



Spring Boot: Creación de un app web

Orientado a profesionales que están empezando en el desarrollo web con Spring Boot

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)
8. Segurizar con Spring Security
9. Definición de test unitarios
10. Actividad final



01 Creación de 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

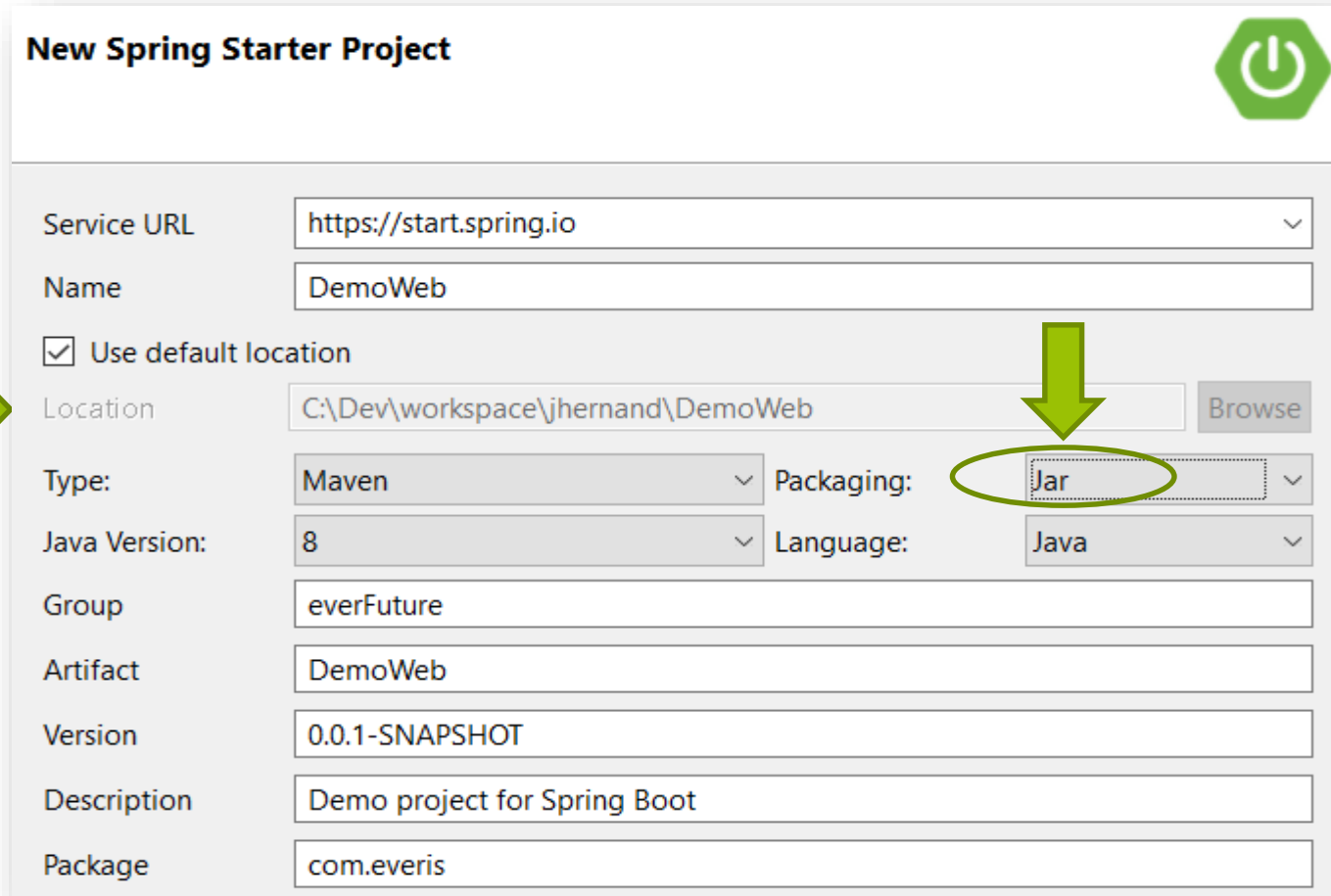
1. Creación de app web simple

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:

Name:

☒ Use default location

Location:

Type: Packaging:

Java Version: Language:

Group:

Artifact:

Version:

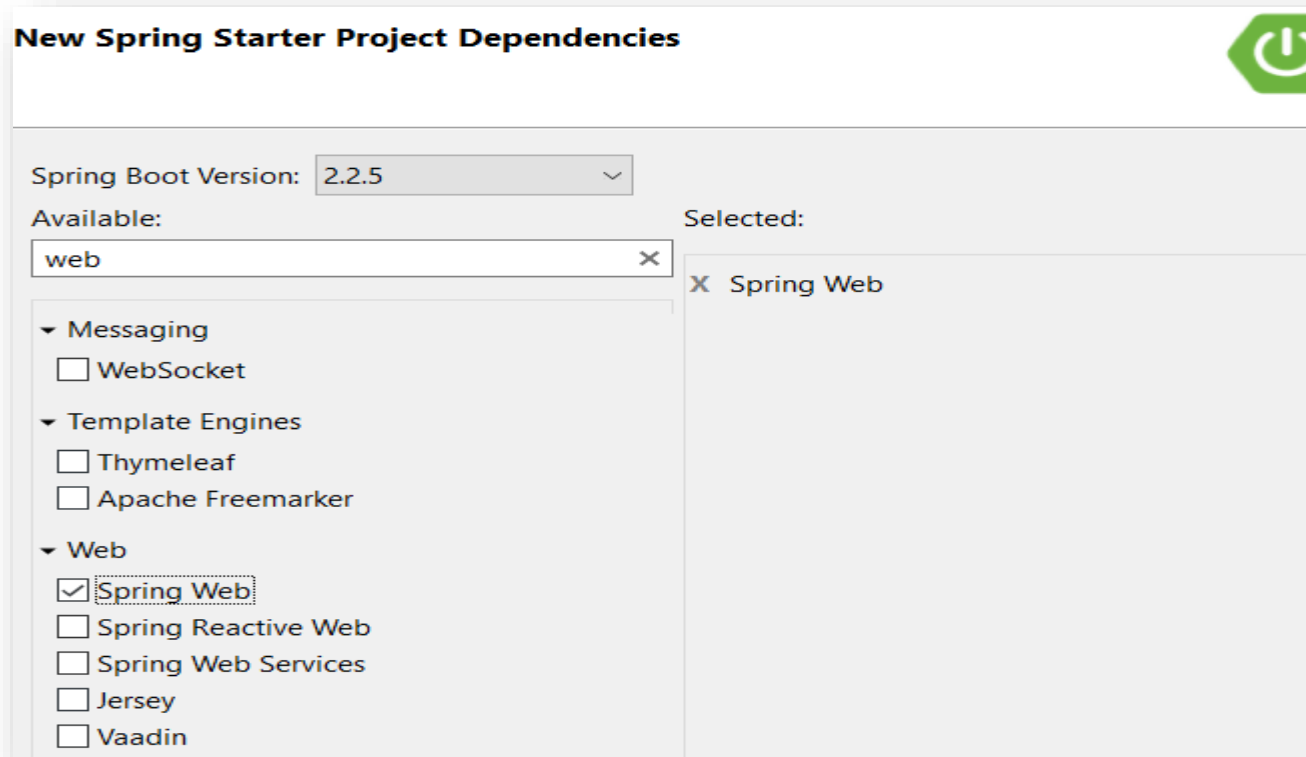
Description:

Package:

1. Creación de app web simple

Creación de app web

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

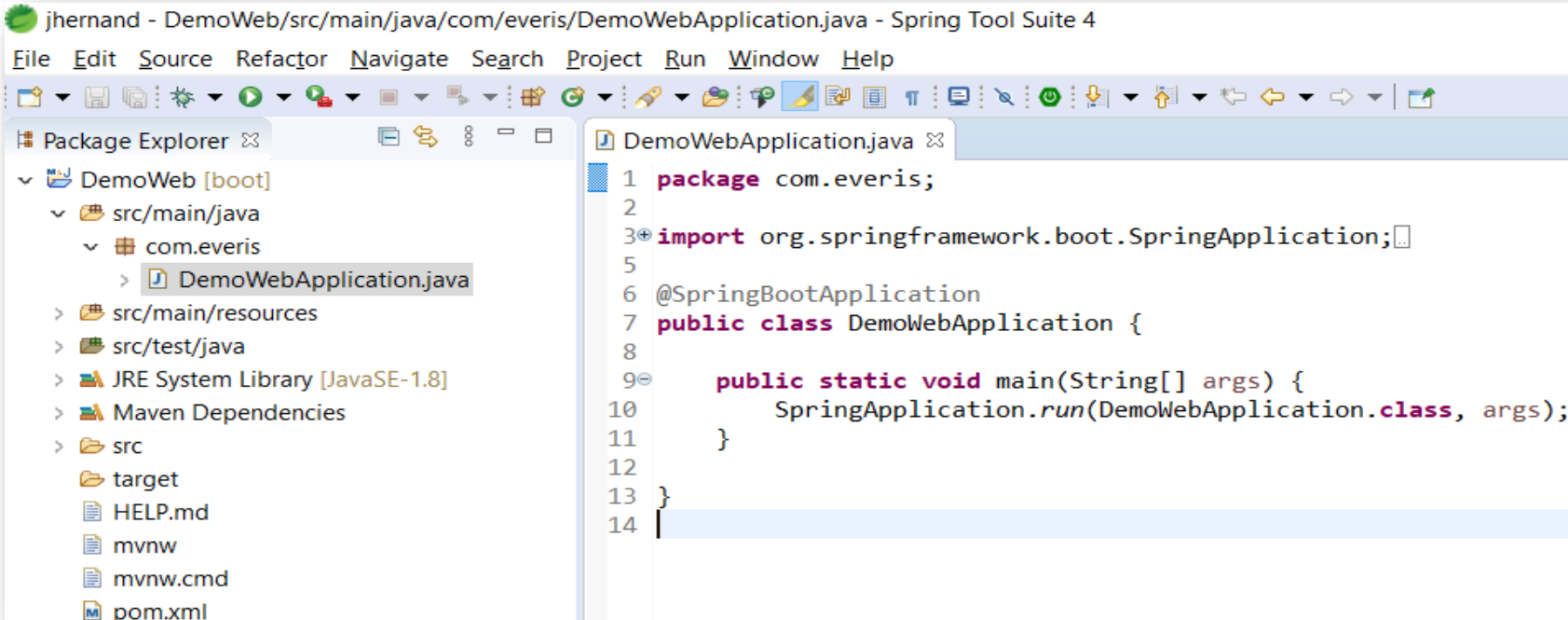


Y tras seleccionarla pulsaremos **'Finish'** para que nos cree el proyecto.

1. Creación de app web simple

Creación de app web

Revisamos el método 'main' creado:



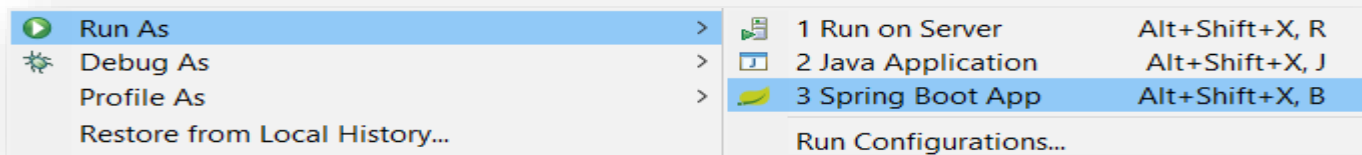
```

jhernand - DemoWeb/src/main/java/com/everis/DemoWebApplication.java - Spring Tool Suite 4
File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer
DemoWeb [boot]
├── 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
└── pom.xml

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

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





```

      \ / _ _ _ \      ( )      \ \ \ \ \
( ) \ _ _ _ \ _ _ _ \ _ _ _ \ _ _ _ \
  \ _ _ _ \ | | | | | | | | | | | | | |
    \ _ _ _ \ | | | | | | | | | | | | | |
      \ _ _ _ \ _ _ _ \ _ _ _ \ _ _ _ \
=====|_|=====|_|/_/_/_/_/_/_/_/_/_/_
:: Spring Boot ::      (v2.5.5.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

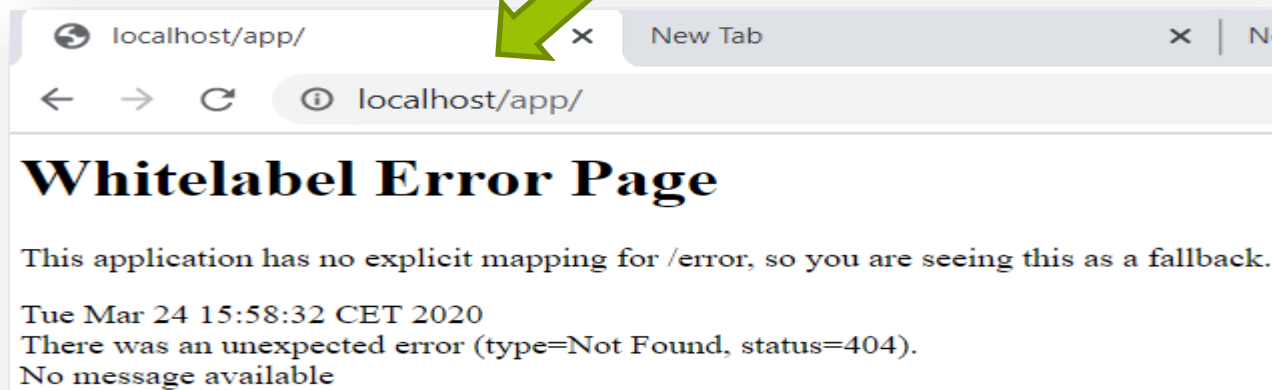
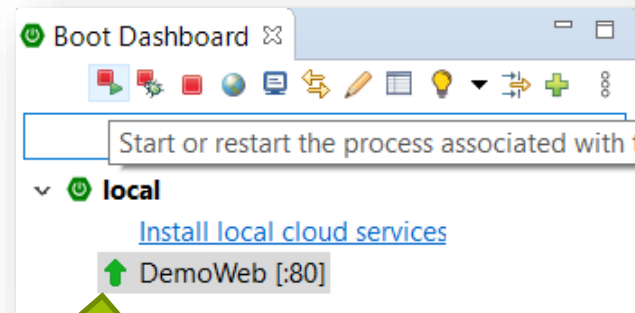
1. Creación de app web simple

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:



1. Creación de app web simple

Creación de app web

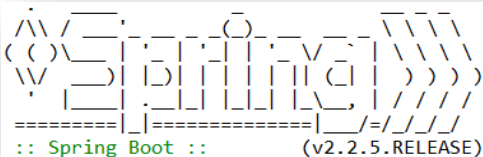
Pero, ¿a cada cambio hay que estar parando y arrancando el servidor?
Si no queremos no.

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 redespiega 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 20.463s)
2020-03-24 15:41:36.135 INFO 11056 --- [ restartedMain] .ConditionEvaluationDeltaLoggingListener : Condition evaluation unchanged
```



ever
FUTURE



2

Incorporando Spring MVC + Thymeleaf

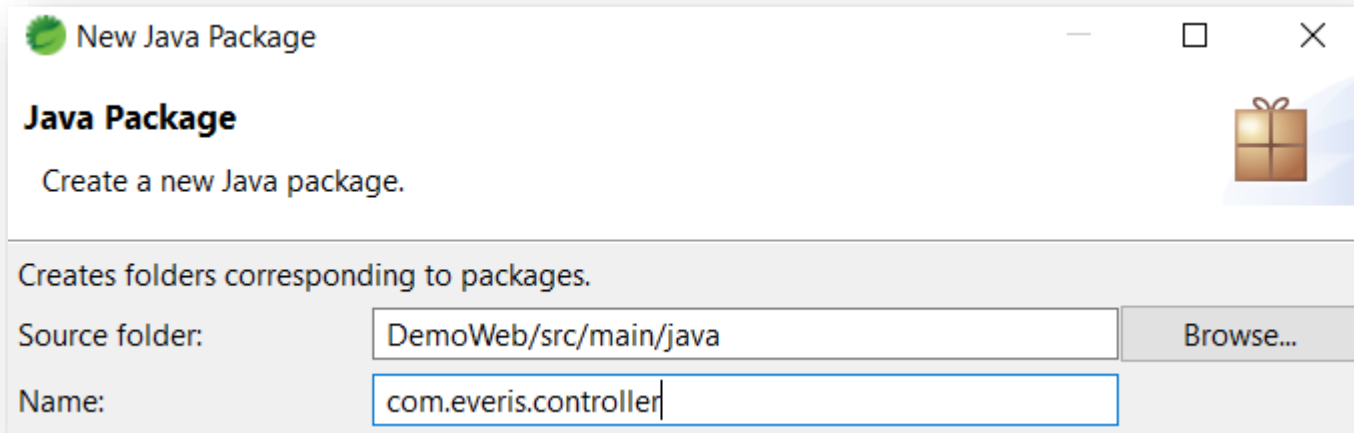
2. Incorporando Spring MVC + Thymeleaf

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**

Create a new Java class.

Source folder:	DemoWeb/src/main/java
Package:	com.everis.controller
<input type="checkbox"/> Enclosing type:	
Name:	DemoController

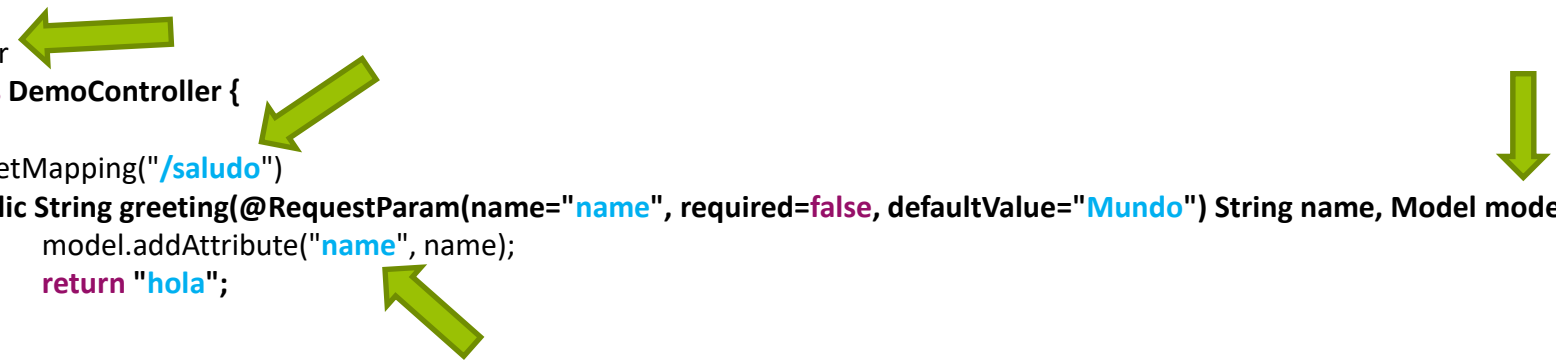
Ahora escribimos el siguiente código en nuestra clase **DemoController.java**:

```
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";
    }
}
```



2. Incorporando Spring MVC + Thymeleaf

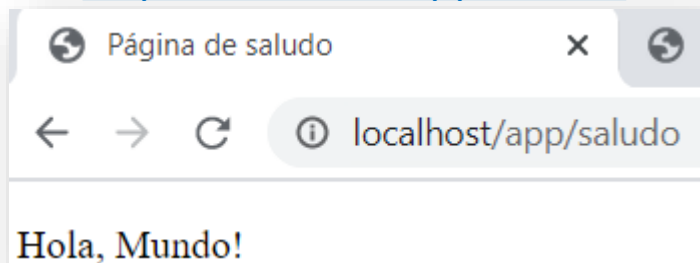
Thymeleaf

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

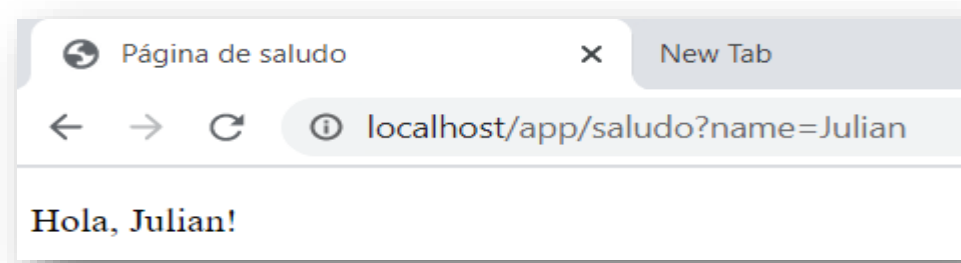
```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
  <title>Página de saludo</title>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
<body>
  <p th:text="'Hola, ' + ${name} + '!'" />
</body>
</html>
```

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.

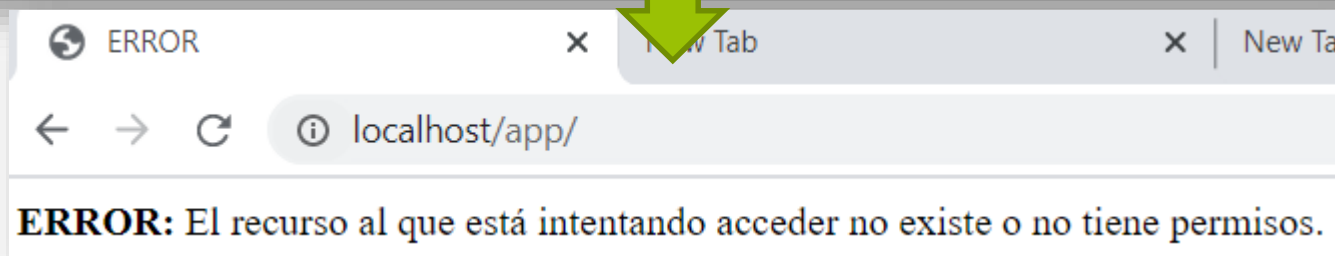
Para ello:

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>
```



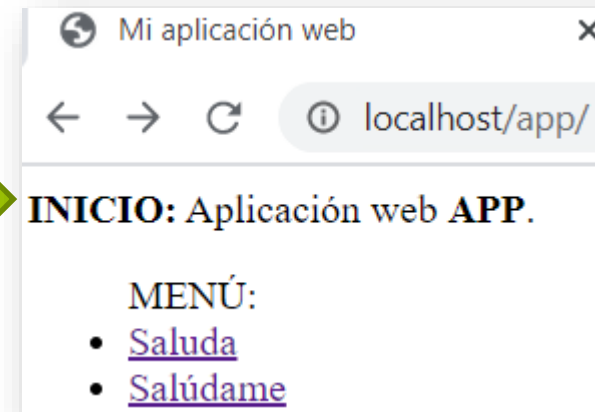
2. Incorporando Spring MVC + Thymeleaf

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>
```





03 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

DemoControlller:

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

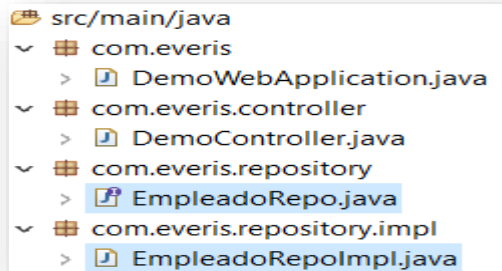


```
INFO 22332 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 80 (http) with context path '/app'
INFO 22332 --- [ restartedMain] com.everis.DemoWebApplication : Started DemoWebApplication in 0.33 seconds (JVM running for 7402.001)
INFO 22332 --- [ restartedMain] .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:



```
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 {

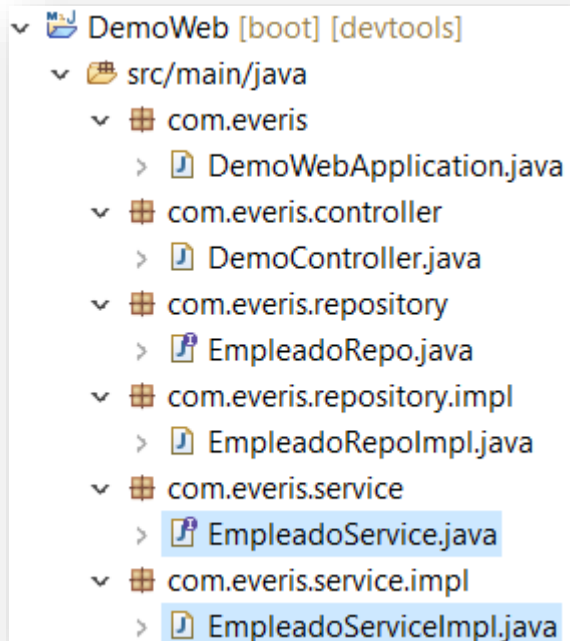
    private static Logger LOG = org.slf4j.LoggerFactory.getLogger(DemoWebApplication.class);

    @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
```




ever
FUTURE



04 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 añadirsele 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 faltarían 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:

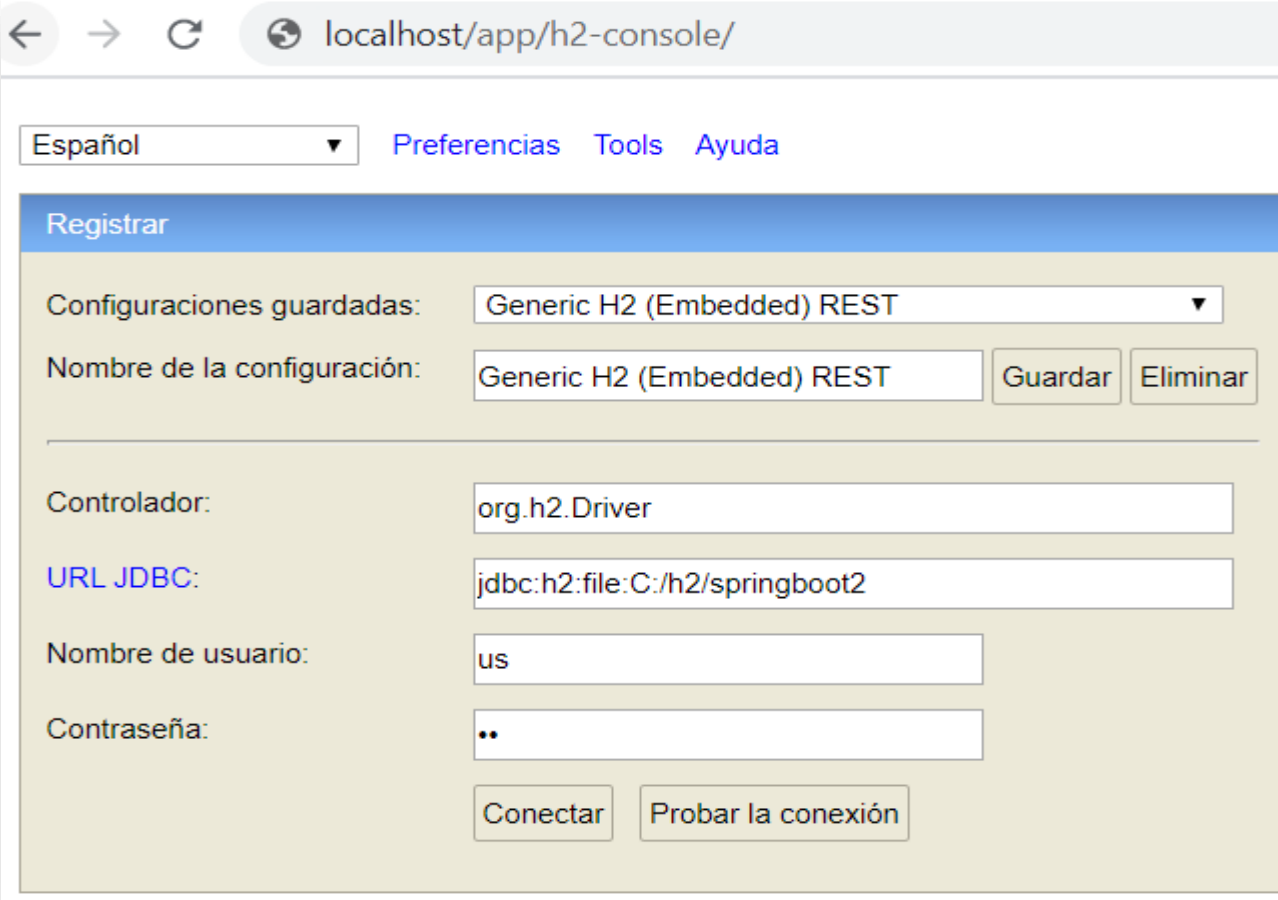
```
INFO 3364 --- [ restartedMain] o.s.web.context.ContextLoader      : Root WebApplicationContext: initialization completed in 1281 ms
INFO 3364 --- [ restartedMain] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
INFO 3364 --- [ restartedMain] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - 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.
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
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

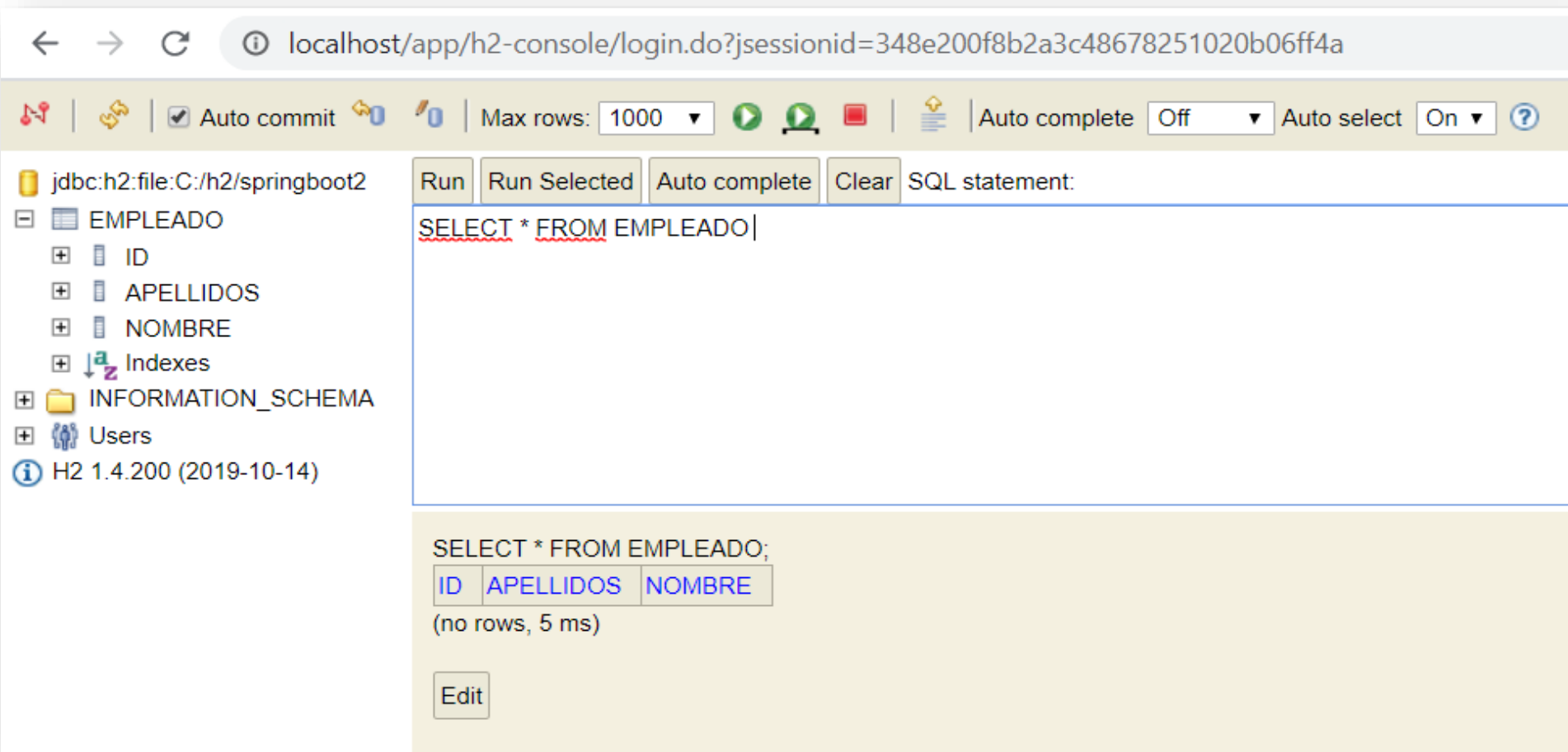
Contraseña: ..

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 console web interface in a browser. The address bar displays the URL: `localhost/app/h2-console/login.do?jsessionId=348e200f8b2a3c48678251020b06ff4a`. The interface includes a toolbar with options like 'Auto commit', 'Max rows' (set to 1000), 'Auto complete' (set to Off), and 'Auto select' (set to On). On the left, a tree view shows the database structure: 'jdbc:h2:file:C:/h2/springboot2' containing 'EMPLEADO' (with columns ID, APELLIDOS, NOMBRE and an Indexes), 'INFORMATION_SCHEMA', 'Users', and 'H2 1.4.200 (2019-10-14)'. The main area shows the SQL statement `SELECT * FROM EMPLEADO;` entered and executed. Below the statement, the results are displayed as a table with columns 'ID', 'APELLIDOS', and 'NOMBRE', and a message '(no rows, 5 ms)'. An 'Edit' button is located at the bottom.

← → ↻ ⓘ localhost/app/h2-console/login.do?jsessionId=348e200f8b2a3c48678251020b06ff4a

🔗 📄 ☒ Auto commit 🔄 📄 | Max rows: 1000 ▶ ▶ ▶ | 📄 | Auto complete Off ▶ Auto select On ▶ ⓘ

📄 jdbc:h2:file:C:/h2/springboot2

- 📄 EMPLEADO
 - + 📄 ID
 - + 📄 APELLIDOS
 - + 📄 NOMBRE
 - + 📄 Indexes
- + 📄 INFORMATION_SCHEMA
- + 📄 Users
- 📘 H2 1.4.200 (2019-10-14)

Run Run Selected Auto complete Clear SQL statement:

SELECT * FROM EMPLEADO|

SELECT * FROM EMPLEADO;

ID	APELLIDOS	NOMBRE
----	-----------	--------

(no rows, 5 ms)

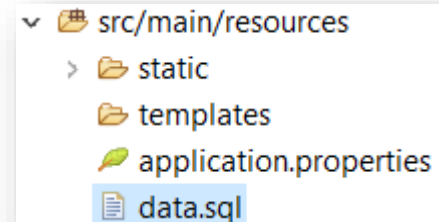
Edit

4. Conectando con BBDD

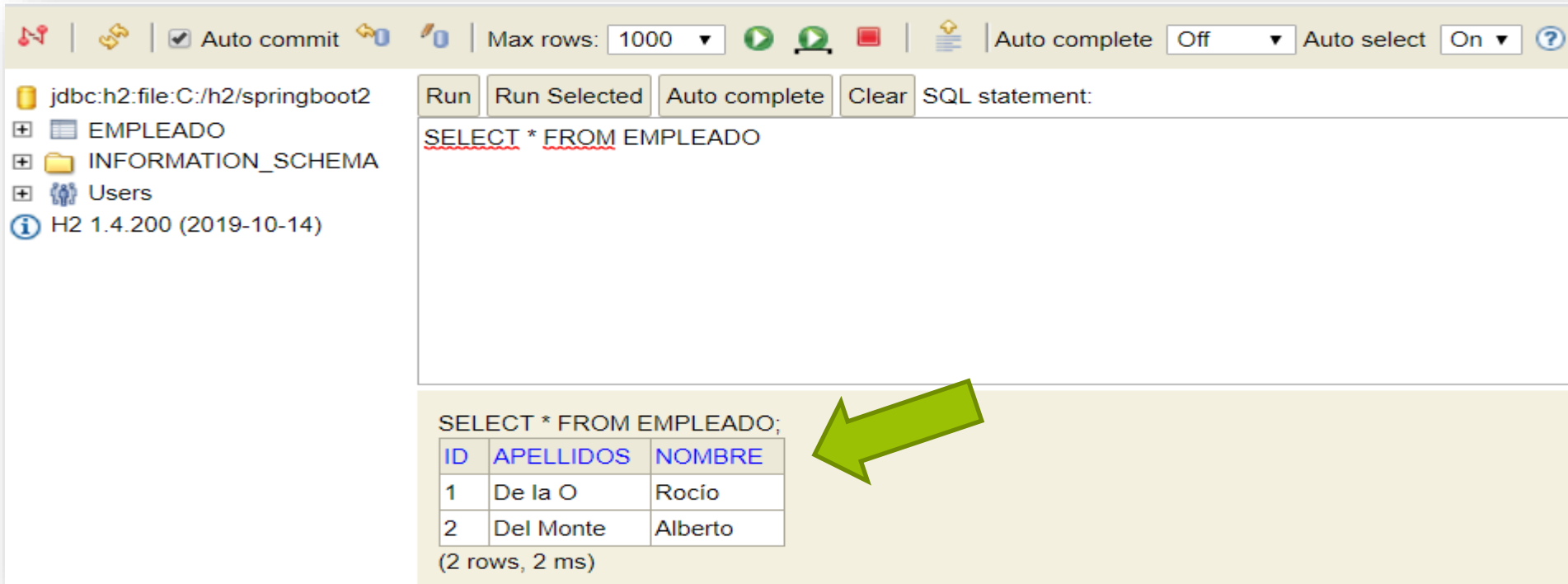
Ahora creamos el fichero `/src/main/resources/data.sql` añadimos las siguientes líneas:

```
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:



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

SELECT * FROM EMPLEADO;

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

(2 rows, 2 ms)



05 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`:

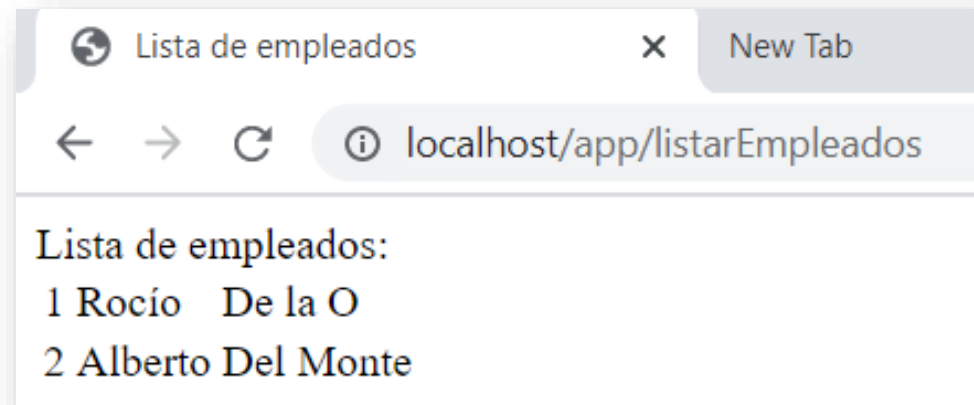
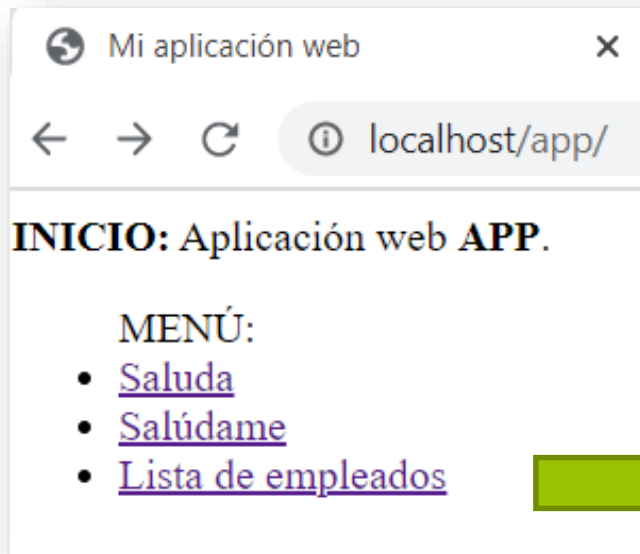
```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Mi aplicación web</title>
</head>
<body>
  <b>INICIO:</b> Aplicación web <b>APP</b>.
  <ul> MENÚ:
    <li> <a href="/app/saludo">Saluda</a> </li>
    <li> <a href="/app/saludo?name=Julian">Salúdame</a> </li>
    <li> <a href="/app/listarEmpleados">Lista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

Nota: separar espacios de ... th:

5. Spring Data JPA

Al acceder ahora a **http://localhost/app** obtendremos:



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:

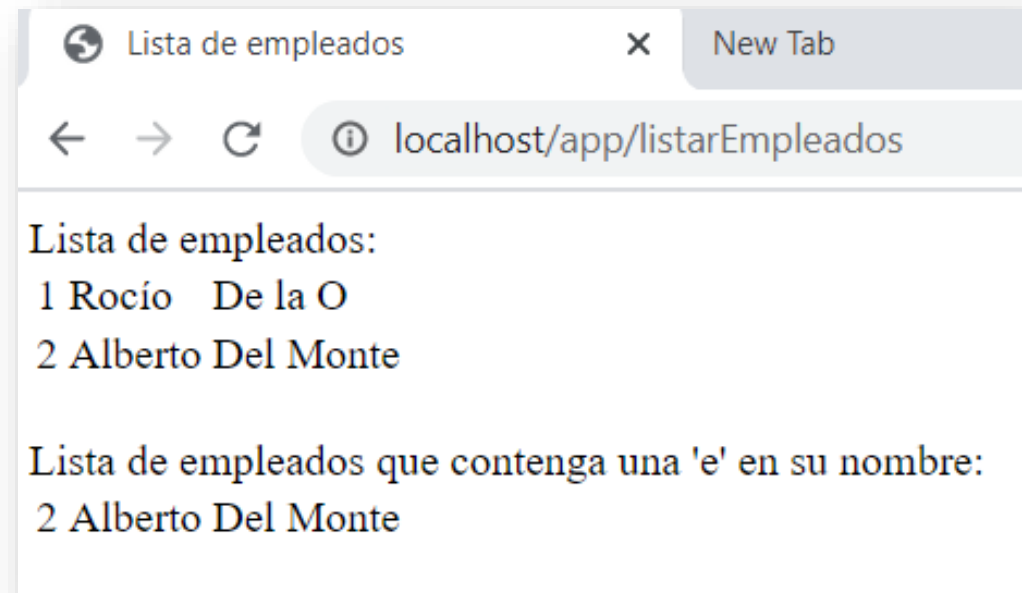
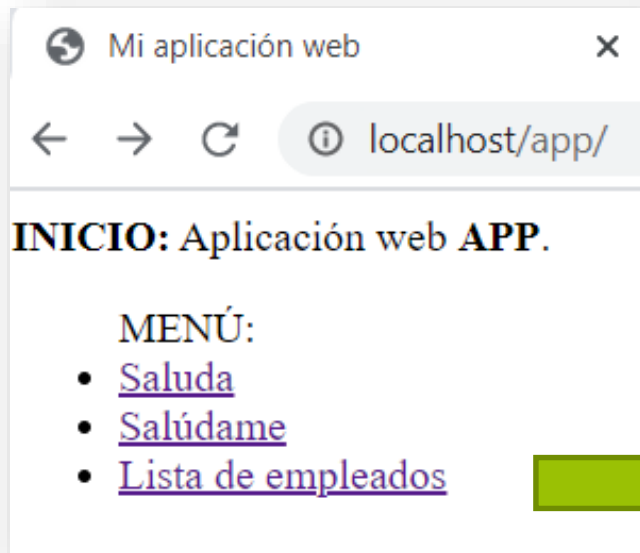
```
@GetMapping("/listarEmpleados")
public String listarEmp(Model model) {
    model.addAttribute("listaEmp", empleadoService.listar() );
    model.addAttribute("listaEmpConE", empleadoService.listarFiltroNombre("e") );
    return "listarDeEmpleados";
}
```

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:



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 **findByIdGreaterThanAndNombreLike**:


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

    List<Empleado> findByIdGreaterThanAndNombreLike (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.findByIdGreaterThanAndNombreLike(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 (id>2 & contenga 'o' en el nombre):

- 4 Roberto Sánchez



06 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 a 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>



```
[
  - {
    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



07 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;

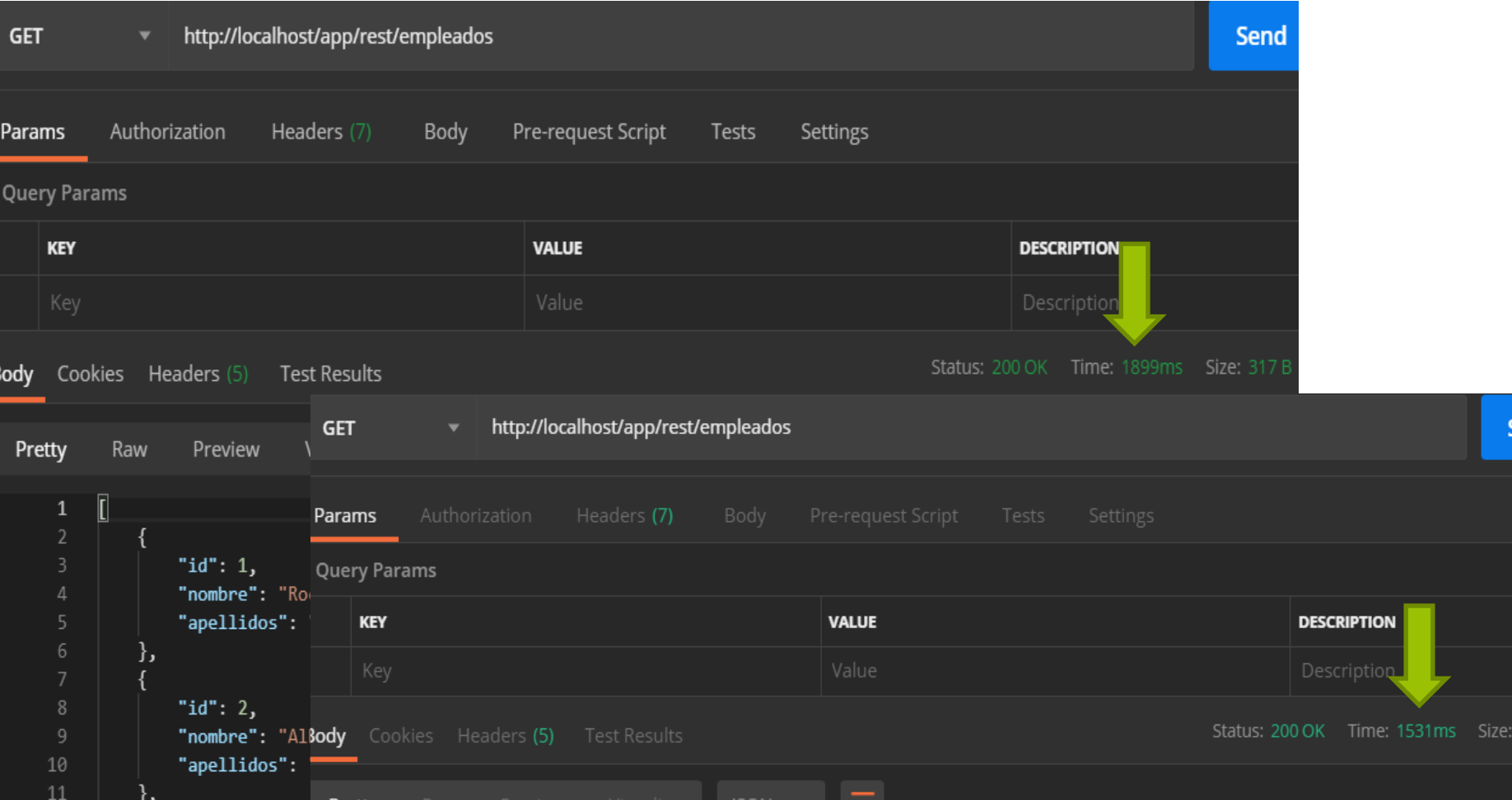
@RestController
@RequestMapping ("/rest/empleados")
public class EmpleadosRestController {

    @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:



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

Body Cookies Headers (5) Test Results Status: 200 OK Time: 1899ms Size: 317 B

Pretty Raw Preview GET ▼ http://localhost/app/rest/empleados Send

```

1  [
2    {
3      "id": 1,
4      "nombre": "Rocio",
5      "apellidos": "Garcia",
6    },
7    {
8      "id": 2,
9      "nombre": "Alfonso",
10     "apellidos": "Garcia",
11   }
12 ]
  
```

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


Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body Cookies Headers (5) Test Results Status: 200 OK Time: 1531ms Size: 317 B

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:

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

Body Cookies Headers (5) Test Results Status: 200 OK Time: 1899ms Size: 317 B

Pretty Raw GET ▼ http://localhost/app/rest/empleados Send

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

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

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body Cookies Headers (5) Test Results Status: 200 OK Time: 9ms Size: 266 B

Pretty Raw Preview Visualize JSON ≡

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;

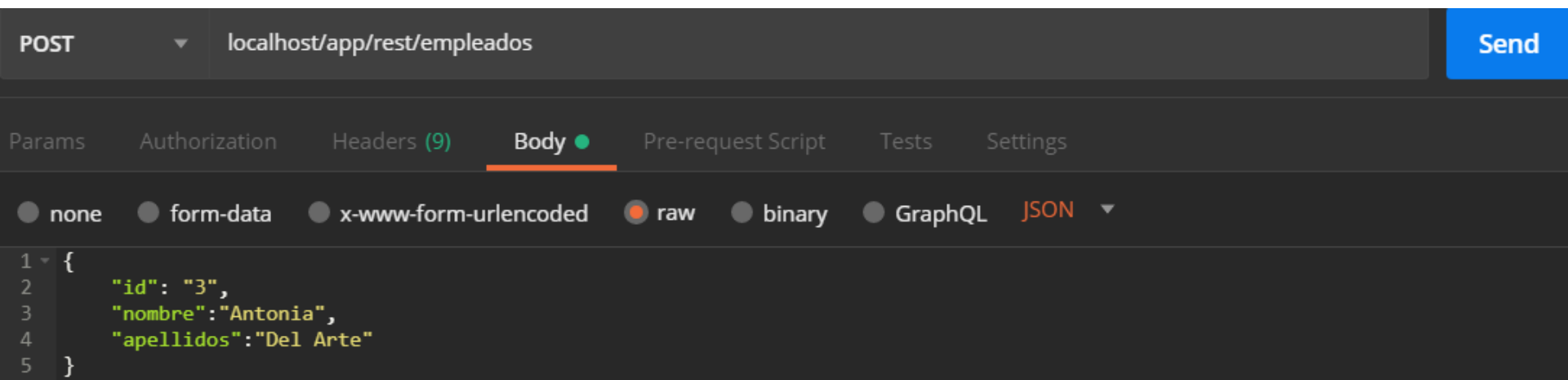
    @PostMapping
    @CacheEvict(value="empleados", allEntries = true)
    public void insertarEmpleado(@RequestBody Empleado emp) {
        empleadoService.inserta(emp);
    }
}
```

A large green arrow points from the left side of the slide towards the `@CacheEvict` annotation in the code block, highlighting the cache eviction configuration.



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:



7. Manejo de caché

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

GET Send

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

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

body Cookies Headers (5) Test Results Status: 200 OK Time: 1511ms Size: 317 B

GET

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

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

body Cookies Headers (5) Test Results Status: 200 OK Time: 22ms

Pretty Raw Preview Visualize JSON

```

1
2
3 {
4   "id": 1,
5   "nombre": "Rocío",
6   "apellidos": "De la O"
7 }
8
9 {
10  "id": 2,
11  "nombre": "Alberto",
12  "apellidos": "Del Monte"
13 }
14
15 {
16  "id": 3,
17  "nombre": "Antonia",
18  "apellidos": "Del Arte"
19 }
20
  
```

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";
}
```



08 Segurizar con Spring Security

8. Segurizar con Spring Security

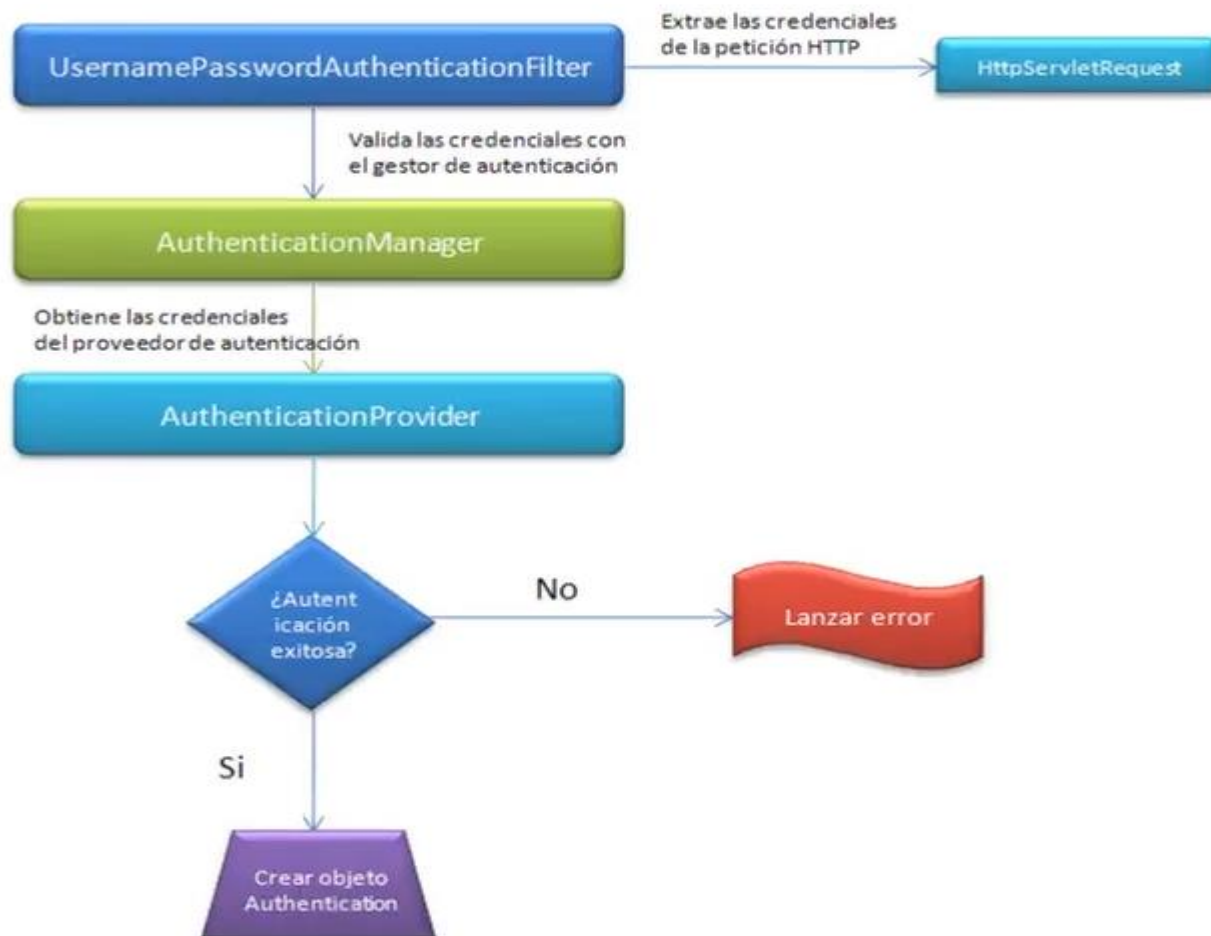
Spring Security es un módulo de Spring que nos permite manejar autenticaciones.

¿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:

Mi aplicación web

localhost/app/

INICIO: Aplicación web APP.

MENÚ:

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

Elements Console Sources Network Performance Memory Application Security Audits

Filter ☐ Hide data URLs **All** XHR JS CSS Img Media Font Doc WS Manifest Other

☐ Only show requests with SameSite issues

10 ms 20 ms 30 ms 40 ms 50 ms 60 ms 70 ms 80 ms 90

Name app/

1 requests | 581 B trans

Headers Preview Response Initiator Timing

▼ **Response Headers** view source

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

▼ **Request Headers** view source

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

8. Segurizar con Spring Security

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

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

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

Headers
7 hidden

KEY	VALUE	DESCRIPTION	...	Bulk Edit	Presets
Key	Value	Description			

Body
Cookies
Headers (5)
Test Results
Status: 200 OK Time: 37ms Size: 317 B Save Response

KEY	VALUE
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.ThreadPoolTaskExecutor : 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:

A screenshot of a web browser window. The address bar shows 'localhost/app/login'. The page title is 'Please sign in'. The form has two input fields: 'Username' and 'Password'. Below them is a blue 'Sign in' button.

Please sign in

Credenciales erróneas

us

..

Sign in

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)

A screenshot of a web browser window. The address bar shows 'localhost/app/'. The page title is 'Mi aplicación web'. The content shows 'INICIO: Aplicación web APP.' followed by a 'MENÚ:' section with three links: 'Saluda', 'Salúdame', and 'Lista de empleados'.

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 la 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.b0Os4hAWblEkry', 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.b0Os4hAWblEkry', 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}]]),
    tu rol es [[${usuario.rol.rol}]]. </h5>

    <b>INICIO:</b> Aplicación web <b>APP</b>.

    <ul> MENÚ:
```

Mi aplicación web x +

localhost/app/

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


INICIO: Aplicación web APP.

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**:



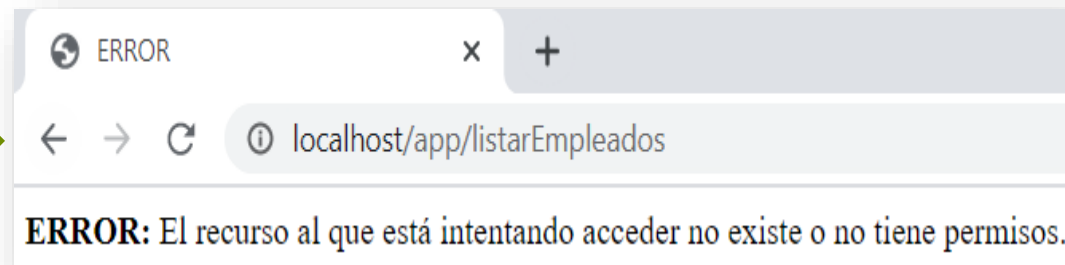
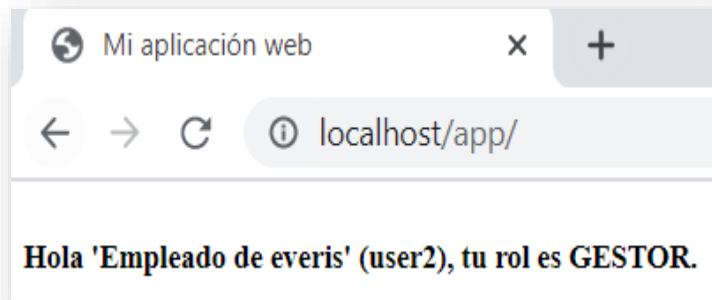
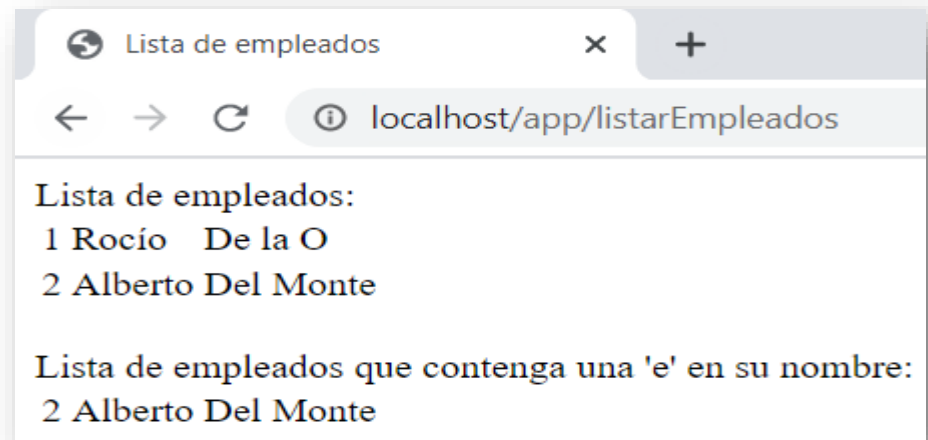
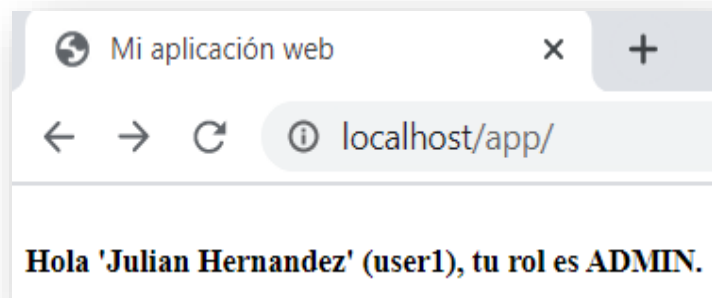
```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class WebSecurityConfiguration extends WebSecurityConfigurerAdapter {
```



```
@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):



8. Segurizar con Spring Security

Parar 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:

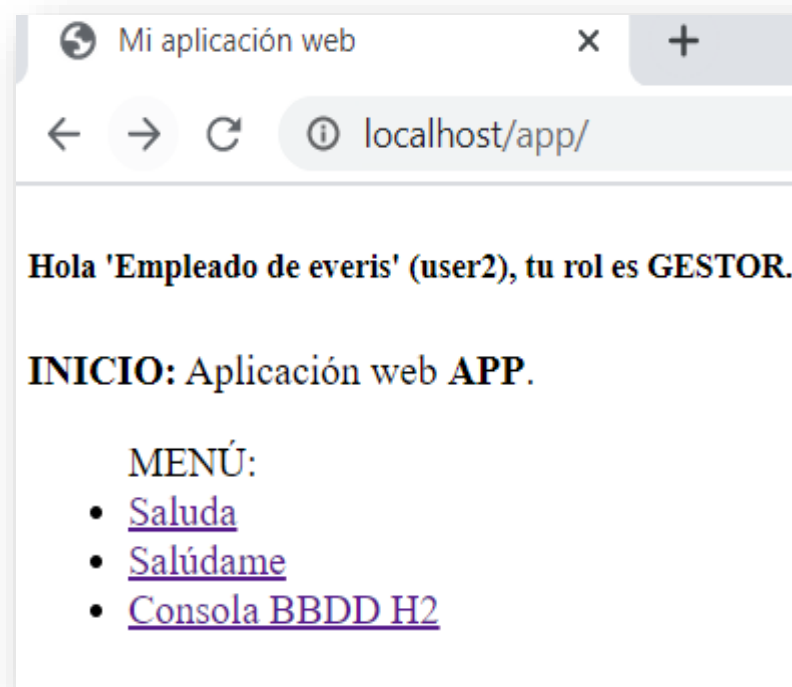
```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
  <meta charset="UTF-8">
  <title>Mi aplicación web</title>
</head>
<body>
  <h5 th:inline="text">Hola '[[${usuario.nombreYapellidos}]]' ([[${usuario.username}]]),
  tu rol es [[${usuario.rol.rol}]]. </h5>

  <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:





09 Definición de test unitarios

9. 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.

9. 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.datasource.url=jdbc:h2:file:C:/h2/springboot2_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
```

9. 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);

insert into rol (id, rol)
    select 9999, 'ENTORNO TEST' from dual where not exists (select 1 from rol where id = 3);

/* 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 everis',
'$2a$10$5xOe75pbLcAjp0TbVWaluunrSshgYdH82YNwGd.b0Os4hAWbIEkry', 2 from dual where not
exists (select 1 from usuario where username = 'user2');
```

9. Definición de test unitarios

Primero vamos a crear una clase **CheckearEntornoTest** dentro de **src/test/java** muy simple que se va a encargar únicamente de validar que estamos ejecutando contra la BBDD correcta (springboot2_test) haciendo un pequeño apaño que nos sirva como ejemplo:

```
package com.everis;

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

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

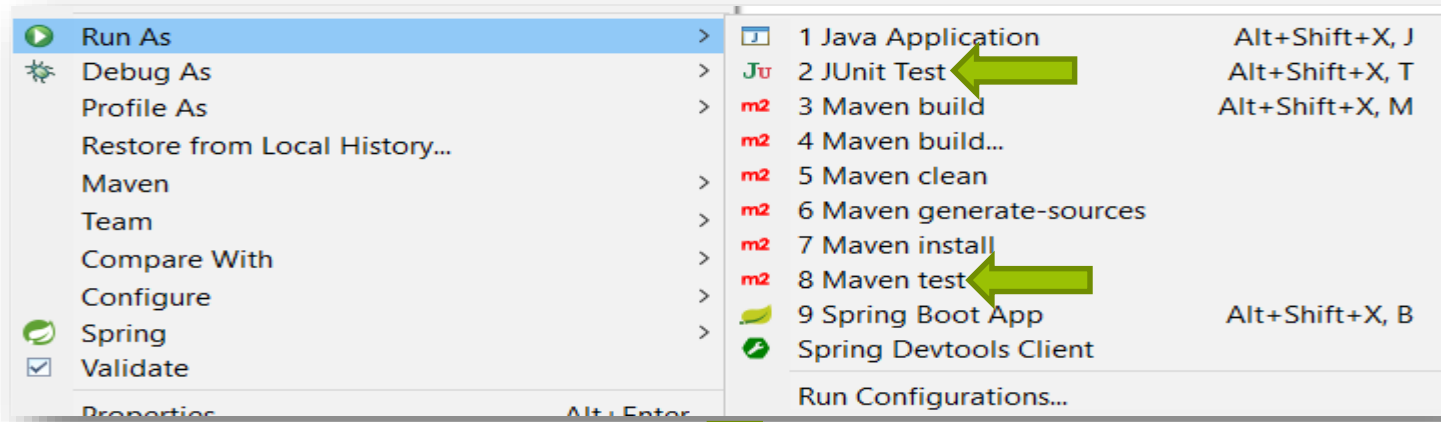
    @Autowired
    RolRepoJPA rolDAO;

    @Test
    void validarEntorno() {
        List<Rol> lr = rolDAO.findAll();
        LOG.info("=====");
        LOG.info("=====> " + lr.get(lr.size()-1).getRol() + " <=====");
        LOG.info("=====");

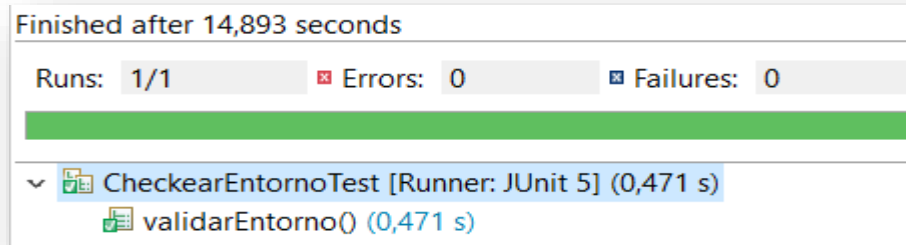
        assertTrue ( lr.get(lr.size()-1).getRol().equals("ENTORNO TEST") );
    }
}
```

9. Definición de test unitarios

Podremos ejecutar nuestro test de dos formas sencillas. Botón derecho sobre el nombre de nuestro proyecto, y luego en la opción '**Run As**' escogemos o '**JUnit Test**' o '**Maven test**':



```
[ main] com.everis.CheckearEntornoTest : =====> ENTORNO TEST <=====
[ main] com.everis.CheckearEntornoTest : =====> ENTORNO TEST <=====
[ main] com.everis.CheckearEntornoTest : =====> ENTORNO TEST <=====
```



9. Definición de test unitarios

Ahora vamos a crear un test unitario más válido, que se encargue de testear el método `'listarCuyoNombreContiene'` que implementamos en la clase `'EmpleadoRepoImpl'`. Para ello vamos a crearnos la clase `'EmpleadoRepoImplTest'` en `src/test/java`:

```
package com.everis.repository;

import org.junit.jupiter.api.Test;
import org.slf4j.Logger;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.TestPropertySource;

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

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

    @Autowired
    EmpleadoRepoJPA empleadoDAO;
}
```



9. Definición de test unitarios

Dentro vamos a definir el método **testListarCuyoNombreContiene** de la siguiente forma:

```
@Test
void testListarCuyoNombreContiene () {
    empleadoDAO.deleteAll();

    Empleado e1 = new Empleado(1001, "Abcdws", "A"); //Constructor con parámetros
    Empleado e2 = new Empleado(1002, "Abcdws", "A");
    Empleado e3 = new Empleado(1003, "Abcds", "Awa");
    Empleado e4 = new Empleado(1004, "Abcds", "AWa");
    Empleado e5 = new Empleado(1005, "aaa", "aaa");
    Empleado e6 = new Empleado(1006, "www", "aaa");

    empleadoDAO.save(e1);
    empleadoDAO.save(e2);
    empleadoDAO.save(e3);
    empleadoDAO.save(e4);
    empleadoDAO.save(e5);
    empleadoDAO.save(e6);

    List<Empleado> le = empleadoDAO.findAll();
    le = empleadoDAO.listarCuyoNombreContiene("w");
    LOG.info(" ==> testListarCuyoNombreContiene: 'w' = " +le.size() );

    empleadoDAO.deleteAll();

    assertTrue(le.size() == 3);
}
```

9. Definición de test unitarios

Dentro vamos a definir el método **testListarCuyoNombreContiene** de la siguiente forma:

```
@Test
void testListarCuyoNombreContiene () {
    empleadoDAO.deleteAll();

    Empleado e1 = new Empleado(1001, "AbcdWs", "A"); //Constructor con parámetros
    Empleado e2 = new Empleado(1002, "Abcdws", "A");
    Empleado e3 = new Empleado(1003, "Abcds", "AWa");
    Empleado e4 = new Empleado(1004, "Abcds", "AWa");
    Empleado e5 = new Empleado(1005, "aaa", "aaa");
    Empleado e6 = new Empleado(1006, "www", "aaa");

    empleadoDAO.save(e1);
    empleadoDAO.save(e2);
    empleadoDAO.save(e3);
    empleadoDAO.save(e4);
    empleadoDAO.save(e5);
    empleadoDAO.save(e6);

    List<Empleado> le = empleadoDAO.findAll();
    le = empleadoDAO.listarCuyoNombreContiene("w");
    LOG.info("====> testListarCuyoNombreContiene: 'w' = " + le.size() );

    empleadoDAO.deleteAll();

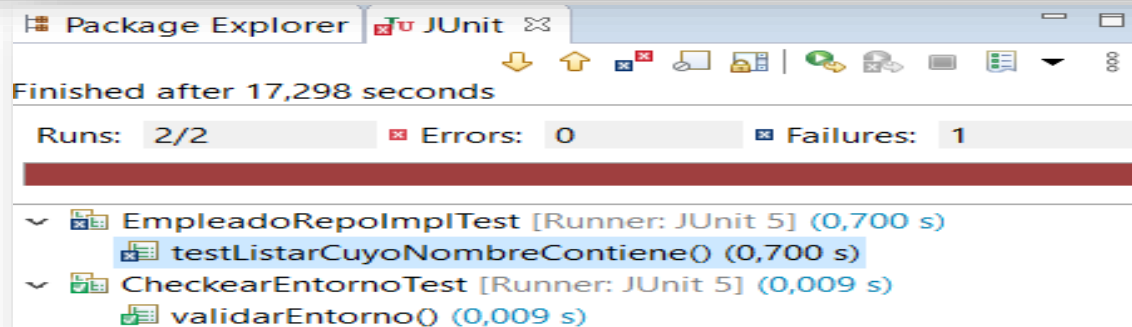
    assertTrue(le.size() == 3);
}
```


9. Definición de test unitarios

Ahora cuando ejecutemos los test unitarios:

```
2020-04-06 22:31:40.646 INFO 24068 --- [main] com.everis.CheckearEntornoTest : =====> ENTORNO TEST <=====
2020-04-06 22:31:40.647 INFO 24068 --- [main] com.everis.CheckearEntornoTest : =====> ENTORNO TEST <=====
2020-04-06 22:31:40.647 INFO 24068 --- [main] com.everis.CheckearEntornoTest : =====> ENTORNO TEST <=====
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 20.59 s - in com.everis.CheckearEntornoTest
[INFO] Running com.everis.repository.EmpleadoRepoImplTest
2020-04-06 22:31:40.709 INFO 24068 --- [main] .b.t.c.SpringBootTestContextBootstrapper : Neither @ContextConfiguration nor @Context
2020-04-06 22:31:40.710 INFO 24068 --- [main] o.s.t.c.support.AbstractContextLoader : Could not detect default resource location
2020-04-06 22:31:40.711 INFO 24068 --- [main] t.c.s.AnnotationConfigContextLoaderUtils : Could not detect default configuration c
2020-04-06 22:31:40.726 INFO 24068 --- [main] .b.t.c.SpringBootTestContextBootstrapper : Found @SpringBootConfiguration com.everi
2020-04-06 22:31:40.728 INFO 24068 --- [main] .b.t.c.SpringBootTestContextBootstrapper : Loaded default TestExecutionListener cla
2020-04-06 22:31:40.729 INFO 24068 --- [main] .b.t.c.SpringBootTestContextBootstrapper : Using TestExecutionListeners: [org.spring
2020-04-06 22:31:40.743 INFO 24068 --- [main] c.e.repository.EmpleadoRepoImplTest : ====> testListarCuyoNombreContiene: 'w'
[ERROR] Tests run: 1, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.202 s <<< FAILURE! - in com.everis.repository.EmpleadoRepoImplTest
[ERROR] testListarCuyoNombreContiene Time elapsed: 0.201 s <<< FAILURE!
org.opentest4j.AssertionFailedError: expected: <true> but was: <false>
    at com.everis.repository.EmpleadoRepoImplTest.testListarCuyoNombreContiene(EmpleadoRepoImplTest.java:52)
```

```
c.e.repository.EmpleadoRepoImplTest : Started EmpleadoRepoImplTest in 15.69 seconds (JVM running for 17.537)
c.e.repository.EmpleadoRepoImplTest : ====> testListarCuyoNombreContiene: 'w' = 2
```



9. Definición de test unitarios

Tendríamos que arreglar el problema:

- O que el método '**testListarCuyoNombreContiene**' tenga en cuenta mayúsculas y minúsculas.
- O que mire en nombre y también en los apellidos.
- O si está correcto, cambiarlo para que compruebe que devuelve dos resultados, y ya de paso añadirle otras validaciones más específicas.

```
@Test
void testListarCuyoNombreContiene () {
    empleadoDAO.deleteAll();


    Empleado e1 = new Empleado(1001, "AbcdWs", "A");
    Empleado e2 = new Empleado(1002, "Abcdws", "A");
    Empleado e3 = new Empleado(1003, "Abcds", "Awa");
    Empleado e4 = new Empleado(1004, "Abcds", "AWa");
    Empleado e5 = new Empleado(1005, "aaa", "aaa");
    Empleado e6 = new Empleado(1006, "www", "aaa");

    empleadoDAO.save(e1);
    empleadoDAO.save(e2);
    empleadoDAO.save(e3);
    empleadoDAO.save(e4);
    empleadoDAO.save(e5);
    empleadoDAO.save(e6);

    List<Empleado> le = empleadoDAO.findAll();
    le = empleadoDAO.listarCuyoNombreContiene("w");
    LOG.info(" ====> testListarCuyoNombreContiene: 'w' = " + le.size() );
    LOG.info(" ====> emp1: " + le.get(0).getId() );
    LOG.info(" ====> emp1: " + le.get(1).getId() );

    empleadoDAO.deleteAll();

    assertTrue(le.size() == 2);
    assertTrue( le.get(0).getId() == 1002 );
    assertTrue( le.get(1).getId() == 1006 );
}
```



Package Explorer JUnit

Finished after 16,874 seconds

Runs: 2/2 Errors: 0 Failures: 0

- ✓ EmpleadoRepoImplTest [Runner: JUnit 5] (0,861 s)
 - testListarCuyoNombreContiene() (0,861 s)
- ✓ CheckearEntornoTest [Runner: JUnit 5] (0,038 s)
 - validarEntorno() (0,038 s)

Started EmpleadoRepoImplTest in 15.18 seconds (JV

```
====> testListarCuyoNombreContiene: 'w' = 2
====> emp1: 1002
====> emp1: 1006

Neither @ContextConfiguration nor @ContextHierarchy
Could not detect default resource locations for t
Could not detect default configuration classes fo
Found @SpringBootTestConfiguration com.everis.DemoWeb
Loaded default TestExecutionListener class names
Using TestExecutionListeners: [org.springframework
```

====> ENTORNO TEST <====>



Actividad final



Actividad final:

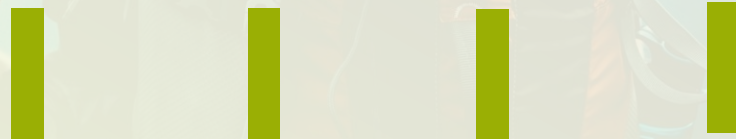
A la aplicación web que hemos desarrollado en paralelo se le pueden añadir algunas funcionalidades. Implementa las que más te interesen y consulta con el profesor de soporte asociado a esta formación las dudas o problemas que puedas tener:

- Definir un nuevo rol 'CONSULTA' y crear dos usuarios con dicho rol.
- Implementar una funcionalidad que muestre el listado de usuarios que son de un rol en concreto (por ejemplo del rol 'CONSULTA').
- Realizar el test unitario correspondiente para probar la funcionalidad del punto anterior.
- Implementar una página que muestre el listado de usuarios con dicho rol.
- Restringir la anterior para el rol 'ADMIN'.

everis

an NTT DATA Company

ever
FUTURE



everis (an NTT DATA Company)

Consulting, IT & Outsourcing Professional Services