#)
 - [1/8 Recopa](#)

BALONCESTO

- [Liga ACB.](#)

Jornada 14

BALONMANO

- [Liga Asobal.](#)

Jornada 16

NOTICIAS DEL DIA

LIGA FANTASTICA

AJEDREZ

La página de...

Tu deporte

- [Presentación](#)
- [Carreras](#)
- [Actividades](#)
- [Eventos](#)
- [Instalaciones Deportivas](#)
- [FIFA Masters Regatta](#)
- [Noticias](#)
- [Voluntariado](#)
- [Album de Fotos](#)
- [Cartelería](#)
- [Enlaces](#)
- [andalucia.org](#)



GASPART DICE QUE VAN GAAL NO ES CUESTIONADO

Más crédito

El [Barcelona](#) ha decidido, pese a la derrota del sábado ante el Deportivo, ampliar el crédito a su entrenador, Louis Van Gaal. Según la directiva azulgrana, el equipo mejoró en Riazor.

[Liga 98-99. Jornada 13:](#) El Mallorca sigue tercera jornada consecutiva

• Haga [click aquí para ver todas las](#)

Especial

► [Liga 98-99](#)

Toda la información sobre las jornadas de Liga, todas las jornadas, partidos y jugadores.

► [Copa Intercontinental](#)

► [MARCA Leyenda](#)

► [Mundial de Francia '98](#)

La Empresa Pública Deporte Andaluz, nace el 26 de Febrero de 1997.

El Consejo de Gobierno de la Junta de Andalucía acordó constituir una Sociedad mercantil, de las previstas en el artículo 6º.1.a) de la Ley General de la Hacienda Pública de la Comunidad Autónoma de Andalucía, que adoptará la forma de sociedad anónima, denominándose "Empresa Pública de Deporte Andaluz, S.A.", en virtud Decreto 496/1996, de 26 de Noviembre, por el que se modifica determinados aspectos del Reglamento General de la Empresa Pública de Gestión de Programas Culturales aprobado por el Decreto 40/1993, de 20 de Abril, y se autoriza a la constitución de la Empresa de la Junta de Andalucía, Empresa Pública Deporte Andaluz, S.A.

CONSEJO DE ADMINISTRACIÓN DE LA EMPRESA PÚBLICA DEPORTE ANDALUZ, S.A.

• **PRESIDENTE:**

Excmo. Sr. D. José Núñez Castaño - Consejero de Turismo y Deporte.

• **VICEPRESIDENTE:**

Ilmo. Sr. D. Javier Sánchez-Palencia Dabán - Secretario General para el Deporte.

• **VOCAL:**

Ilmo. Sr. D. Baltasar Quintero Almendro - Director General de Actividades y Promoción Deportiva - Vocal y Consejero Delegado.

• **VOCAL:**

Ilmo. Sr. D. Enrique Naz Pajares Director de Infraestructura y Tecnología Deportiva.

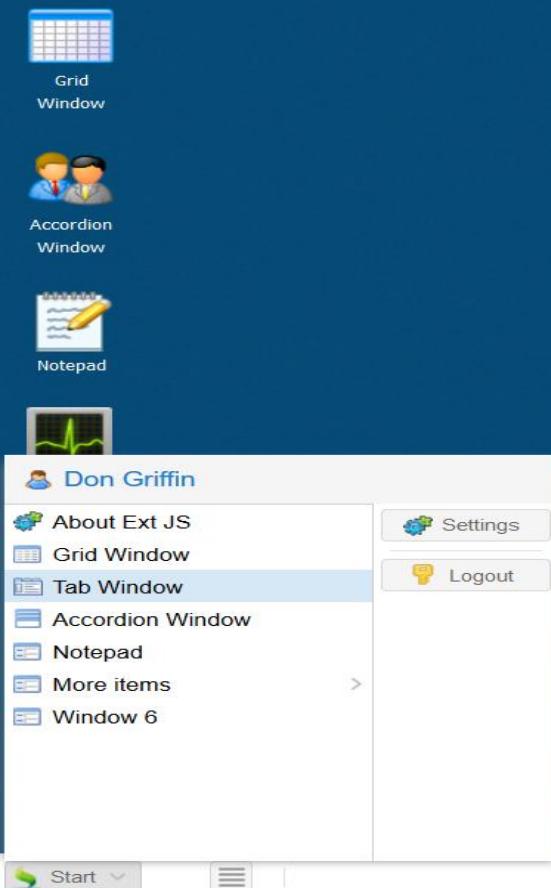
• **SECRETARIO:**

Sr. D. Francisco Sanz Lucena.

9. Repaso de conceptos sobre servicios REST

Gracias a los servicios REST, surgen las aplicaciones RIA (Rich Internet application)

<https://examples.sencha.com/extjs/5.1.0/examples/desktop/index.html>



The screenshot shows a desktop application interface. On the left, there is a vertical sidebar with four items: 'Grid Window' (grid icon), 'Accordion Window' (two people icon), 'Notepad' (notepad icon), and a 'More items' section with 'Window 6' (grid icon). To the right of the sidebar is a main area with a large green and yellow swoosh logo. At the bottom left is a 'Start' button.

- Grid Window
- Accordion Window
- Notepad
- Don Griffin
 - About Ext JS
 - Grid Window
 - Tab Window
 - Accordion Window
 - Notepad
 - More items >
 - Window 6

Start

9. Repaso de conceptos sobre servicios REST

Recursos

- Un recurso es “cualquier cosa” que pueda ser accedido y transferido a través de la red:
 - Es una correspondencia lógica y temporal con un concepto del dominio del problema
 - Ejemplos: una noticia de un periódico, la temperatura de Murcia, un valor de IVA almacenado en una BBDD...
- Cada uno de los recursos puede ser accedido directamente, y de forma independiente, pero diferentes peticiones podrían “apuntar” al mismo dato.
- La representación de un recurso depende del tipo solicitado por el cliente (tipo *MIME*)

9. Repaso de conceptos sobre servicios REST

Representación

- Lo que se intercambia entre los consumidores (clientes) y los productores de servicios son representaciones de los recursos.
- Una representación muestra el estado de un dato real almacenado en algún dispositivo de almacenamiento en el momento de la petición.
- Durante el ciclo de vida del servicio web, puede haber varios clientes solicitando recursos. Clientes diferentes pueden solicitar diferentes representaciones del mismo recurso.
- El “lenguaje” de intercambio de información (representación) con el servicio queda a elección del desarrollador.
- Ejemplos de representaciones comunes son:

Formato	Tipo MIME
Texto plano	text/plain
HTML	text/html
XML	application/xml
JSON	application/json

9. Repaso de conceptos sobre servicios REST

URI

- Una URI (Uniform Resource Identifier), en un servicio web RESTful es un hiper-enlace a un recurso, y es la única forma de intercambiar representaciones entre clientes y servidores (también conocido como ‘path’).
- En un sistema REST la URI no cambia a lo largo del tiempo.
- Si, por ejemplo, en nuestro sistema tenemos información de cursos, podríamos acceder a una lista de cursos disponibles mediante la siguiente URL:
 - <http://everis.com/resources/cursos>
 - Esto nos podría devolver un documento como el siguiente:

```
<?xml version="1.0"?>
<j:Cursos xmlns:j="http://www.jtech.ua.es" xmlns:xlink="http://www.w3.org/1999/xlink">
    <Curso id="1" xlink:href="http://everis.es/resources/cursos/1"/>
    <Curso id="2" xlink:href="http://everis.es/resources/cursos/2"/>
    <Curso id="4" xlink:href="http://everis.es/resources/cursos/4"/>
    <Curso id="6" xlink:href="http://everis.es/resources/cursos/6"/>
</j:Cursos>
```

9. Repaso de conceptos sobre servicios

Obtención de recursos

```
{  
    "id": 1,  
    "name": "Leanne Graham",  
    "username": "Bret",  
    "email": "Sincere@april.biz",  
    "address": {  
        "street": "Kulas Light",  
        "suite": "Apt. 556",  
        "city": "Gwenborough",  
        "zipcode": "92998-3874",  
        "geo": {  
            "lat": "-37.3159",  
            "lng": "81.1496"  
        }  
    },  
    "phone": "1-770-736-8031 x56442",  
    "website": "hildegard.org",  
    "company": {  
        "name": "Romaguera-Crona",  
        "catchPhrase": "Multi-layered client-server neural-net",  
        "bs": "harness real-time e-markets"  
    }  
}
```

<https://jsonplaceholder.typicode.com/users/1>

```
[  
    {  
        "id": 1,  
        "name": "Leanne Graham",  
        "username": "Bret",  
        "email": "Sincere@april.biz",  
        "address": {  
            "street": "Kulas Light",  
            "suite": "Apt. 556",  
            "city": "Gwenborough",  
            "zipcode": "92998-3874",  
            "geo": {  
                "lat": "-37.3159",  
                "lng": "81.1496"  
            }  
        },  
        "phone": "1-770-736-8031 x56442",  
        "website": "hildegard.org",  
        "company": {  
            "name": "Romaguera-Crona",  
            "catchPhrase": "Multi-layered client-server neural-net",  
            "bs": "harness real-time e-markets"  
        }  
    },  
    {  
        "id": 2,  
        "name": "Ervin Howell",  
        "username": "Antonette",  
        "email": "Shanna@melissa.tv",  
        "address": {  
            "street": "Victor Plains",  
            "suite": "Suite 879",  
            "city": "Wisokyburgh",  
            "zipcode": "90566-7771",  
            "geo": {  
                "lat": "-43.9509",  
                "lng": "-34.4618"  
            }  
        },  
        "phone": "010-692-6593 x09125",  
        "website": "anastasia.net",  
        "company": {  
            "name": "Deckow-Crist",  
            "catchPhrase": "Proactive didactic contingency",  
            "bs": "synergize scalable supply-chains"  
        }  
    },  
    {  
        "id": 3,  
        "name": "Clementine Bauch",  
        "username": "Samantha",  
        "email": "Nathan@yesenia.net",  
        "address": {  
            "street": "Douglas Extension",  
            "suite": "Suite 847",  
            "city": "Rathbury",  
            "zipcode": "92998-3874",  
            "geo": {  
                "lat": "-68.6102",  
                "lng": "23.4329"  
            }  
        },  
        "phone": "1-464-946-5464",  
        "website": "yousicily.com",  
        "company": {  
            "name": "Yost-Crona",  
            "catchPhrase": "Proactive didactic contingency",  
            "bs": "synergize scalable supply-chains"  
        }  
    }  
]
```

<https://jsonplaceholder.typicode.com/users>

9. Repaso de conceptos sobre servicios REST

Tipos de peticiones HTTP

- El desarrollo de REST está centrado en el concepto de nombres (intercambio de recursos).
- Un servicio RESTful modifica el estado de los datos a través de la representación de los recursos.
- Un servicio RESTful limita la ambigüedad en el diseño y la implementación restringiendo las operaciones que podemos realizar con los recursos: *create, retrieve, update, delete*:
- La correspondencia de estas acciones con métodos HTTP son:

Acción sobre los datos	Protocolo HTTP equivalente
CREATE	POST
RETRIEVE	GET
UPDATE	PUT
DELETE	DELETE

Un servicio REST puede ejecutar lógica en el lado del servidor, pero cada respuesta debe ser una representación del recurso del dominio en cuestión

9. Repaso de conceptos sobre servicios REST

Estructura JSON

- JSON es una representación muy utilizada para formatear los recursos solicitados a un servicio web RESTful.
- Se trata de ficheros de texto planos que pueden ser manipulados muy fácilmente utilizando *Javascript*.
- Los elementos están contenidos entre llaves.
- Los valores de los elementos se organizan en pares con la estructura “nombre:valor” separados por comas.
- Las secuencias de elementos están contenidas entre corchetes.
- Ejemplo:

```
{ "firstName": "John", "lastName": "Smith",
  "age": 25, "address":
    {
      "streetAddress": "21 2nd Street",
      "city": "New York", "state": "NY",
      "postalCode": "10021" },
  "phoneNumber":
    [ { "type": "home", "number": "212 555-1234"}, { "type": "fax", "number": "646 555-4567"} ]}
```



10

Creación de API REST con Spring Boot

10. Creación de API REST con Spring Boot

Una vez recordados los principios en los que se basan los servicios **REST**, vamos a incorporarlos a nuestro proyecto.

Vamos a trabajar principalmente con el paquete **com.nttdata.controller.rest**

Retomaremos la clase **EmpleadoRestController**, pero le vamos a cambiar el **endpoint** para que se encargue del mapeo **"/api/empleados"**:

```
@RestController
@RequestMapping ("/api/empleados")
public class EmpleadoRestController {
    @Autowired
    EmpleadoService empleadoService;

    @Cacheable (value="empleados")
    @GetMapping
    public List<Empleado> listarEmpleados() {
        try {
            Thread.sleep(1500);
        } catch (InterruptedException exjj) { }
        return empleadoService.listar();
    }
}
```

Este controlador se va a encargar de atender las peticiones REST sobre empleados.

10. Creación de API REST con Spring Boot

Recordamos el **entity** en el que nos basaremos:

```
@Entity  
@Table  
public class Empleado {  
    @Id  
    @Column  
    private Integer id;  
  
    @Column (nullable=false, length=30)  
    private String nombre;  
  
    @Column  
    private String apellidos;
```

Importante revisar que tiene creados todos los getters y setters.

10. Creación de API REST con Spring Boot

Ya tenemos implementado el método GET que nos devuelve un listado de empleados:

```
@GetMapping  
public List<Empleado> listarEmpleados() {  
    try {  
        Thread.sleep(1500);  
    } catch (InterruptedException exjj) { }  
    return empleadoService.listar();  
}
```

Y al acceder a la URL <http://localhost/app/api/empleados> obtenemos:

The screenshot shows the Postman application interface. At the top, there's a header bar with 'GET' selected, a dropdown for 'Authorization', and the URL 'http://localhost/app/api/empleados'. Below the header are tabs for 'Params', 'Authorization', 'Headers (8)', 'Body', 'Pre-request Script', 'Tests', and 'Settings'. Under 'Params', there's a 'Query Params' section with a 'KEY' column, a 'VALUE' column, and a 'DESCRIPTION' column. The 'Value' column contains the URL 'http://localhost/app/api/empleados'. The 'DESCRIPTION' column shows status information: 'Status: 200 OK', 'Time: 23 ms', and 'Size: 1.69 KB'. Below these tabs is a table with columns 'Body', 'Cookies (2)', 'Headers (10)', and 'Test Results'. The 'Body' tab is active, showing a preview of the response in 'Pretty' format. The response content is a partial HTML document with meta tags and a title 'Please sign in'. The code block below shows the full JSON representation of this response.

```
1  <!DOCTYPE html>  
2  <html lang="en">  
3  <head>  
4      <meta charset="utf-8">  
5      <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">  
6      <meta name="description" content="">  
7      <meta name="author" content="">  
8      <title>Please sign in</title>  
9      <link href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta/css/bootstrap.min.css" rel="stylesheet"  
10         integrity="sha384-/Y6pD6FV/Vv2HjnA6t+vslU6fwYkjCFTcEpHbNJ0lyAFsXTsjBbfadJzALeQsN6M" crossorigin="anonymous">  
11      <link href="https://getbootstrap.com/docs/4.0/examples/signin/signin.css" rel="stylesheet"  
12         crossorigin="anonymous" />  
13  </head>  
14  <body>  
15      <div class="container">  
16          <form class="form-signin" method="post" action="/app/login">  
17              <h2 class="form-signin-heading">Please sign in</h2>
```

10. Creación de API REST con Spring Boot

GET <http://localhost/app/api/empleados>

Params Authorization Headers (8) Body Pre-request Script Tests Settings

Query Params

KEY	VALUE	DESCRIPTION
-----	-------	-------------

Body Cookies (2) Headers (10) Test Results

Status: 200 OK Time: 23 ms Size: 1.69 KB

Pretty Raw Preview Visualize HTML

```
1 <!DOCTYPE html>
2 <html lang="en">
3
4 <head>
5   <meta charset="utf-8">
6   <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
7   <meta name="description" content="">
8   <meta name="author" content="">
9   <title>Please sign in</title>
10  <link href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta/css/bootstrap.min.css" rel="stylesheet"
11    integrity="sha384-Y6pD6FVv2HJnA6t+vslU6fwYXjCFtcEpHbNJ0lyAFsXTsjBbfaDjzALeQsN6M" crossorigin="anonymous">
12  <link href="https://getbootstrap.com/docs/4.0/examples/signin/signin.css" rel="stylesheet"
13    crossorigin="anonymous" />
14 </head>
15
16 <body>
17   <div class="container">
18     <form class="form-signin" method="post" action="/app/login">
19       <h2 class="form-signin-heading">Please sign in</h2>
20       <p>
21         <label for="username" class="sr-only">Username</label>
```

10. Creación de API REST con Spring Boot

Tenemos que permitir que se ejecuten los servicios REST sin autenticación (lo ideal sería implementar una autenticación basada en JWT o similar):

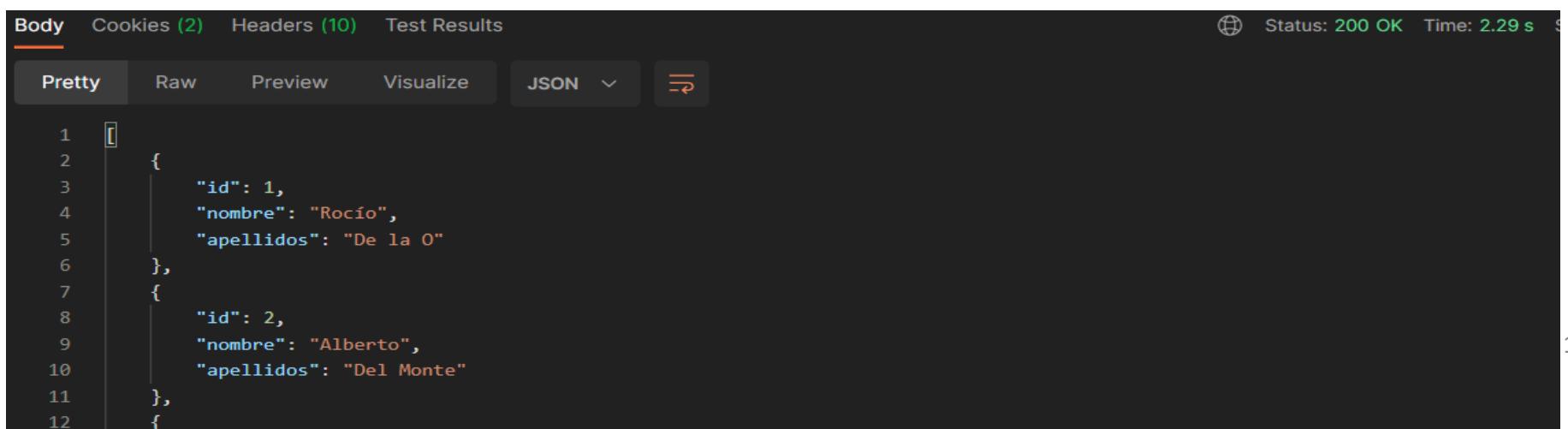
```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity (prePostEnabled = true)
public class WebSecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Autowired
    private UserDetailsService serviceUsuario;

    String[] resources = new String[] { "/include/**", "/js/**", "/css/**", "/api/**"};
```



Y al acceder a la URL <http://localhost/app/api/empleados> obtenemos:



The screenshot shows a Postman request for the URL <http://localhost/app/api/empleados>. The response status is 200 OK, and the response body is a JSON array of employee objects:

```
[{"id": 1, "nombre": "Rocío", "apellidos": "De la O"}, {"id": 2, "nombre": "Alberto", "apellidos": "Del Monte"}]
```

The JSON response is displayed in the "Pretty" tab of the Postman interface.

10. Creación de API REST con Spring Boot

A esta url se ha respondido porque hemos implementado el método **GET** para nuestra **API REST**, permitiendo la consulta de los usuarios de nuestro dominio.

Ahora vamos a implementar los otros métodos principales que nos permitan **modificar, eliminar y añadir** nuevos usuarios.

Primeramente vamos a realizar el método que permita **insertar un nuevo empleado**, el cuál deberá ser un método **POST**:

```
@PostMapping  
public void insertarEmpleado (@RequestBody Empleado empleado) {  
    empleado.setId(null);  
    empleadoService.insertar(empleado)];  
}
```

10. Creación de API REST con Spring Boot

Aunque le dejaremos a **JPA** que se encargue de la **primary key**.

```
@Entity  
@Table  
public class Empleado {  
    @Id  
    @GeneratedValue(strategy=GenerationType.IDENTITY)  
    @Column  
    private Integer id;
```



También sería positivo que limpiásemos la **caché** de ‘empleados’ tras insertar un nuevo empleado:

```
@CacheEvict(value="empleados", allEntries=true)  
@PostMapping  
public void insertarEmpleado (@RequestBody Empleado empleado) {
```



10. Creación de API REST con Spring Boot

Ahora ya sólo nos quedará crear método ‘**insertar**’ en el servicio:

```
public interface EmpleadoService {  
    void registrar (String name);  
    List<Empleado> listar();  
    public List<Empleado> listarFiltroNombre(String cad);  
    List<Empleado> listarConJPA (Integer pID, String contiene);  
    void insertar(Empleado empleado); ←  
}
```

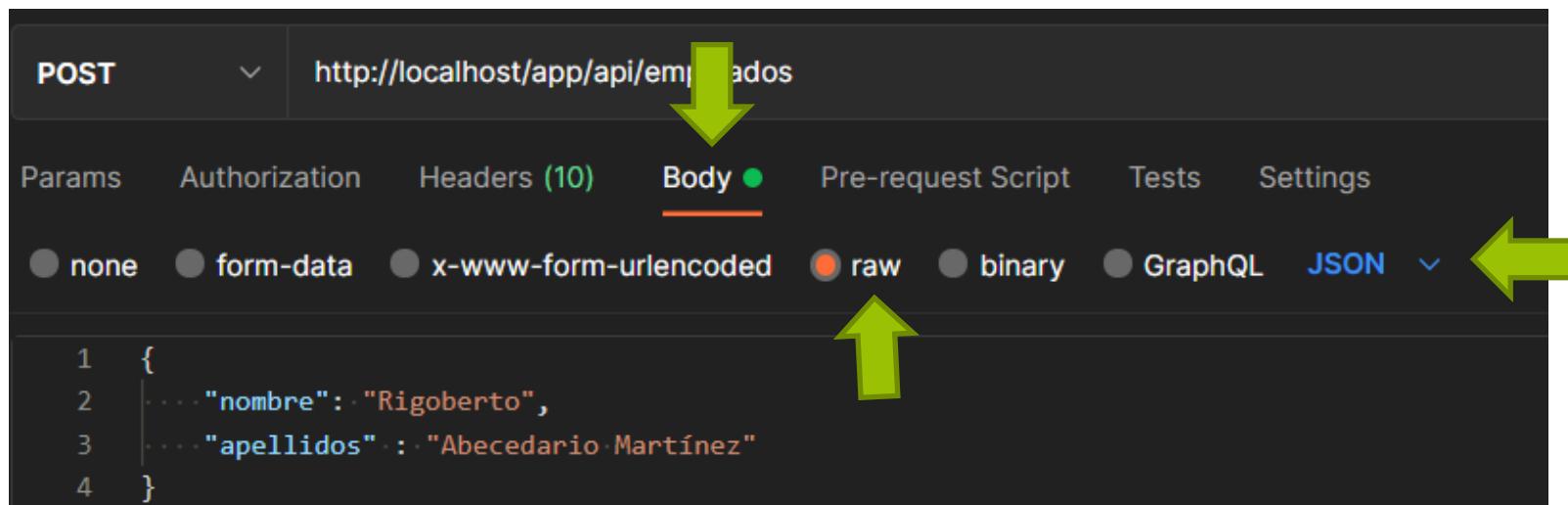


```
@Override  
public void insertar(Empleado empleado) {  
    empleadoRepo.save(empleado);  
}
```

10. Creación de API REST con Spring Boot

Pero al ser un método **POST** no vamos a poder probarlo directamente en el navegador, tendremos que hacer una petición **POST** y para ello usaremos el software **POSTMAN**.

Así crearemos una petición **POST** indicando en la URL '<http://localhost/app/api/empleados>', y en la parte del Body seleccionaremos '**RAW**' y posteriormente **JSON**, y aquí es dónde añadiremos nuestro nuevo usuario (el id no se lo pasamos ya que se lo asigna el sistema):



10. Creación de API REST con Spring Boot

Tras ejecutarlo nos deberá devolver un **código 200** como que todo ha ido **OK**:



Body Cookies Headers (4) Test Results Status: 200 OK Time: 181ms Size: 123 B Save Response ▾

Si ahora volvemos a consultar el listado de clientes con una petición GET (ya sea por POSTMAN o modo web), obtendremos que aparece el nuevo usuario añadido:

```
12      {
13          "id": 4,
14          "nombre": "Lorena",
15          "apellidos": "Muñoz"
16      },
17      {
18          "id": 8,
19          "nombre": "Lucía",
20          "apellidos": "De la O"
21      },
22      {
23          "id": 9,
24          "nombre": "Roberto",
25          "apellidos": "Del Monte"
26      },
27      {
28          "id": 10,
29          "nombre": "Rigoberto",
30          "apellidos": "Abecedario Martínez"
31      }
32 ]
```



10. Creación de API REST con Spring Boot

Ya implementados la creación y la consulta, vamos a ver cómo quedarían de una forma sencilla la modificación:

```
@CacheEvict(value="empleados", allEntries=true)
@PutMapping
public void modificarEmpleado (@RequestBody Empleado empleado) {
    empleadoService.modificar(empleado);
}
```



```
public interface EmpleadoService {
    void modificar(Empleado empleado);
```



```
@Override
public void modificar(Empleado empleado) {
    empleadoRepo.save(empleado);
}
```

10. Creación de API REST con Spring Boot

Tras ello ejecutamos:



```
PUT http://localhost/app/api/empleados
```

Params Authorization Headers (10) **Body** Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1 {
2   "id": 1,
3   "nombre": "Rocío",
4   "apellidos": "De la O"
5 }
```

```
[{"id": 1, "nombre": "Rocío", "apellidos": "De la O"}, {"id": 2, "nombre": "Alberto", "apellidos": "Del Monte"}]
```

Consultamos de nuevo el listar empleados:



```
1 [
2   {
3     "id": 1,
4     "nombre": "Rocío",
5     "apellidos": "De la O"
6   },
7   {
8     "id": 2,
9     "nombre": "Alberto",
10    "apellidos": "Del Monte"
11  }
12 ]
```

Nota: si hemos parado y arrancado el servidor, nos habrá desaparecido el usuario creado con el método 'insertar'.

10. Creación de API REST con Spring Boot

Y ya por último nos quedará la eliminación (método **DELETE**):

```
@CacheEvict(value="empleados", allEntries=true)
@DeleteMapping (value="/{id}")
public void eliminarEmpleado (@PathVariable("id") Integer id) {
    empleadoService.eliminarEmpleado(id);
}
```



```
public interface EmpleadoService {
    void eliminarEmpleado(Integer id);
```



```
@Override
public void eliminarEmpleado(Integer id) {
    empleadoRepo.deleteById(id);
}
```

```
Pretty Raw Preview Visualize JSON ⌂
1 [
2   {
3     "id": 1,
4     "nombre": "RociITO",
5     "apellidos": "De la O"
6   },
7   {
8     "id": 2,
9     "nombre": "Alberto",
10    "apellidos": "Del Monte"
11  }
]
```

10. Creación de API REST con Spring Boot

Para probarlo basta con realizar una petición **DELETE** a <http://localhost/app/api/empleados/1> (si por ejemplo queremos eliminar el empleado con id=1):

The screenshot shows the Postman interface with a DELETE request to <http://localhost/app/api/empleados/1>. The 'Params' tab is selected, showing a table for 'Query Params' with columns: KEY, VALUE, and DESCRIPTION. There is one row with 'Key' in the KEY column and 'Value' in the VALUE column, both empty. The 'Body' tab is also visible at the bottom.

Y al obtener el listado vemos que ha desaparecido el usuario con id=1:

```
Pretty Raw Preview Visualize JSON ⌂
1 [
2   {
3     "id": 2,
4     "nombre": "Alberto",
5     "apellidos": "Del Monte"
6   }
]
```

10. Creación de API REST con Spring Boot

Una vez implementada nuestra API, vamos a mejorar algunos aspectos.

Lo primero de todo será crear en el service un método que me recupere un empleado en base a su id:

```
public interface EmpleadoService {  
    Empleado getById(Integer id);
```

```
@Override  
public Empleado getById(Integer id) {  
    return empleadoRepo.findById(id).orElse(null);  
}
```

10. Creación de API REST con Spring Boot

Tras ello, vamos a implementar un servicio REST que dado un id, devuelva el usuario correspondiente, y si no existe ningún usuario con dicho id devuelva un 404.

```
@GetMapping (value="/{id}")
public ResponseEntity<Empleado> devuelveEmpleado(@PathVariable("id") Integer id) {
    Empleado emp = empleadoService.getById(id);
    if (emp==null)
        return new ResponseEntity<> (null, HttpStatus.NOT_FOUND);
    else
        return new ResponseEntity<>(emp, HttpStatus.OK);
}
```

GET http://localhost/app/api/empleados/1

Status: 200 OK

```
{ "id": 1, "userName": "lucas", "password": "pass1" }
```

GET http://localhost/app/api/empleados/20

Status: 404 Not Found

10. Creación de API REST con Spring Boot

Ahora vamos a tener en cuenta también las excepciones y códigos de error en el POST:

```
@CacheEvict(value="empleados", allEntries=true)
@PostMapping
public ResponseEntity<List<Empleado>> insertarEmpleado_v2 (@RequestBody Empleado empleado) {
    try {
        empleado.setId(null);
        empleadoService.insertar(empleado);
        return new ResponseEntity<> (empleadoService.listar(), HttpStatus.CREATED);
    }
    catch (Exception ex) {
        return new ResponseEntity<> (new ArrayList(), HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

The screenshot shows a POST request in Postman to the URL `http://localhost/app/api/empleados`. The request method is POST, and the URL is specified in the header. The 'Body' tab is selected, showing a JSON array of three employee objects. A large green arrow points from the request to the response.

Body

```
[{"id": 9, "nombre": "Roberto", "apellidos": "Del Monte"}, {"id": 10, "nombre": "Rigoberto", "apellidos": "Abecedario Martínez"}, {"id": 11, "nombre": "AAA", "apellidos": "BBBB CCCCz"}]
```

```
    @Override  
    public Empleado insertar(Empleado empleado) {  
        return empleadoRepo.save(empleado);  
    }
```

10. Creación de API REST con Spring Boot

Mejorando nuestra API REST:

```
@CacheEvict(value="empleados", allEntries=true)  
@PostMapping  
public ResponseEntity<Empleado> insertarEmpleado_v3 (@RequestBody Empleado empleado) {  
    try {  
        HttpHeaders headers = new HttpHeaders();  
        if (empleado.getId()!=null) {  
            headers.set("Message", "Para dar de alta un nuevo empleado, el ID debe llegar vacío");  
            return new ResponseEntity<>(headers, HttpStatus.NOT_ACCEPTABLE);  
        }  
        else if (empleado.getNombre()==null || empleado.getNombre().equals("")  
                || empleado.getApellidos()==null || empleado.getApellidos().equals("")) {  
            headers.set("Message", "Ni NOMBRE ni APELLIDOS pueden ser nulos");  
            return new ResponseEntity<>(headers, HttpStatus.NOT_ACCEPTABLE);  
        }  
  
        Empleado emp = empleadoService.insertar(empleado);  
        URI newPath = new URI("/api/empleados/" + emp.getId());  
        headers.setLocation(newPath);  
        headers.set("Message", "Empleado insertado correctamente con id: " + emp.getId());  
  
        return new ResponseEntity<> (emp, headers, HttpStatus.CREATED);  
    }  
    catch (Exception ex) {  
        return new ResponseEntity<> (null, HttpStatus.INTERNAL_SERVER_ERROR);  
    }  
}
```

10. Creación de API REST con Spring Boot

Mejorando nuestra API REST:

1)

POST http://localhost/app/api/empleados

```
{  
    "nombre": "11111",  
    "apellidos": ""  
}
```



Body	Cookies (2)	Headers (10)	Test Results	
KEY				Status: 406 Not Acceptable Time: 20 ms Size: 344 B
Message ⓘ				Ni NOMBRE ni APELLIDOS pueden ser nulos

2)

POST http://localhost/app/api/empleados

```
{  
    "nombre": "11111",  
    "apellidos": "222222"  
}
```



Body	Cookies (2)	Headers (11)	Test Results	
KEY				Status: 201 Created Time: 377 ms Size: 369 B
Location ⓘ				/api/empleados/45
Message ⓘ				Cliente insertado correctamente con id: 45



11

Definición de test unitarios

11. Definición de test unitarios

Cuando tenemos una aplicación pequeña es sencillo probar todas las funcionalidades, pero a medida que va creciendo cada vez necesitamos más tiempo y se complica el poder probar todas las funcionalidades implementadas a bajo nivel.

Por esto lo ideal es automatizar nuestros test unitarios, para que ante cualquier fallo en nuestra aplicación salte automáticamente un fallo en el test.

Si tenemos nuestros test automatizados vamos a poder dar un siguiente paso y montar un entorno de integración continua (por ejemplo con Jenkins).

No va a tener sentido probar métodos que nos proporcione una librería (como por ejemplo los que ya nos da JPA), ni servicios que lo único que hagan sea mapear la información recogida en la capa repository/DAO, pero sí es importante que probemos toda funcionalidad donde hayamos metido cierta lógica.

11. Definición de test unitarios

Primeramente vamos a preparar el fichero de propiedades para que nuestras pruebas de test tiren de otra BBDD distinta. Para ello crearemos el fichero '**application-test.properties**' en **src/test/resources** con el siguiente contenido:

```
spring.main.web-application-type=none
spring.datasource.url=jdbc:h2:file:./springboot_test;AUTO_SERVER=TRUE
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=us
spring.datasource.password=pa
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=update
spring.sql.init.mode =always
```

11. Definición de test unitarios

A continuación crearemos el fichero **data.sql** en **src/test/resources** conteniendo:

```
insert into rol (id, rol)
    select 1, 'ADMIN' from dual where not exists (select 1 from rol where id = 1);

insert into rol (id, rol)
    select 2, 'GESTOR' from dual where not exists (select 1 from rol where id = 2);

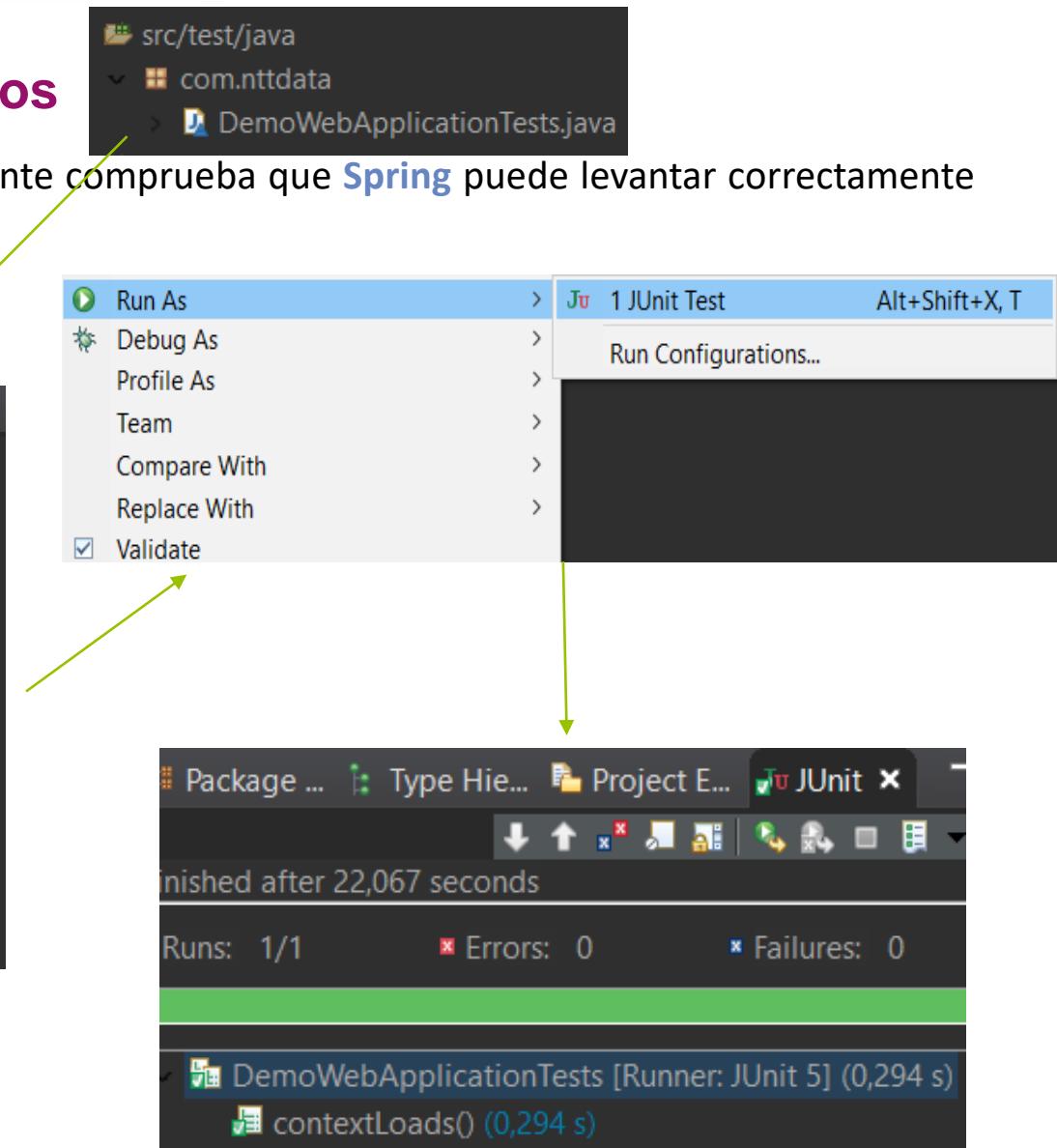
/* pass = 1111 */
insert into usuario (username, nombre, password, rol_id)
    select 'user1', 'Julian Hernandez',
'$2a$10$5xOe75pbLcAjp0TbVWaluunrSshgYdH82YNwGd.b0Os4hAWbIEkry', 1 from dual where not
exists (select 1 from usuario where username = 'user1');

insert into usuario (username, nombre, password, rol_id)
    select 'user2', 'Empleado de NTTData',
'$2a$10$5xOe75pbLcAjp0TbVWaluunrSshgYdH82YNwGd.b0Os4hAWbIEkry', 2 from dual where not
exists (select 1 from usuario where username = 'user2');
```

11. Definición de test unitarios

Ejecutamos el primer test que simplemente comprueba que **Spring** puede levantar correctamente los **beans** de la aplicación:

```
DemoWebApplicationTests.java x
1 package com.nttdata;
2
3+import org.junit.jupiter.api.Test;
4
5 @SpringBootTest
6 class DemoWebApplicationTests {
7
8
9     @Test
10    void contextLoads() {
11    }
12
13 }
```



```

@Entity
@Table
public class Empleado {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column
    private Integer id;

    @Column (nullable=false, length=30)
    private String nombre;

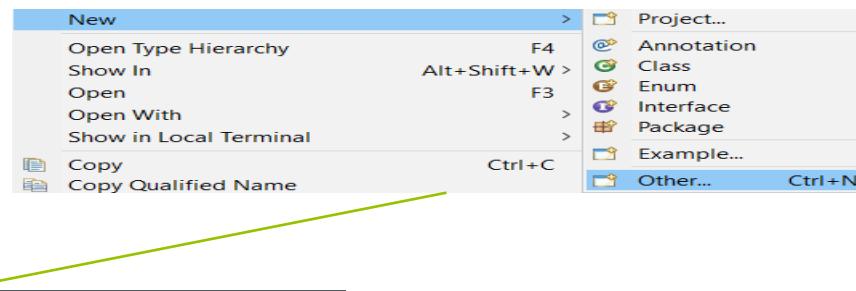
    @Column
    private String apellidos;
}

```

11. Definición de test unitarios

Test unitario para un entity:

Pulsamos sobre el nombre de la clase (botón derecho) → New → Other... → Junit Test Case



JUnit Test Case

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

- New JUnit 3 test New JUnit 4 test New JUnit Jupiter test

Source folder: DemoWebREST/src/test/java

Package: com.nttdata.repository.entity

Name: EmpleadoTest

Sin marcar nada

```

package com.nttdata.repository.entity;

import static org.junit.jupiter.api.Assertions.*;

class EmpleadoTest {

    @Test
    void test() {
        fail("Not yet implemented");
    }
}

```

11. Definición de test unitarios

Test unitario para un entity:

```
package com.nttdata.repository.entity;

import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;

class EmpleadoTest {

    @Test
    void test() {
        Empleado e1 = new Empleado();

        e1.setId(1);
        assertEquals(1, e1.getId(), "Mismo id");

        String nombre="Nombre Prueba";
        e1.setNombre(nombre);
        assertEquals (nombre, e1.getNombre(), "Mismo nombre");

        String apellidos="Apellidos Prueba";
        e1.setApellidos(apellidos);
        assertEquals(apellidos, e1.getApellidos(), "Mismos apellidos");

        Empleado e2 = new Empleado();
        e2.setId(1);
        e2.setNombre(nombre);
        e2.setApellidos(apellidos);
        assertEquals(e1, e2, "Mismo empleado");
    }
}
```

¿Qué está pasando?

The screenshot shows the JUnit 5 test results for the 'EmpleadoTest' class. The overall status is 'Finished after 0,237 seconds'. A summary bar at the top indicates 'Runs: 1/1' (1 run, 0 errors), 'Failures: 1', and has a large red 'X' icon. Below this, the test class 'EmpleadoTest [Runner: JUnit 5] (0,004 s)' is expanded, showing a single test method 'test() (0,004 s)' which failed. At the bottom, there are icons for 'Failure Trace', 'Run All', 'Stop', and 'Copy All'. The failure trace details show an 'AssertionFailedError' with the message 'Mismo empleado ==> expected: <com.nttdata.repository.entity.Empleado> but was: <com.nttdata.repository.entity.Empleado@432f33d>'.

11. Definición de test unitarios

Test unitario para un entity:

Debemos redefinir en el entity el método **equals** para que dos entidades sean iguales sólo si comparten el mismo **id**:

```
@Override  
public int hashCode() {  
    return Objects.hash(id);  
}  
  
@Override  
public boolean equals(Object obj) {  
    if (this == obj) {  
        return true;  
    }  
    if (!(obj instanceof Empleado)) {  
        return false;  
    }  
    Empleado other = (Empleado) obj;  
    return Objects.equals(id, other.id);  
}
```



```
Finished after 0,182 seconds  
Runs: 1/1 Errors: 0 Failures: 0  
> EmpleadoTest [Runner: JUnit 5] (0,036 s)
```

11. Definición de test unitarios

Test unitario para un entity:

Metemos también en el test la prueba del método hash para asegurarnos:

```
@Test
void test() {
    Empleado e1 = new Empleado();

    e1.setId(1);
    assertEquals(1, e1.getId(), "Mismo id");

    String nombre="Nombre Prueba";
    e1.setNombre(nombre);
    assertEquals (nombre, e1.getNombre(), "Mismo nombre");

    String apellidos="Apellidos Prueba";
    e1.setApellidos(apellidos);
    assertEquals(apellidos, e1.getApellidos(), "Mismos apellidos");

    Empleado e2 = new Empleado();
    e2.setId(1);
    e2.setNombre(nombre);
    e2.setApellidos(apellidos);
    assertEquals(e1, e2, "Mismo empleado");

    assertEquals(Objects.hash(e1.getId()), e1.hashCode(), "Mismo Hash");
}
```



Finished after 0,182 seconds

Runs: 1/1	Errors: 0	Failures: 0
-----------	-----------	-------------

> EmpleadoTest [Runner: JUnit 5] (0,036 s)



11. Definición de test unitarios

Test unitario para el repository: (EmpleadoRepoImpl)

```
@Repository
public class EmpleadoRepoImpl implements EmpleadoRepo {
    private static Logger LOG = org.slf4j.LoggerFactory.getLogger(EmpleadoRepoImpl.class);

    @PersistenceContext
    EntityManager entityManager;

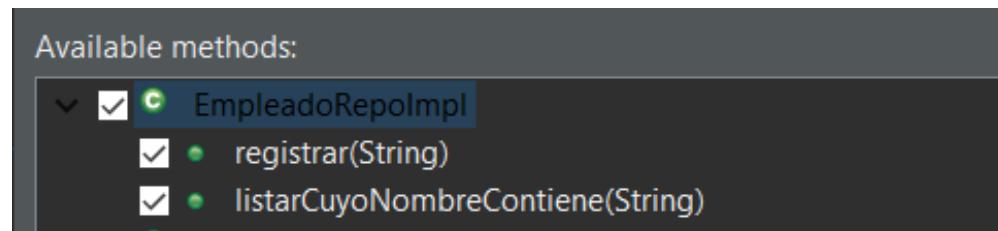
    @Override
    public void registrar(String nombre) {
        LOG.info("Se ha saludado al empleado: " + nombre);
    }

    @Override
    public List<Empleado> listarCuyoNombreContiene(String texto_nombre) {
        Query query = entityManager.createNativeQuery("SELECT * FROM empleado " +
                "WHERE nombre LIKE ? ", Empleado.class);
        query.setParameter(1, "%" + texto_nombre + "%");
        return query.getResultList();
    }
}
```

Pulsamos sobre el nombre de la clase (botón derecho) → New → Other... → Junit Test Case

11. Definición de test unitarios

Test unitario para el repository: (EmpleadoRepoImpl)



The screenshot shows a Java code editor with a dark theme. At the top, there is a list of "Available methods" for the class "EmpleadoRepoImpl". The methods listed are "registrar(String)" and "listarCuyoNombreContiene(String)". Both methods have checkmarks next to them, indicating they are selected or part of the current context. Below this, the actual Java code for the test class is visible:

```
package com.nttdata.repository.impl;

import static org.junit.jupiter.api.Assertions.*;

class EmpleadoRepoImplTest {

    @Test
    void testRegistrar() {
        fail("Not yet implemented");
    }

    @Test
    void testListarCuyoNombreContiene() {
        fail("Not yet implemented");
    }
}
```

11. Definición de test unitarios

Test unitario para el repository: (EmpleadoRepoImpl)

```
package com.nttdata.repository.impl;

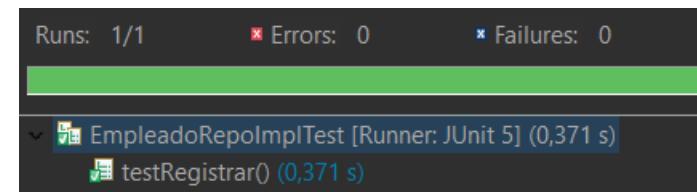
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.TestPropertySource;

import com.nttdata.repository.EmpleadoRepoJPA;

@TestPropertySource(locations = "classpath:application-test.properties")
@SpringBootTest
class EmpleadoRepoImplTest {

    @Autowired
    EmpleadoRepoJPA repo;

    @Test
    void testRegistrar() {
        repo.registrar("Mensaje prueba");
    }
}
```



11. Definición de test unitarios

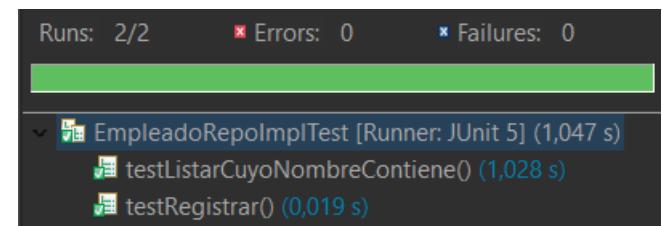
Test unitario para el repository: (EmpleadoRepoImpl)

```
@Test
void testListarCuyoNombreContiene() {
    //GIVEN
    Empleado e1 = new Empleado();
    e1.setNombre("Lucas");
    e1.setApellidos("Ape1 Ape2");
    e1=repo.save(e1);

    Empleado e2 = new Empleado();
    e2.setNombre("Ana");
    e2.setApellidos("Ape1 Ape2");
    e2=repo.save(e2);

    //WHEN:
    List<Empleado> l1 = repo.listarCuyoNombreContiene("u");
    List<Empleado> l2 = repo.listarCuyoNombreContiene("2");
    repo.delete(e1);
    repo.delete(e2);

    //THEN:
    assertAll(
        () -> assertEquals(1, l1.size(), "Sólo 1 empleado contiene la 'u' en el nombre"),
        () -> assertEquals(0, l2.size(), "Ningún empleado contiene un '2' en el nombre")
    );
}
```



11. Definición de test unitarios

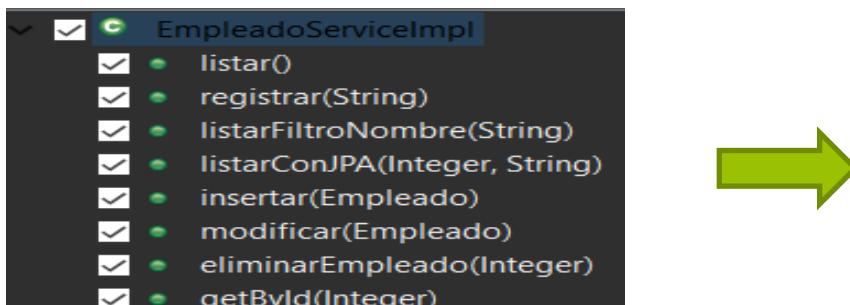
Test unitario para el repository: (EmpleadoRepoImpl)

¿Buena o mala práctica?

```
@BeforeEach  
void setUp() {  
    repo.deleteAll();  
}  
  
@AfterEach  
void tearDown() {  
    repo.deleteAll();  
}
```

11. Definición de test unitarios

Test unitario para el service: (EmpleadoServiceImpl)



```
class EmpleadoServiceImplTest {  
  
    @Test  
    void testListar() {  
        fail("Not yet implemented");  
    }  
  
    @Test  
    void testRegistrar() {  
        fail("Not yet implemented");  
    }  
  
    @Test  
    void testListarFiltroNombre() {  
        fail("Not yet implemented");  
    }  
  
    @Test  
    void testListarConJPA() {  
        fail("Not yet implemented");  
    }  
  
    @Test  
    void testInsertar() {  
        fail("Not yet implemented");  
    }  
  
    @Test  
    void testModificar() {  
        fail("Not yet implemented");  
    }  
  
    @Test  
    void testEliminarEmpleado() {  
        fail("Not yet implemented");  
    }  
  
    @Test  
    void testGetById() {  
        fail("Not yet implemented");  
    }  
}
```

11. Definición de test unitarios

Test unitario para el service: (EmpleadoServiceImpl)

```
@TestPropertySource(locations = "classpath:application-test.properties")
@SpringBootTest
class EmpleadoServiceImplTest {

    @Autowired
    EmpleadoService service;

    @Autowired
    EmpleadoRepoJPA repo;

    @BeforeEach
    void setUp() throws Exception {
    }

    @AfterEach
    void tearDown() throws Exception {
    }
}
```

```
class EmpleadoServiceImplTest {  
    private Empleado e1, e2;  
  
    @Autowired  
    EmpleadoService service;  
  
    @Autowired  
    EmpleadoRepoJPA repo;  
  
    @BeforeEach  
    void setUp() throws Exception {  
        repo.deleteAll();  
        e1 = new Empleado();  
        e1.setNombre("Manuel");  
        e1.setApellidos("Muñoz Martínez");  
        e1=repo.save(e1);  
  
        e2 = new Empleado();  
        e2.setNombre("Ana");  
        e2.setApellidos("Alexa Armani");  
        e2=repo.save(e2);  
    }  
  
    @AfterEach  
    void tearDown() throws Exception {  
        repo.deleteAll();  
    }  
}
```

```
@Test  
void testRegistrar() {  
    service.registrar("texto prueba");  
}
```

```
@Test  
void testListar() {  
    //GIVEN:  
    //Hay dos usuarios en la BBDD:  
  
    //WHEN:  
    List<Empleado> le = service.listar();  
  
    //THEN:  
    assertEquals(2, le.size(), "Hay dos empleados en BBDD");  
}
```

```
@Test  
void testListarFiltroNombre() {  
    //GIVEN:  
    //Hay dos usuarios en la BBDD: 'Manuel' y 'Ana'  
    assertEquals(2, service.listar().size(), "Hay dos empleados en BBDD");  
  
    //WHEN:  
    List<Empleado> le = service.listarFiltroNombre("u");  
  
    //THEN:  
    assertEquals(1, le.size(), "Hay un empleado en BBDD con nombre que contenga una 'u'");  
}
```

```
@Test
void testListarConJPA() {
    //GIVEN:
    //Hay dos usuarios en la BBDD: 'Manuel' y 'Ana'
    assertEquals(2, service.listar().size(), "Hay dos empleados en BBDD");

    //WHEN:
    List<Empleado> le = service.listarConJPA(e1.getId(), "%a%");

    //THEN:
    assertEquals(1, le.size(), "Solo hay un empleado con id>" + e1.getId() + " y que contenga 'a'");
}

@Test
void testInsertar() {
    //GIVEN:
    //Hay dos usuarios en la BBDD:
    assertEquals(2, service.listar().size(), "Hay dos empleados en BBDD");

    //WHEN:
    Empleado e3 = new Empleado();
    e3.setNombre("N3");
    e3.setApellidos("AP3");
    e3 = service.insertar(e3);

    //THEN:
    assertEquals(3, service.listar().size(), "Hay dos empleados en BBDD");
}
```

```
@Test
void testModificar() {
    //GIVEN:
    //Hay dos usuarios en la BBDD:
    assertEquals(2, service.listar().size(), "Hay dos empleados en BBDD");

    //WHEN:
    String nuevoNombre= "Nuevo Nombre";
    e2.setNombre(nuevoNombre);
    service.modificar(e2);

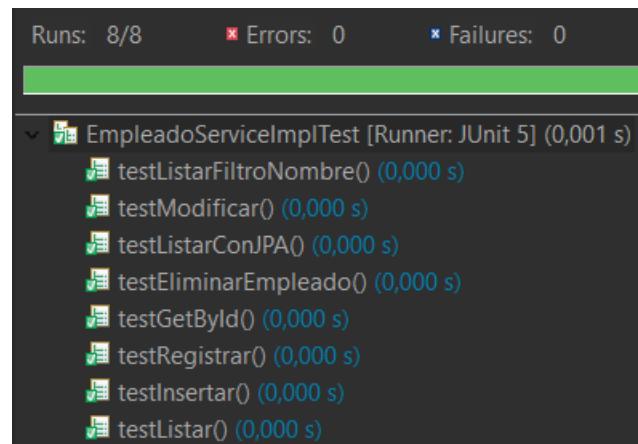
    //THEN:
    assertEquals(2, service.listar().size(), "Sigue habiendo dos empleados en BBDD");
    assertEquals(nuevoNombre, service.getById( e2.getId() ).getNombre(),
                "Ha sido modificado el nombre del empleado con id: "+e2.getId());
}
```

```
@Test
void testEliminarEmpleado() {
    //GIVEN:
    //Hay dos usuarios en la BBDD:
    assertEquals(2, service.listar().size(), "Hay dos empleados en BBDD");

    //WHEN:
    service.eliminarEmpleado(e2.getId());

    //THEN:
    assertEquals(1, service.listar().size(), "Solo queda un empleado en BBDD");
}
```

```
@Test  
void test GetById() {  
    //GIVEN:  
    //Hay dos usuarios en la BBDD:  
  
    //WHEN:  
    Empleado e3 = service.getById( e2.getId() );  
  
    //THEN:  
    assertNotNull(e3, "Encontrado empleado con id: "+e2.getId());  
}
```



11. Definición de test unitarios

Test unitario para el controller: (EmpleadoRestController)

Which method stubs would you like to create?

- setUpBeforeClass() tearDownAfterClass()
- setUp() tearDown()
- constructor

Available methods:

- ✓ EmpleadoRestController
 - ✓ • listarEmpleados()
 - ✓ • insertarEmpleado(Empleado)
 - ✓ • modificarEmpleado(Empleado)
 - ✓ • eliminarEmpleado(Integer)
 - ✓ • devuelveEmpleado(Integer)
 - ✓ • insertarEmpleado_v2(Empleado)
 - ✓ • insertarEmpleado_v3(Empleado)



```
package com.nttdata.controller.rest;

import static org.junit.jupiter.api.Assertions.*;

class EmpleadoRestControllerTest {

    @BeforeEach
    void setUp() throws Exception {
    }

    @AfterEach
    void tearDown() throws Exception {
    }

    @Test
    void testListarEmpleados() {
        fail("Not yet implemented");
    }

    @Test
    void testInsertarEmpleado() {
        fail("Not yet implemented");
    }

    @Test
    void testModificarEmpleado() {
        fail("Not yet implemented");
    }

    @Test
    void testEliminarEmpleado() {
        fail("Not yet implemented");
    }

    @Test
    void testDevuelveEmpleado() {
        fail("Not yet implemented");
    }

    @Test
    void testInsertarEmpleado_v2() {
        fail("Not yet implemented");
    }

    @Test
    void testInsertarEmpleado_v3() {
        fail("Not yet implemented");
    }
}
```

11. Definición de test unitarios

Test unitario para el controller: (EmpleadoRestController)

```
@TestPropertySource(locations = "classpath:application-test.properties")
@SpringBootTest
class EmpleadoRestControllerTest {

    private Empleado e1, e2;

    @Autowired
    EmpleadoRestController controller;

    @Autowired
    EmpleadoService service;

    @Autowired
    EmpleadoRepoJPA repo;

    @BeforeEach
    void setUp() throws Exception {
        repo.deleteAll();
        e1 = new Empleado();
        e1.setNombre("Manuel");
        e1.setApellidos("Muñoz Martínez");
        e1=repo.save(e1);

        e2 = new Empleado();
        e2.setNombre("Ana");
        e2.setApellidos("Alexa Armani");
        e2=repo.save(e2);
    }

    @AfterEach
    void tearDown() throws Exception {
        repo.deleteAll();
    }
}
```

11. Definición de test unitarios

Test unitario para el controller: (EmpleadoRestController)

```
@Test
void testListarEmpleados() {
    //GIVEN:
    //Hay dos usuarios en la BBDD:

    //WHEN:
    List<Empleado> le = controller.listarEmpleados();

    //THEN:
    assertEquals(2, le.size(), "Hay dos empleados en BBDD");
}
```

```
@Test
void testInsertarEmpleado() {
    //GIVEN:
    //Hay dos usuarios en la BBDD:
    assertEquals(2, service.listar().size(), "Hay dos empleados en BBDD");

    //WHEN:
    Empleado e3= new Empleado();
    e3.setNombre("N3");
    e3.setApellidos("AP3");
    controller.insertarEmpleado_v3(e3);

    //THEN:
    assertEquals(3, service.listar().size(), "Hay dos empleados en BBDD");
}
```

11. Definición de test unitarios

Test unitario para el controller: (EmpleadoRestController)

```
@Test
void testModificarEmpleado() {
    //GIVEN:
    //Hay dos usuarios en la BBDD:
    assertEquals(2, service.listar().size(), "Hay dos empleados en BBDD");

    //WHEN:
    String nuevoNombre= "Nuevo Nombre";
    e2.setNombre(nuevoNombre);
    controller.modificarEmpleado(e2);

    //THEN:
    assertEquals(2, service.listar().size(), "Sigue habiendo dos empleados en BBDD");
    assertEquals(nuevoNombre, service.getById( e2.getId() ).getNombre(),
                "Ha sido modificado el nombre del empleado con id: "+e2.getId());
}
```

11. Definición de test unitarios

Test unitario para el controller: (EmpleadoRestController)

```
@Test
void testEliminarEmpleado() {
    //GIVEN:
    //Hay dos usuarios en la BBDD:
    assertEquals(2, service.listar().size(), "Hay dos empleados en BBDD");

    //WHEN:
    controller.eliminarEmpleado(e2.getId());

    //THEN:
    assertEquals(1, service.listar().size(), "Solo queda un empleado en BBDD");
}
```

11. Definición de test unitarios

Test unitario para el controller: (EmpleadoRestController)

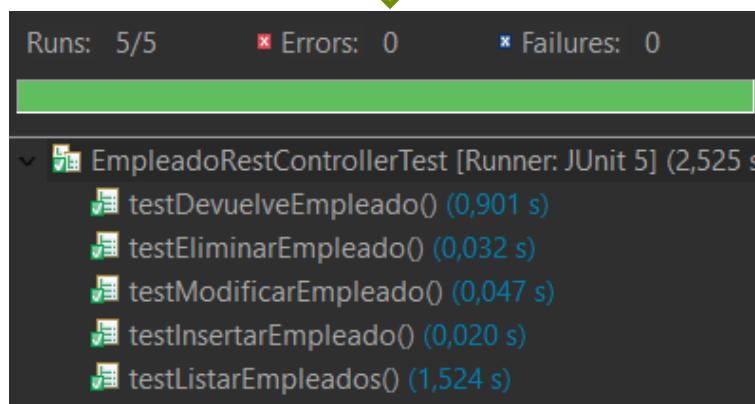
```
@Test
void testDevuelveEmpleado() {
    //GIVEN:
    //Hay dos usuarios en la BBDD:

    //WHEN:
    ResponseEntity<Empleado> re = controller.devuelveEmpleado( e2.getId() );

    //THEN:
    assertNotNull(re.getBody(), "Encontrado empleado con id: "+e2.getId());
}
```



Runs: 5/5 ✘ Errors: 0 ✘ Failures: 0

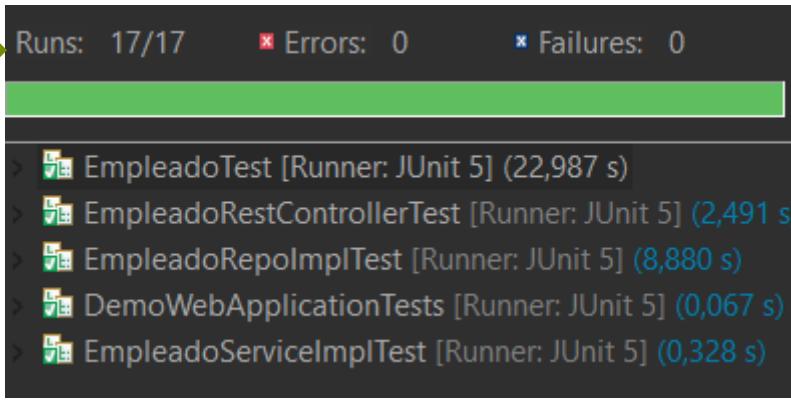


EmpleadoRestControllerTest [Runner: JUnit 5] (2,525 s)

- testDevuelveEmpleado() (0,901 s)
- testEliminarEmpleado() (0,032 s)
- testModificarEmpleado() (0,047 s)
- testInsertarEmpleado() (0,020 s)
- testListarEmpleados() (1,524 s)



Runs: 17/17 ✘ Errors: 0 ✘ Failures: 0



EmpleadoTest [Runner: JUnit 5] (22,987 s)

EmpleadoRestControllerTest [Runner: JUnit 5] (2,491 s)

EmpleadoRepoImplTest [Runner: JUnit 5] (8,880 s)

DemoWebApplicationTests [Runner: JUnit 5] (0,067 s)

EmpleadoServiceImplTest [Runner: JUnit 5] (0,328 s)

11. Definición de test unitarios

Test unitario para el controller: (EmpleadoRestController)

Debemos probar todos los caminos, pero ¿qué sucede cuando se produce una excepción?

```
@CacheEvict(value="empleados", allEntries=true)
@PostMapping
public ResponseEntity<List<Empleado>> insertarEmpleado_v3 (@RequestBody Empleado empleado) {
    try {
        HttpHeaders headers = new HttpHeaders();
        if (empleado.getId()!=null) {
            headers.set("Message", "Para dar de alta un nuevo empleado, el ID debe llegar vacío");
            return new ResponseEntity<>(headers, HttpStatus.NOT_ACCEPTABLE);
        }
        else if (empleado.getNombre()==null || empleado.getNombre().equals(""))
            || empleado.getApellidos()==null || empleado.getApellidos().equals("")) {
            headers.set("Message", "Ni NOMBRE ni APELLIDOS pueden ser nulos");
            return new ResponseEntity<>(headers, HttpStatus.NOT_ACCEPTABLE);
        }

        Empleado emp = empleadoService.insertar(empleado);
        URI newPath = new URI("/api/empleados/" + emp.getId());
        headers.setLocation(newPath);
        headers.set("Message", "Cliente insertado correctamente con id: " + emp.getId());
        return new ResponseEntity<> (headers, HttpStatus.CREATED);
    }
    catch (Exception ex) {
        return new ResponseEntity<> (new HttpHeaders(), HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

11. Definición de test unitarios

Test unitario para el controller: (EmpleadoRestController)

Vamos a utilizar a Mockito:

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

```
@TestPropertySource(locations = "classpath:application-test.properties")
@SpringBootTest
class EmpleadoRestControllerTest {

    private Empleado e1, e2;

    @Autowired
    EmpleadoRestController controller;

    @Autowired
    EmpleadoService service;

    @Autowired
    EmpleadoRepoJPA repo;

    @InjectMocks
    EmpleadoRestController restControllerMock; →

    @Mock
    EmpleadoService serviceMock; →
}
```

11. Definición de test unitarios

Test unitario para el controller: (EmpleadoRestController)

Primero indicaremos que el método ‘insertar’ del servicio puede devolver una Excepción:

```
@Override  
public Empleado insertar(Empleado empleado) throws Exception {  
    return empleadoRepo.save(empleado);  
}
```

```
public interface EmpleadoService {  
    void registrar (String name);  
    List<Empleado> listar();  
    public List<Empleado> listarFiltroNombre(String cad);  
    List<Empleado> listarConJPA (Integer pID, String contiene);  
    Empleado insertar(Empleado empleado) throws Exception; ←  
    void modificar(Empleado empleado);  
    void eliminarEmpleado(Integer id);  
    Empleado getById(Integer id);  
}
```

```
@Test  
void testInsertar() throws Exception { ←
```

```
@CacheEvict(value="empleados", allEntries=true)  
//@PostMapping  
public void insertarEmpleado (@RequestBody Empleado empleado) throws Exception {  
    empleado.setId(null);  
    empleadoService.insertar(empleado);  
}
```

11. Definición de test unitarios

Test unitario para el controller: (EmpleadoRestController)

```
@Test
void testInsertarEmpleado_v3Exception () throws Exception {
    //GIVEN: // El servicio devolverá una excepción:
    when(serviceMock.insertar(e1)).thenThrow(new Exception ());

    //WHEN:
    ResponseEntity<List<Empleado>> re = restControllerMock.insertarEmpleado_v3(e1);

    //THEN:
    assertThat( re.getStatusCodeValue() ).isEqualTo(406);
    //assertThat( re.getStatusCodeValue() ).isEqualTo(HttpStatus.INTERNAL_SERVER_ERROR.value());
    //assertThat( re.getBody().size() ).isZero();
}
```



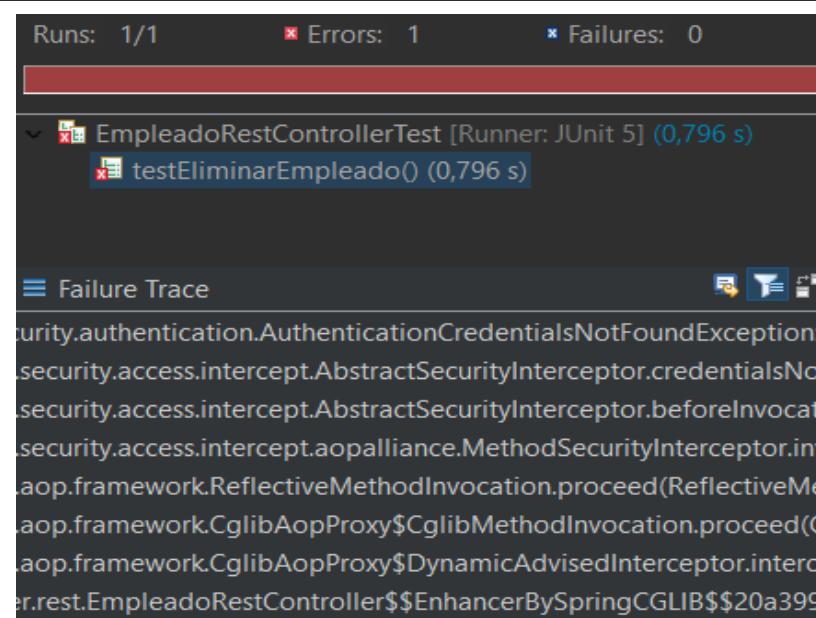
Runs: 1/1	Errors: 0	Failures: 0
<hr/>		
▼	EmpleadoRestControllerTest [Runner: JUnit 5] (1,044 s)	
▼	testInsertarEmpleado_v3Exception() (1,044 s)	

11. Definición de test unitarios

Test unitario para el controller: (EmpleadoRestController)

¿Qué sucede si para el método está restringido con **Spring Security** y sólo puede acceder un rol?

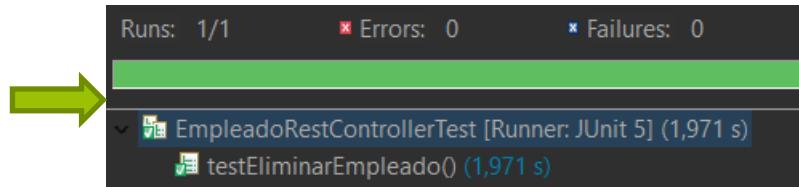
```
→ @PreAuthorize("hasRole('ROLE_ADMIN')")
  @CacheEvict(value="empleados", allEntries=true)
  @DeleteMapping (value="/{id}")
  public void eliminarEmpleado (@PathVariable("id") Integer id) {
    empleadoService.eliminarEmpleado(id);
}
```



11. Definición de test unitarios

Test unitario para el controller: (EmpleadoRestController)

```
→ @Test  
@WithMockUser(username = "user1", roles={"ADMIN"})  
void testEliminarEmpleado() {  
    //GIVEN:  
    //Hay dos usuarios en la BBDD:  
    assertEquals(2, service.listar().size(), "Hay dos empleados en BBDD");  
  
    //WHEN:  
    controller.eliminarEmpleado(e2.getId());  
  
    //THEN:  
    assertEquals(1, service.listar().size(), "Solo queda un empleado en BBDD");  
}
```





NTT DATA