
M6.UF4.A6.P4
CREACION DE UN SERVICIO
API REST SENCILLO

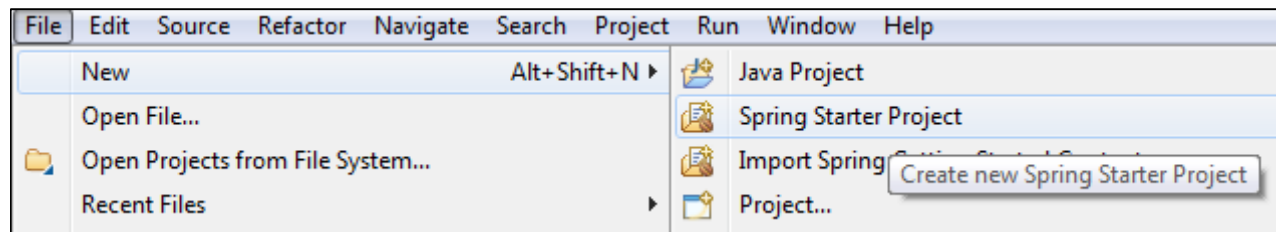
Eduard Lara

INDICE

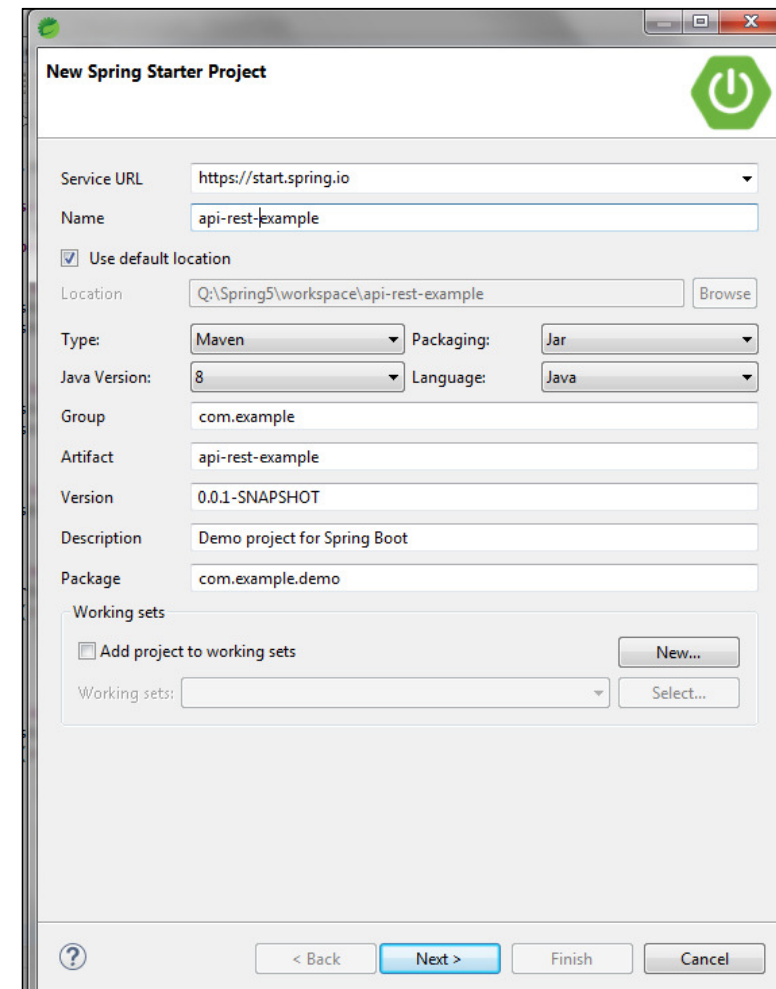
1. Creación proyecto
2. Controlador Rest
3. Modelo de datos
4. Inicialización H2
5. Inicializacion Mysql
6. Clase DAO
7. Servicios API REST
8. Métodos derivados JPA

1. CREACION PROYECTO

Paso 1) Creamos un proyecto Spring Boot, en la opción de menu File/New/Spring Starter Project:



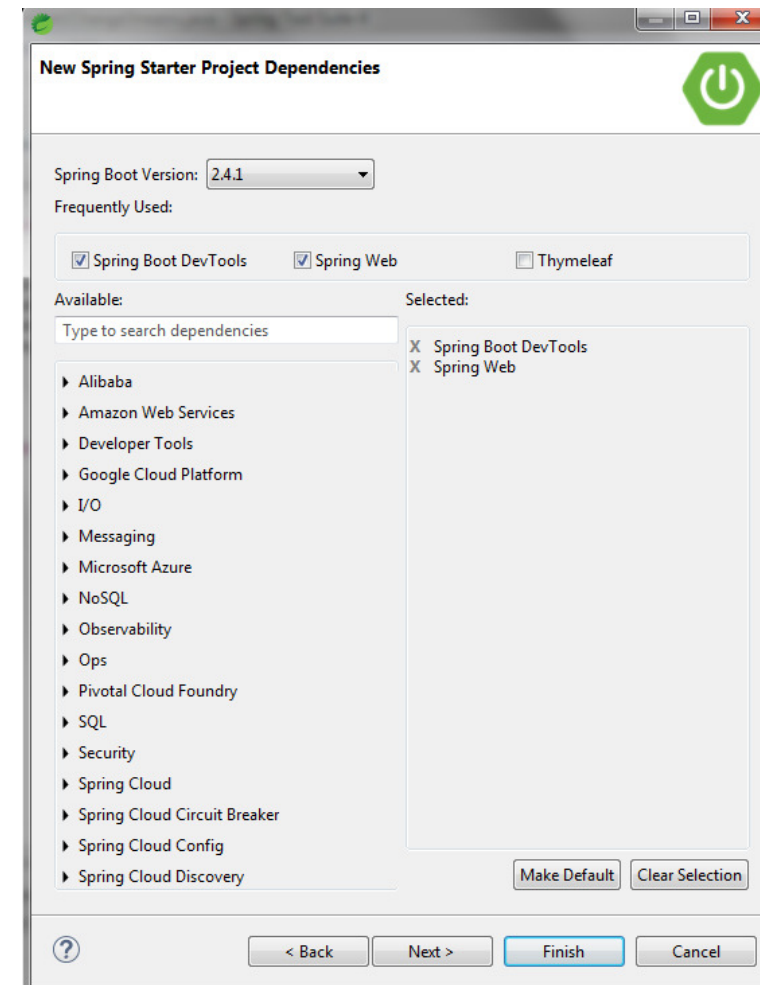
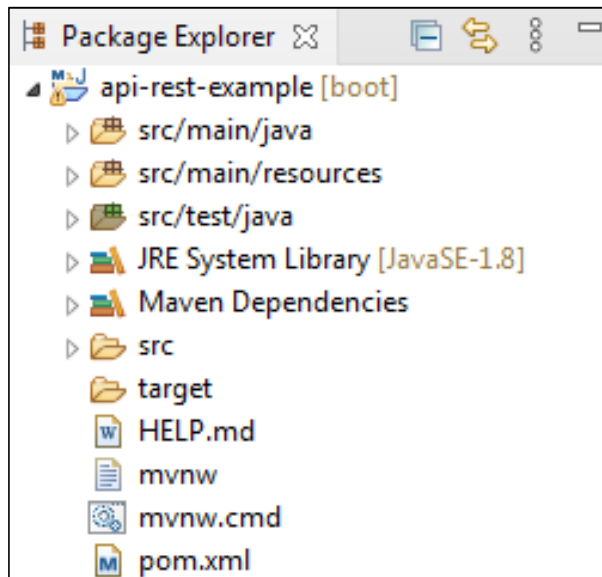
Podemos dejar por defecto los valores que nos presenta el wizard. Si se desea se puede cambiar el nombre de proyecto, el package raíz, el tipo de proyecto (Maven o Gradle) y/o la versión de Java.



1. CREACION PROYECTO

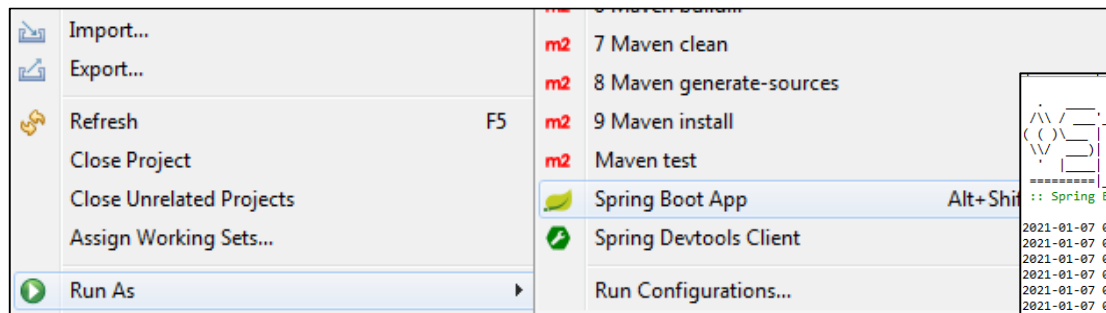
Paso 2) Agregamos las librerías:

- Spring Web (necesaria)
- Spring Boot Dev Tools (muy importante ya que cualquier cambio que hagamos en nuestro código java, de forma automática se va a actualizar en el despliegue sin tener que reiniciar el servidor)



1. CREACION PROYECTO

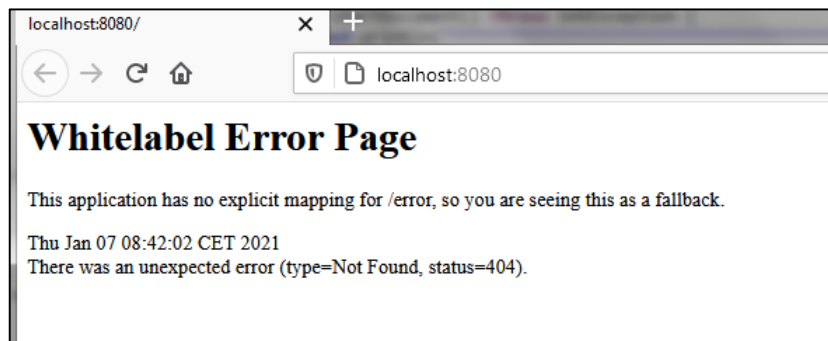
Paso 3) Probamos de ejecutar el proyecto, para ello levantamos el servidor Tomcat haciendo Run As/Spring Boot App. Una vez vemos que ha arrancado correctamente el servidor, vamos a un navegador y ponemos **localhost:8080**. Nos da error porque no tenemos ninguna página de inicio. Pero también significa que ya hay un servidor respondiendo en el puerto 8080.



```

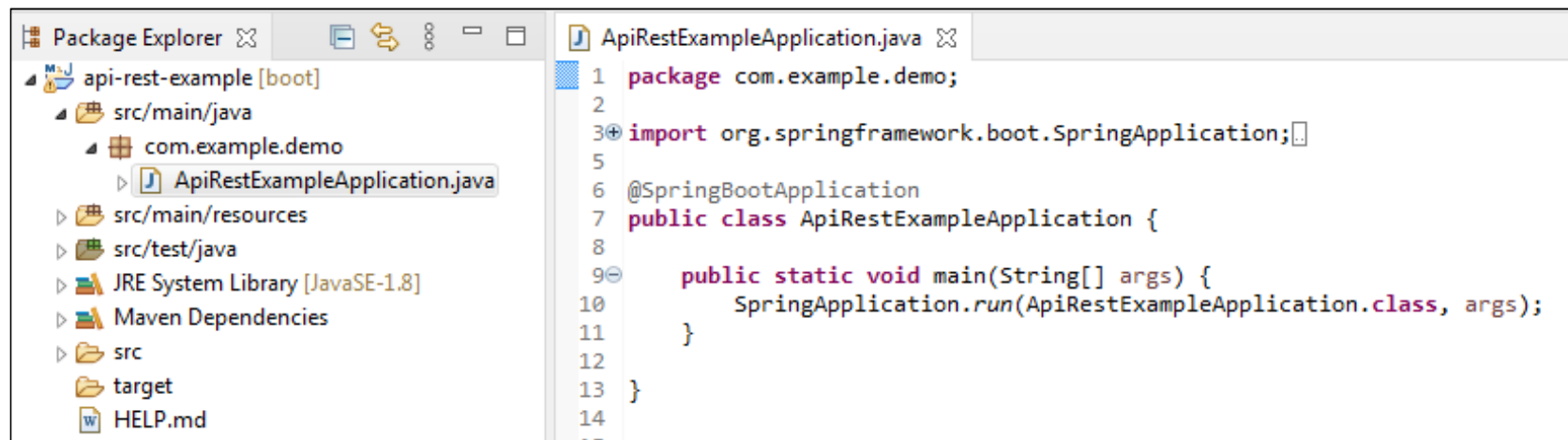
:: Spring Boot ::
(v2.4.1)

2021-01-07 08:41:46.209 INFO 16080 --- [main] c.e.demo.ApiRestExampleApplication : Starting ApiRestExampleApplication using Java 15.0
2021-01-07 08:41:46.214 INFO 16080 --- [main] c.e.demo.ApiRestExampleApplication : No active profile set, falling back to default profile
2021-01-07 08:41:47.260 INFO 16080 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2021-01-07 08:41:47.278 INFO 16080 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2021-01-07 08:41:47.279 INFO 16080 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.41]
2021-01-07 08:41:47.375 INFO 16080 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed
2021-01-07 08:41:47.375 INFO 16080 --- [main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2021-01-07 08:41:47.569 INFO 16080 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path: /
2021-01-07 08:41:47.799 INFO 16080 --- [nio-8080-exec-1] c.e.demo.ApiRestExampleApplication : Started ApiRestExampleApplication in 1.994 seconds
2021-01-07 08:42:02.577 INFO 16080 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'
2021-01-07 08:42:02.579 INFO 16080 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2021-01-07 08:42:02.579 INFO 16080 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 2 ms
```



1. CREACION PROYECTO

Paso 4) Podemos observar en el package raíz indicado al principio en la creación del proyecto, la clase generada automáticamente que inicia nuestro servidor y la aplicación:

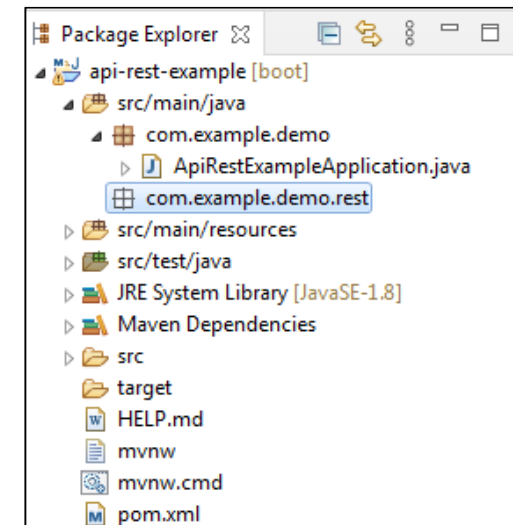
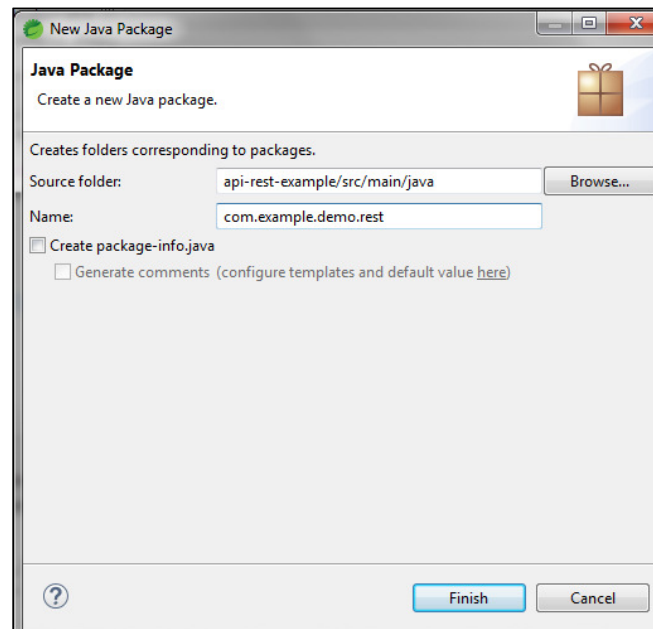
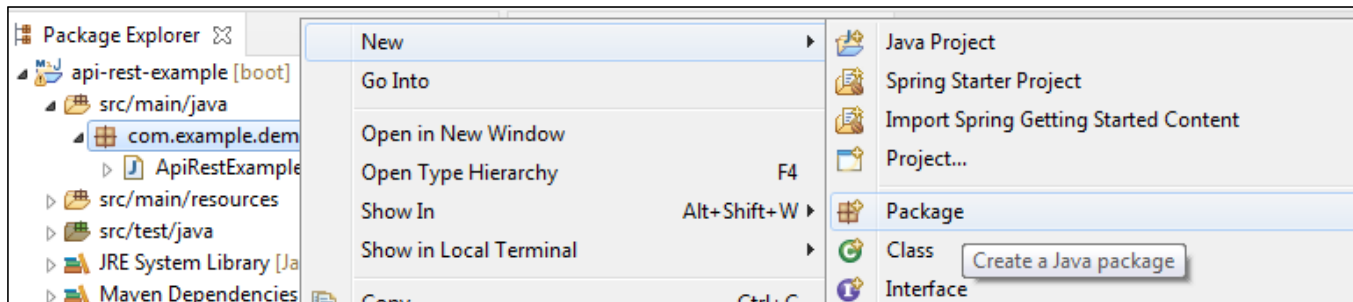


The screenshot shows an IDE interface. On the left, the 'Package Explorer' displays the project structure for 'api-rest-example [boot]'. The hierarchy is: 'src/main/java' > 'com.example.demo' > 'ApiRestExampleApplication.java'. On the right, the source code of 'ApiRestExampleApplication.java' is displayed. The code is as follows:

```
1 package com.example.demo;
2
3 import org.springframework.boot.SpringApplication;
4
5
6 @SpringBootApplication
7 public class ApiRestExampleApplication {
8
9     public static void main(String[] args) {
10         SpringApplication.run(ApiRestExampleApplication.class, args);
11     }
12 }
13
14
15
```

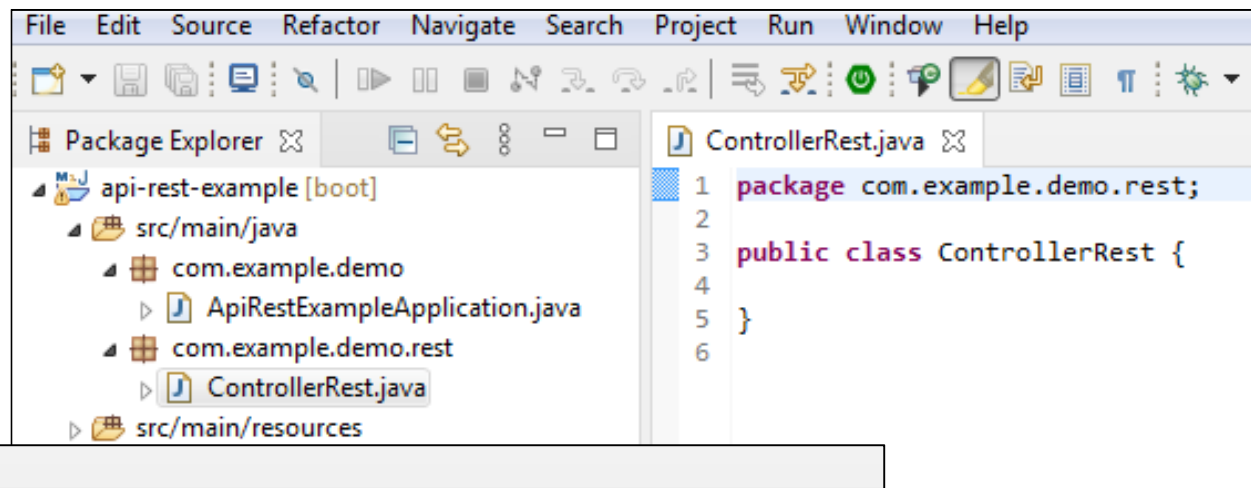
2. CONTROLADOR REST

Paso 1) Generamos un package dentro del existente con la extensión rest :



2. CONTROLADOR REST

Paso 2) Dentro de este package creamos una clase a la que le pondremos la etiqueta de controlador Rest. Aquí pondremos todos los servicios Rest que queremos que nuestra Api tenga:

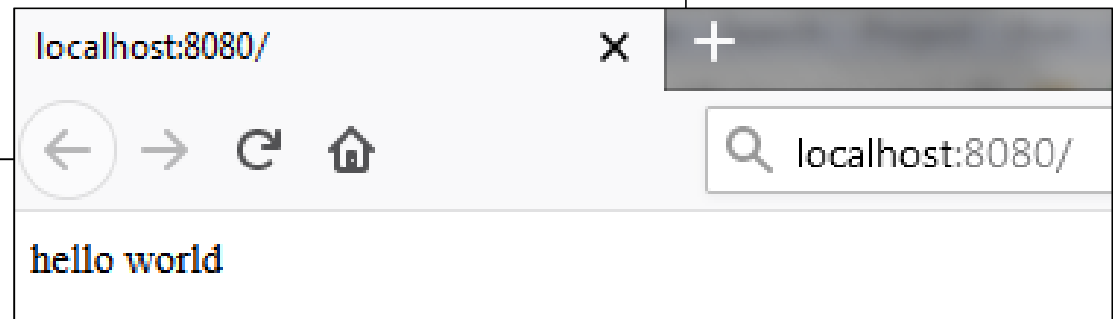


```
ControllerRest.java
1 package com.example.demo.rest;
2
3 import org.springframework.web.bind.annotation.RequestMapping;
4
5
6 @RestController           //Indica que esta clase va a ser un servicio REST
7 @RequestMapping("/")     //En que URL se va a exponer los servicios de esta clase
8 public class ControllerRest {
9
10 }
```


2. CONTROLADOR REST

Paso 3) Creamos una función hello, que retorna “hello world”, y le asignamos la etiqueta @GetMapping, habilitándola a que atienda peticiones HTTP de tipo Get. En concreto da servicio en la url localhost:8080/

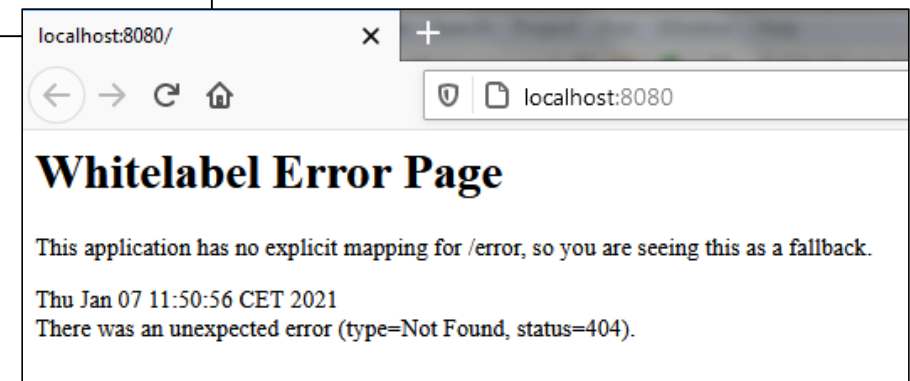
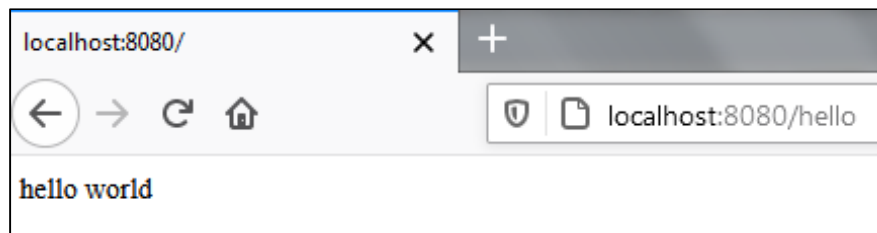
```
ControllerRest.java
1 package com.example.demo.rest;
2
3 import org.springframework.web.bind.annotation.GetMapping;
4
5
6 @RestController //Indica que esta clase va a ser un servicio REST
7 @RequestMapping("/") //En que URL se va a exponer los servicios de esta clase
8 public class ControllerRest {
9
10     @GetMapping("") //Servicio disponible mediante GET (localhost:8080/)
11     //@RequestMapping(value="hello", method=RequestMethod.GET) //En que url esta el servicio
12     public String hello() {
13         return "hello world";
14     }
15 }
16
17
18
```



2. CONTROLADOR REST

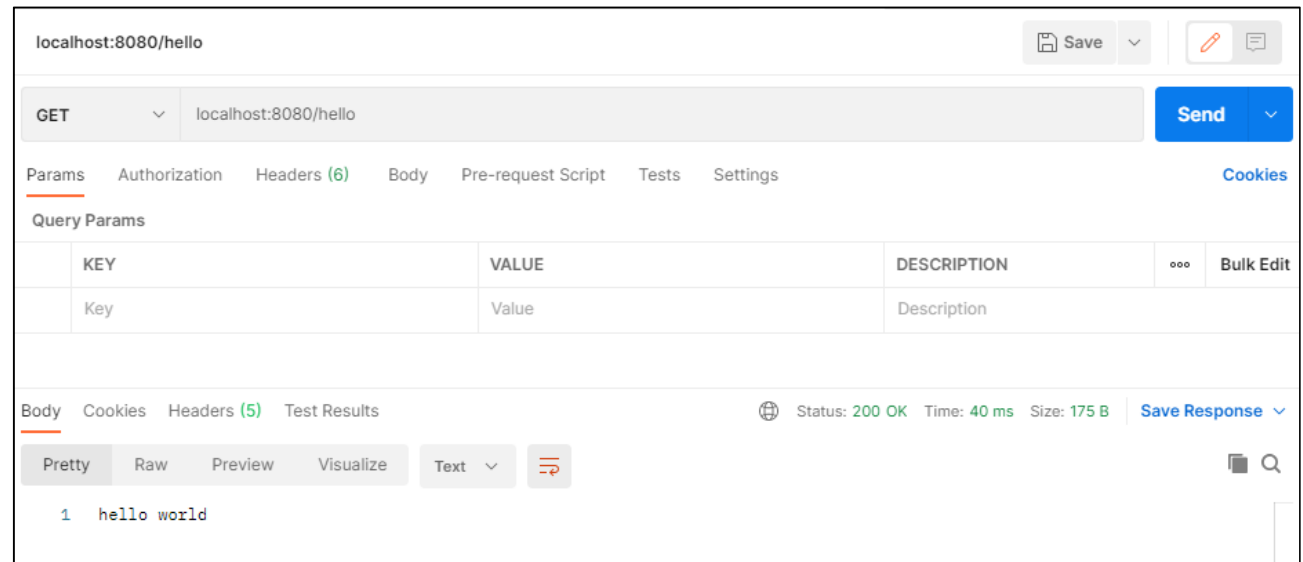
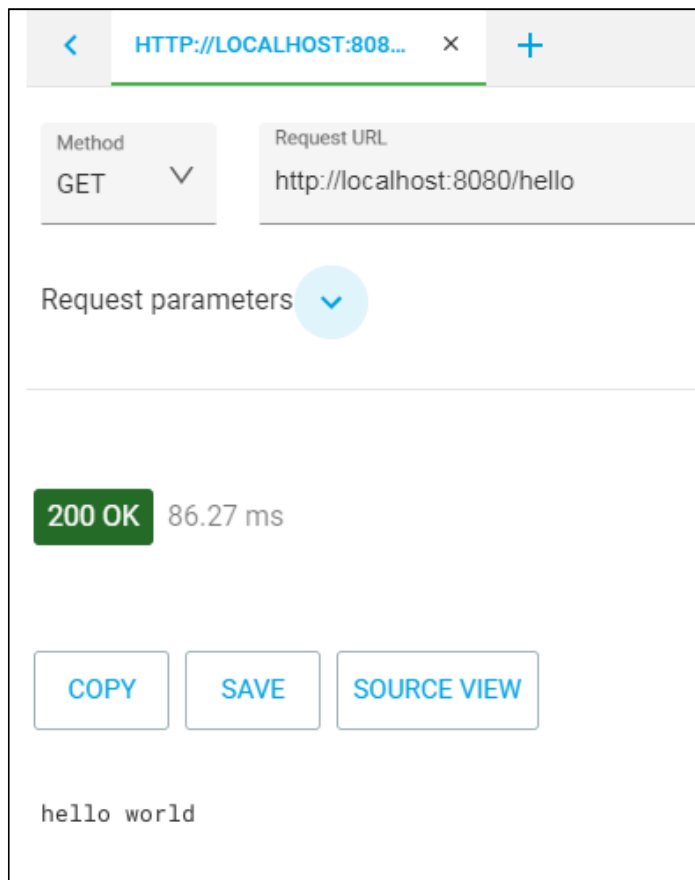
Paso 4) Si añadimos en el GetMapping el path “hello”, entonces la función daría servicio en la url localhost:8080/hello:

```
ControllerRest.java
1 package com.example.demo.rest;
2
3 import org.springframework.web.bind.annotation.GetMapping;
4
5
6
7
8 @RestController           //Indica que esta clase va a ser un servicio REST
9 @RequestMapping("/")     //En que URL se va a exponer los servicios de esta clase
10 public class ControllerRest {
11
12     @GetMapping("hello")  //Servicio disponible mediante GET (localhost:8080/hello)
13     //@RequestMapping(value="hello", method=RequestMethod.GET) //En que url esta el servicio
14     public String hello() {
15         return "hello world";
16     }
17 }
```



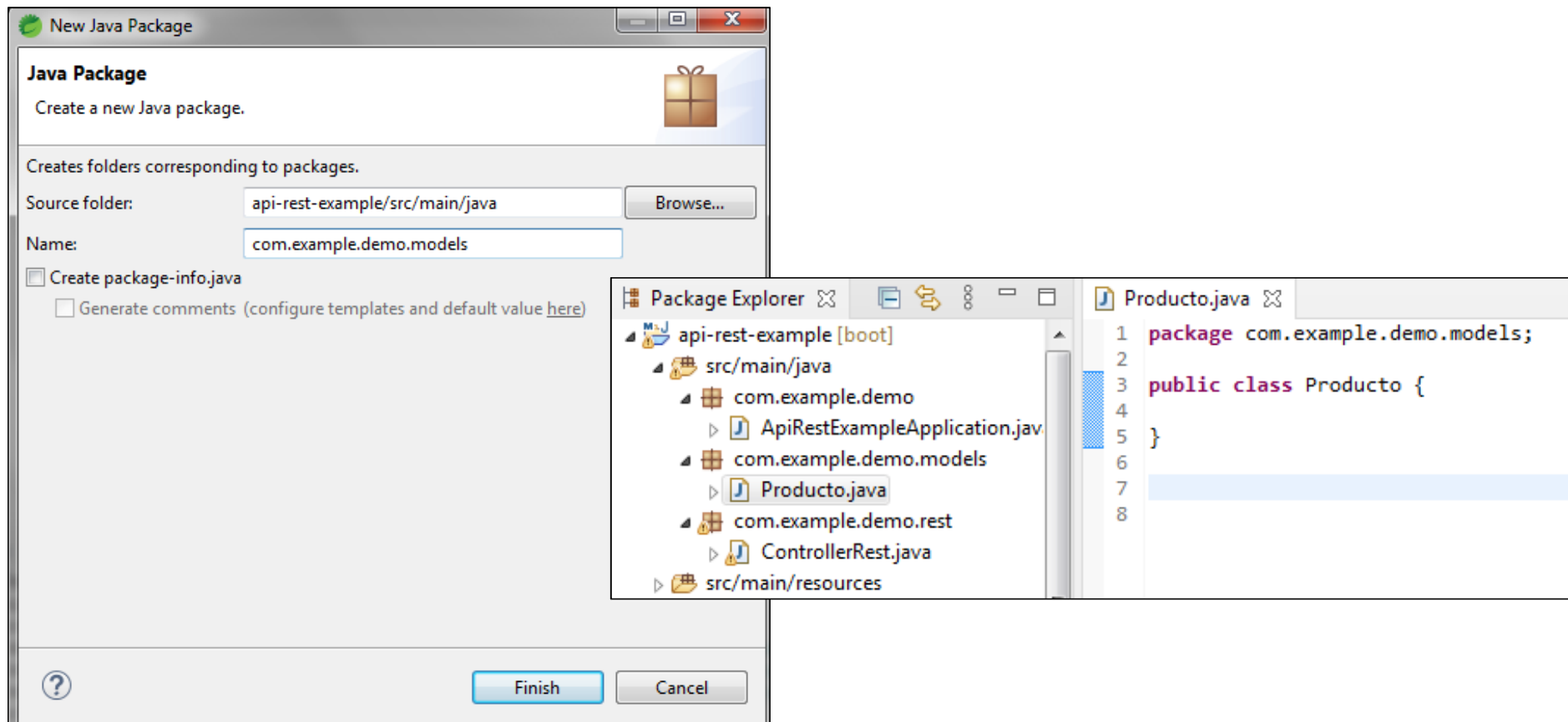
2. CONTROLADOR REST

Paso 5) Podemos probar el servicio con las aplicaciones Advanced Rest Client o Postman:



3. MODELO DE DATOS

Paso 1) Creamos la clase Producto dentro de un nuevo Package con extensión models:



3. MODELO DE DATOS

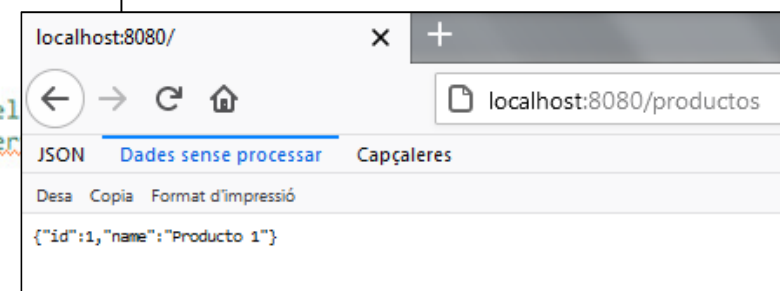
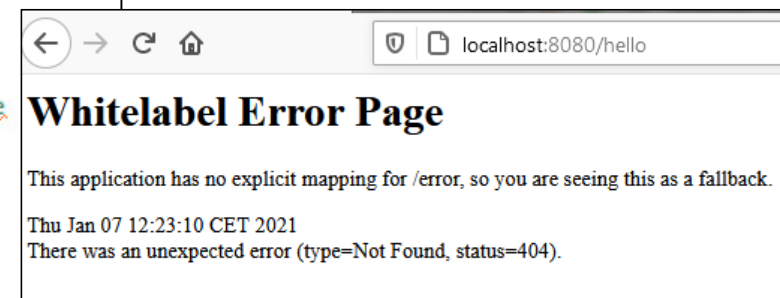
Paso 2) Creamos dos atributos simples en la clase POJO Producto, la cual representará el modelo de los datos de la base de datos:

```
Producto.java
1 package com.example.demo.models;
2
3 public class Producto {
4     private long id;
5     private String name;
6
7     public long getId() {
8         return id;
9     }
10    public void setId(long id) {
11        this.id = id;
12    }
13    public String getName() {
14        return name;
15    }
16    public void setName(String name) {
17        this.name = name;
18    }
19 }
20
```

3. MODELO DE DATOS

Paso 3) Desactivamos la función hello comentando su GetMapping. Ponemos un Mapping general al controlador “/productos”, y creamos una nueva función getProductos() que nos ofrecerá la lista de productos en localhost:8080/productos:

```
ControllerRest.java
1 package com.example.demo.rest;
2
3 import org.springframework.http.ResponseEntity;
4
5 @RestController //Indica que esta clase va a ser un servicio REST
6 @RequestMapping("/productos") //En que URL se va a exponer los servicios de esta clase
7 public class ControllerRest {
8
9     @GetMapping
10     public ResponseEntity<Producto> getProducto() {
11         Producto producto = new Producto();
12         producto.setId(1);
13         producto.setName("Producto 1");
14         return ResponseEntity.ok(producto);
15     }
16
17     // @GetMapping("hello") //Servicio disponible mediante GET (localhost:8080/hello)
18     // @RequestMapping(value="hello", method=RequestMethod.GET) //En que url esta el ser
19     public String hello() {
20         return "hello world";
21     }
22 }
```



3. MODELO DE DATOS

Paso 4) Comprobamos el servicio con ARC (Advanced Rest Client):

The screenshot shows the ARC interface with a GET request to `http://localhost:8080/productos`. The response status is **200 OK** with a response time of **16.67 ms**. The response body is displayed in a light gray box with the following JSON:

```
{
  "id": 1,
  "name": "Producto 1"
}
```

Buttons for **COPY**, **SAVE**, **SOURCE VIEW**, and **DATA TABLE** are visible at the bottom.

The screenshot shows the ARC interface with a GET request to `localhost:8080/productos`. The response status is **200 OK** with a response time of **16.67 ms**. The response body is displayed in a light gray box with the following JSON:

```
{
  "id": 1,
  "name": "Producto 1"
}
```

Buttons for **COPY**, **SAVE**, **SOURCE VIEW**, and **DATA TABLE** are visible at the bottom.

4. INICIALIZACION H2

H2 es un sistema administrador de bases de datos relacionales embebido programado en Java. Puede ser incorporado en aplicaciones Java o ejecutarse en modo cliente-servidor. Tiene dos versiones: en memoria o en fichero físico

Las principales características de H2 son:

- API JDBC de código abierto muy rápido
- Modos integrados y de servidor; bases de datos en memoria
- Tamaño reducido: alrededor de 2 MB de tamaño de archivo jar
- Aplicación de consola basada en navegador

Features

	H2	Derby	HSQLDB	MySQL	PostgreSQL
Pure Java	Yes	Yes	Yes	No	No
Memory Mode	Yes	Yes	Yes	No	No
Encrypted Database	Yes	Yes	Yes	No	No
ODBC Driver	Yes	No	No	Yes	Yes
Fulltext Search	Yes	No	No	Yes	Yes
Multi Version Concurrency	Yes	No	Yes	Yes	Yes
Footprint (embedded)	~2 MB	~3 MB	~1.5 MB	—	—
Footprint (client)	~500 KB	~600 KB	~1.5 MB	~1 MB	~700 KB

4. INICIALIZACION H2

Paso 1) Primero, para que sea posible la generación de la base de datos H2 de forma automática al iniciar un proyecto Spring, debemos agregar el driver Hibernate JPA. JPA o Java Persistence API es la API de persistencia desarrollada para la plataforma Java EE. Hacemos click botón derecho sobre el proyecto y elegimos la opción Add Starters. Seleccionamos y agregamos el driver JPA:

The image illustrates the steps to add Spring Data JPA as a project dependency. It is divided into three main sections:

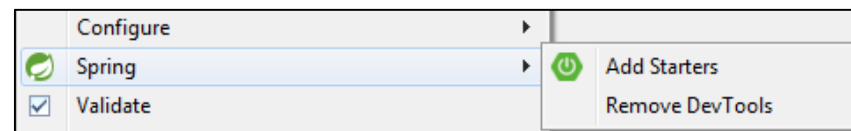
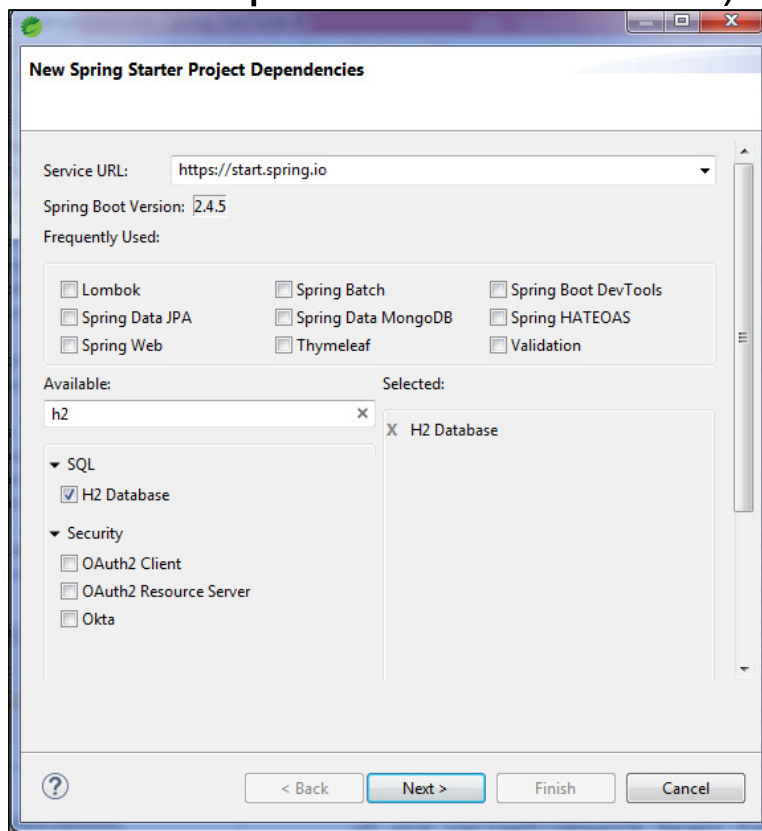
- Top Section:** A context menu is shown with the 'Spring' option selected. A sub-menu is open, displaying 'Add Starters' (highlighted with a green power icon) and 'Remove DevTools'.
- Bottom Left Section:** A snippet of a `pom.xml` file is displayed. The file is named `api-rest-example/pom.xml`. The XML content is as follows:

```
28     <scope>runtime</scope>
29     <optional>true</optional>
30 </dependency>
31 <dependency>
32     <groupId>org.springframework.boot</groupId>
33     <artifactId>spring-boot-starter-test</artifactId>
34     <scope>test</scope>
35 </dependency>
36 <dependency>
37     <groupId>org.springframework.boot</groupId>
38     <artifactId>spring-boot-starter-data-jpa</artifactId>
39 </dependency>
40 </dependencies>
```

The dependency for `spring-boot-starter-data-jpa` (lines 36-39) is highlighted with a red rectangular box.
- Bottom Right Section:** A dialog box titled 'New Spring Starter Project Dependencies' is shown. It has a 'Service URL' of `https://start.spring.io` and a 'Spring Boot Version' of `2.4.5`. Under 'Frequently Used', several dependencies are listed with checkboxes: ☐ Lombok, ☒ Spring Data JPA, ☐ Spring Data MongoDB, ☐ Spring Data Redis, ☐ Spring HATEOAS, ☐ Spring Web, ☐ Thymeleaf, ☐ Spring Boot DevTools, and ☐ Validation. Below this, there are two panes: 'Available:' and 'Selected:'. The 'Available:' pane shows a tree of categories like 'Developer Tools', 'Google Cloud Platform', etc. The 'Selected:' pane shows 'X Spring Data JPA'.

4. INICIALIZACION H2

Paso 2) Agregamos la dependencia del driver H2 a nuestro proyecto (pe. mediante el fichero pom.xml de Maven):



```
api-rest-example/pom.xml
40 <dependency>
41     <groupId>org.springframework.boot</groupId>
42     <artifactId>spring-boot-starter</artifactId>
43 </dependency>
44 <dependency>
45     <groupId>com.h2database</groupId>
46     <artifactId>h2</artifactId>
47     <scope>runtime</scope>
48 </dependency>
49 </dependencies>
50
```

4. INICIALIZACION H2

Paso 3) Transformamos la clase Producto en una clase Entity, mediante las anotaciones **@Entity** y **@Table**.

@Id → indicamos que el atributo de la clase es la primary key de la tabla.

@Column → mapeamos el atributo con la columna de la tabla indicada en @Table.

De esta forma implementamos la persistencia al establecer la correspondencia entre un objeto de la clase entity Producto y un registro-fila de la tabla productos.

Mediante la persistencia no utilizaremos el típico lenguaje DML de SQL para acceder a la base de datos, sino una API mas sencilla y además orientada a objetos.

```
Producto.java
4 import javax.persistence.Entity;
5 import javax.persistence.GeneratedValue;
6 import javax.persistence.GenerationType;
7 import javax.persistence.Id;
8 import javax.persistence.Table;
9
10 @Entity
11 @Table(name="productos")
12 public class Producto {
13
14     @Id
15     @Column(name="id")
16     @GeneratedValue(strategy=GenerationType.IDENTITY)
17     private long id;
18
19     @Column(name="name", nullable=false, length=30)
20     private String name;
21
22     public long getId() {
23         return id;
24     }
25     public void setId(long id) {
26         this.id = id;
27     }
28     public String getName() {
29         return name;
30     }
31     public void setName(String name) {
32         this.name = name;
33     }
34 }
35
```

4. INICIALIZACION H2

Fichero de configuración externa **application.properties**

Este fichero contiene una serie de directivas que ayudan en la configuración de nuestro proyecto Spring: cambio de puerto, configuración de la conexión a la base de datos, inicialización de base de datos y carga de datos, configuración de la persistencia, etc

Organización del fichero **application.properties**

Spring permite repartir la información de configuración en diferentes ficheros. En nuestro caso podemos hacer la siguiente división en 3 ficheros:

- **application.properties** → Puntero hacia un perfil determinado de base de datos
- **application-h2.properties** → Configuración de acceso a H2
- **application-mysql.properties** → Configuración de acceso a mysql

4. INICIALIZACION H2

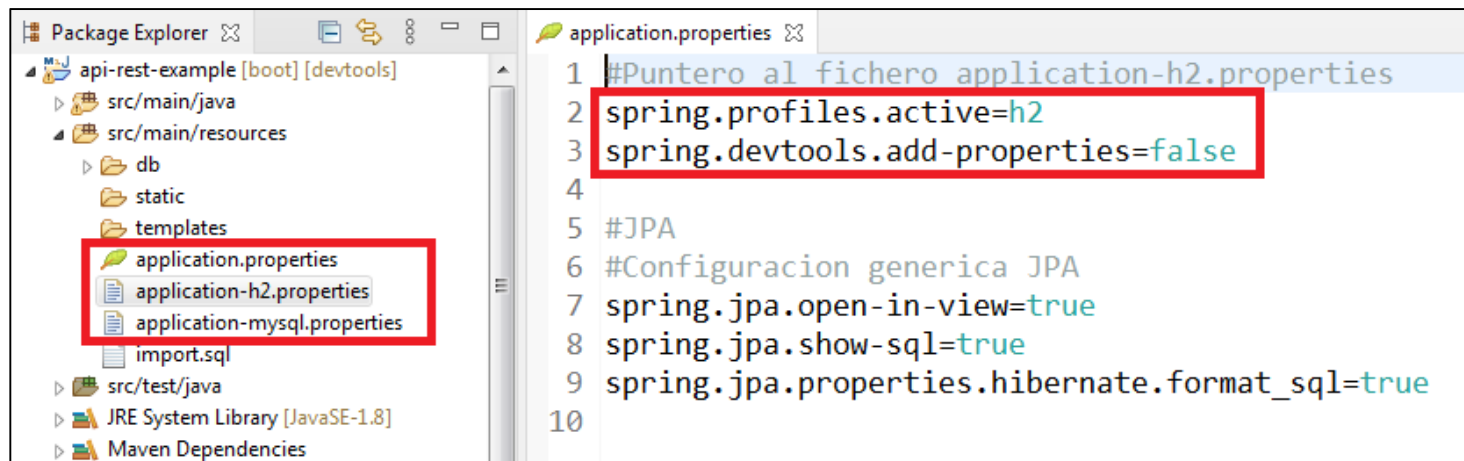
- Spring permite inicializar la base de datos de las siguientes formas:
 - Inicialización DDL → Usando la configuración de las clases entity (Hibernate JPA) o Mediante el fichero externo schema.sql
 - Inicialización DML → Mediante los ficheros import.sql y/o data.sql
- La directiva **spring.jpa.hibernate.ddl-auto** controla este proceso. Puede tomar los siguientes valores:
 - create – Hibernate elimina las tablas existentes y después las crea nuevas
 - create-drop – Hibernate elimina la db después de realizar las operaciones.
 - update – Hibernate actualiza el esquema de la db si hay diferencias. Nunca borra tablas o columnas en caso de que no sean necesarias
 - → Se usa en **entornos de desarrollo**
 - validate – solo valida si las tablas y columnas existen, sino lanza excepción
 - none – Desactiva la generación DDL de JPA → Para **entornos de producción**

4. INICIALIZACION H2

Configuración H2 para inicialización DDL con JPA entity

Paso 3) Creamos la estructura de ficheros application.properties, haciendo copy&paste del archivo original. Indicaremos que application.properties apunte al fichero de configuración específico para h2. En este caso:

- Creación DDL → JPA hace el mapeo de datos basándose en las clases entity
- Carga DML → import.sql y/o schema.sql



4. INICIALIZACION H2

Configuración H2 para inicialización DDL con JPA entity

Paso 4) En application-h2.properties indicamos las siguientes directivas:

```
application-h2.properties
1 #H2
2 #Configuracion del datasource con H2
3 spring.datasource.platform=h2
4 spring.datasource.url=jdbc:h2:mem:tienda
5 #spring.datasource.url=jdbc:h2:file:tienda
6 spring.datasource.driverClassName=org.h2.Driver
7 spring.datasource.username=sa
8 spring.datasource.password=
9 #spring.datasource.initialization-mode=always
10 #spring.datasource.schema=classpath:db/schema.sql
11 spring.datasource.data=classpath:db/data.sql
12
13 spring.h2.console.enabled=true
14 spring.h2.console.path=/h2-console
15
16 #JPA
17 #Configuracion del JPA
18 spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
19 #spring.jpa.generate-ddl=true
20 #spring.jpa.hibernate.ddl-auto=create
```

La inicialización DML se puede realizar con import.sql en resources y/o con el archivo data.sql (activando su correspondiente directiva)

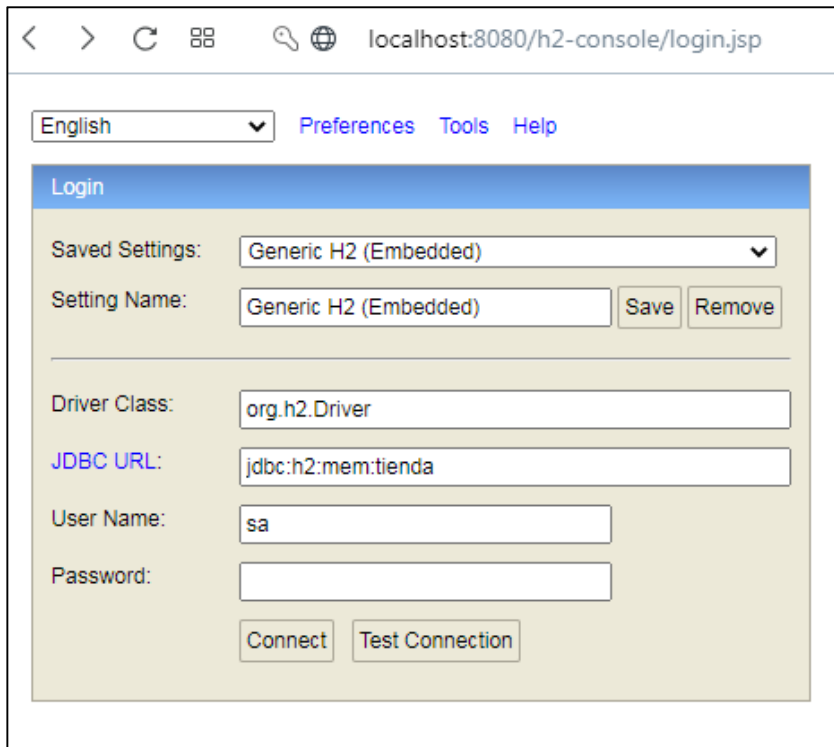
```
data: Bloc de notas
Archivo Edición Formato Ver Ayuda
INSERT INTO PRODUCTOS VALUES (null, 'Producto 1');
INSERT INTO PRODUCTOS VALUES (null, 'Producto 2');
INSERT INTO PRODUCTOS VALUES (null, 'Producto 3');
INSERT INTO PRODUCTOS VALUES (null, 'Producto 4');
```

```
import: Bloc de notas
Archivo Edición Formato Ver Ayuda
INSERT INTO PRODUCTOS VALUES (null, 'Producto 41');
INSERT INTO PRODUCTOS VALUES (null, 'Producto 42');
INSERT INTO PRODUCTOS VALUES (null, 'Producto 43');
INSERT INTO PRODUCTOS VALUES (null, 'Producto 44');
```

4. INICIALIZACION H2

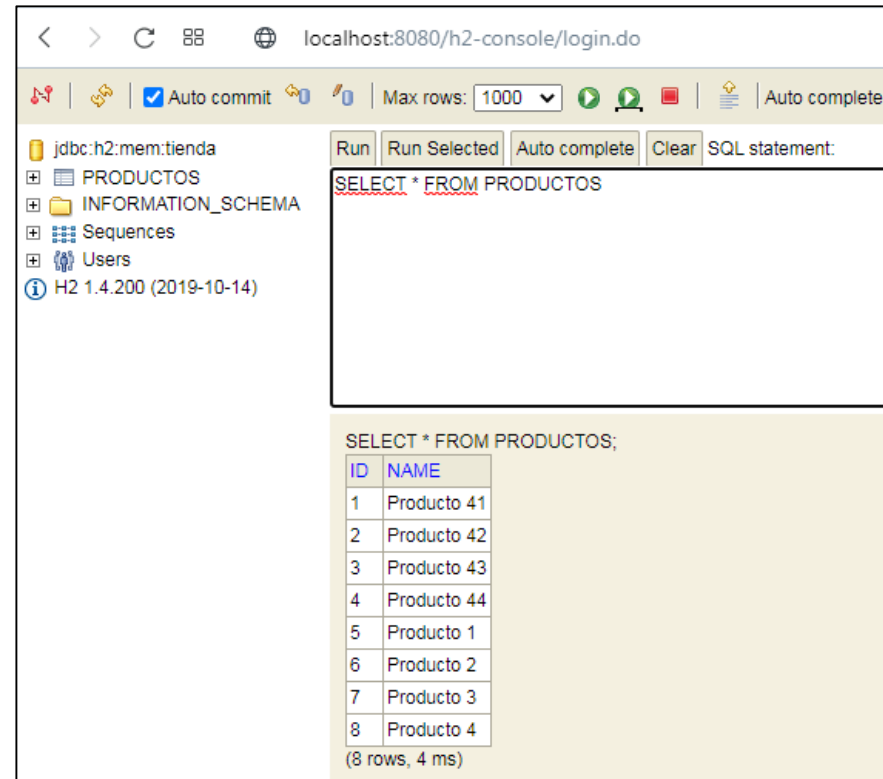
Configuración H2 para inicialización DDL con JPA entity

Paso 5) Al arrancar el servidor, automáticamente se crea la base de datos tienda en H2 en memoria con sus datos. La url de acceso es: **localhost:8080/h2-console**



The screenshot shows the H2 console login page. The browser address bar displays `localhost:8080/h2-console/login.jsp`. The page has a language dropdown set to "English" and links for "Preferences", "Tools", and "Help". The "Login" section contains the following fields and buttons:

- Saved Settings:** A dropdown menu showing "Generic H2 (Embedded)".
- Setting Name:** A text input field containing "Generic H2 (Embedded)", with "Save" and "Remove" buttons.
- Driver Class:** A text input field containing "org.h2.Driver".
- JDBC URL:** A text input field containing "jdbc:h2:mem:tienda".
- User Name:** A text input field containing "sa".
- Password:** An empty text input field.
- Buttons:** "Connect" and "Test Connection".



The screenshot shows the H2 console main page. The browser address bar displays `localhost:8080/h2-console/login.do`. The page includes a toolbar with icons for "Auto commit", "Max rows" (set to 1000), and "Auto complete". The left sidebar shows the database structure:

- jdbc:h2:mem:tienda
 - PRODUCTOS
 - INFORMATION_SCHEMA
 - Sequences
 - Users
 - H2 1.4.200 (2019-10-14)

The SQL statement editor contains the query:

```
SELECT * FROM PRODUCTOS
```

The query results are displayed in a table:

ID	NAME
1	Producto 41
2	Producto 42
3	Producto 43
4	Producto 44
5	Producto 1
6	Producto 2
7	Producto 3
8	Producto 4

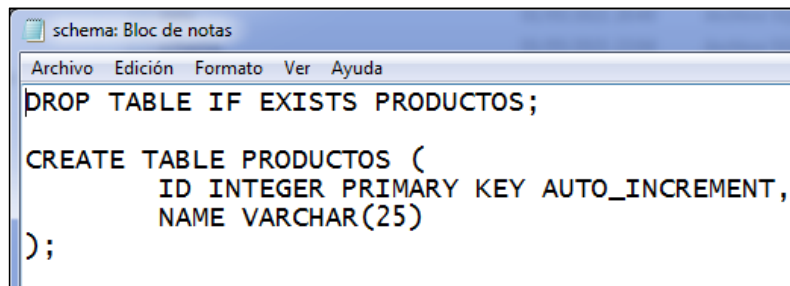
Below the table, it indicates "(8 rows, 4 ms)".

4. INICIALIZACION H2

Configuración H2 para inicialización DDL con schema.sql

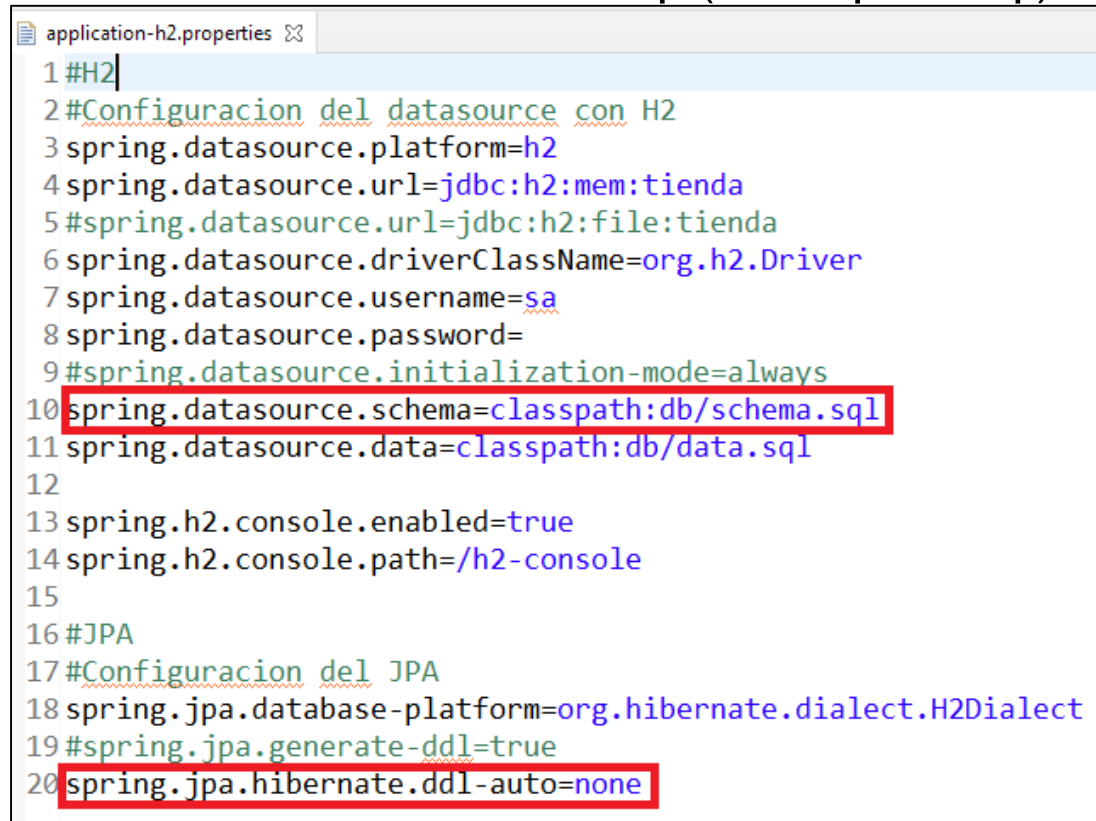
Paso 4 bis) En este caso la inicialización DDL se realiza mediante el archivo schema.sql y la carga DML a través únicamente del archivo data.sql (no import.sql).

Se debe desactivar la generación DDL con JPA mediante la directiva **spring.jpa.hibernate.ddl-auto=none** y activar la directiva de schema.sql **spring.datasource.schema=classpath:db/schema.sql**



```
schema: Bloc de notas
Archivo Edición Formato Ver Ayuda
DROP TABLE IF EXISTS PRODUCTOS;

CREATE TABLE PRODUCTOS (
    ID INTEGER PRIMARY KEY AUTO_INCREMENT,
    NAME VARCHAR(25)
);
```

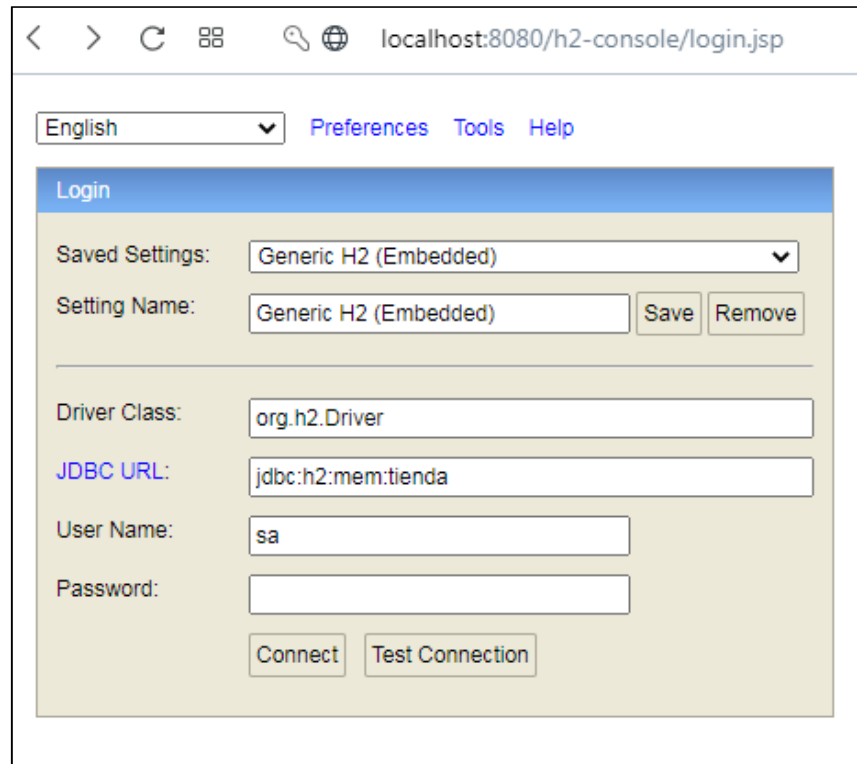


```
application-h2.properties
1 #H2
2 #Configuracion del datasource con H2
3 spring.datasource.platform=h2
4 spring.datasource.url=jdbc:h2:mem:tienda
5 #spring.datasource.url=jdbc:h2:file:tienda
6 spring.datasource.driverClassName=org.h2.Driver
7 spring.datasource.username=sa
8 spring.datasource.password=
9 #spring.datasource.initialization-mode=always
10 spring.datasource.schema=classpath:db/schema.sql
11 spring.datasource.data=classpath:db/data.sql
12
13 spring.h2.console.enabled=true
14 spring.h2.console.path=/h2-console
15
16 #JPA
17 #Configuracion del JPA
18 spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
19 #spring.jpa.generate-ddl=true
20 spring.jpa.hibernate.ddl-auto=none
```

4. INICIALIZACION H2

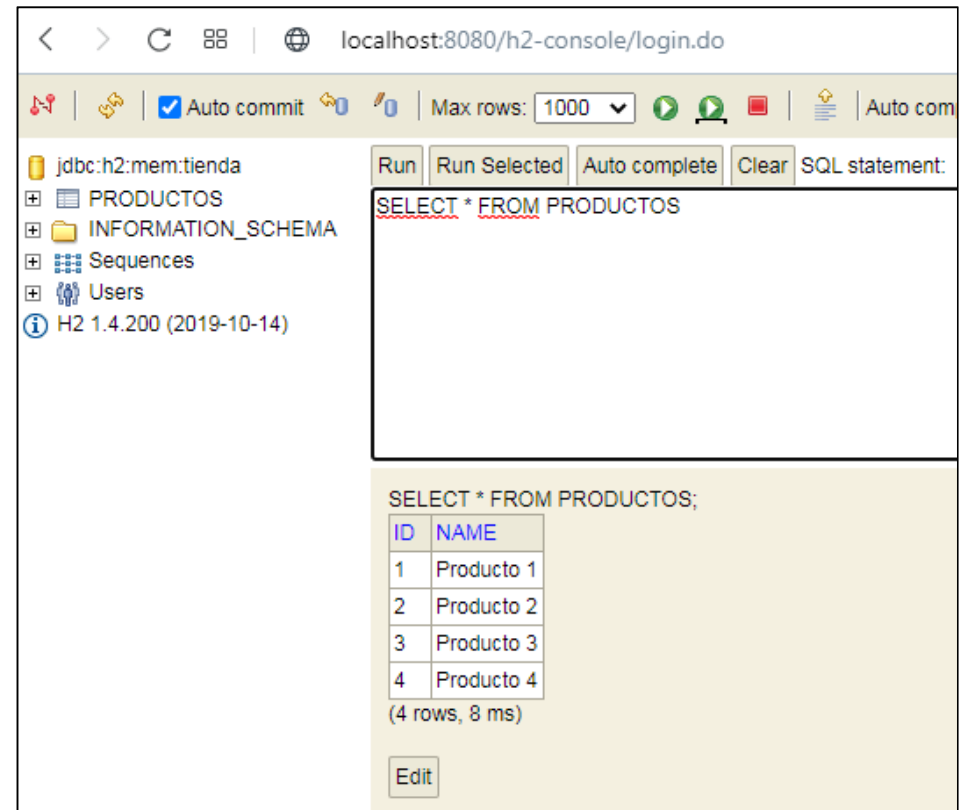
Configuración H2 para inicialización DDL con schema.sql

Paso 5 bis) Al arrancar el servidor, automáticamente se crea la base de datos tienda en H2 en memoria con sus datos. La url de acceso es: **localhost:8080/h2-console**



The screenshot shows the login page of the H2 console at localhost:8080/h2-console/login.jsp. The page has a language dropdown set to 'English' and links for 'Preferences', 'Tools', and 'Help'. The 'Login' section contains the following fields and buttons:

- Saved Settings:** A dropdown menu showing 'Generic H2 (Embedded)'.
- Setting Name:** A text input field containing 'Generic H2 (Embedded)', with 'Save' and 'Remove' buttons.
- Driver Class:** A text input field containing 'org.h2.Driver'.
- JDBC URL:** A text input field containing 'jdbc:h2:mem:tienda'.
- User Name:** A text input field containing 'sa'.
- Password:** An empty text input field.
- Buttons:** 'Connect' and 'Test Connection'.



The screenshot shows the main interface of the H2 console at localhost:8080/h2-console/login.do. The interface includes a toolbar with 'Run', 'Run Selected', 'Auto complete', and 'Clear' buttons, along with a 'SQL statement:' input field. The left sidebar displays the database structure for 'jdbc:h2:mem:tienda':

- PRODUCTOS
- INFORMATION_SCHEMA
- Sequences
- Users
- H2 1.4.200 (2019-10-14)

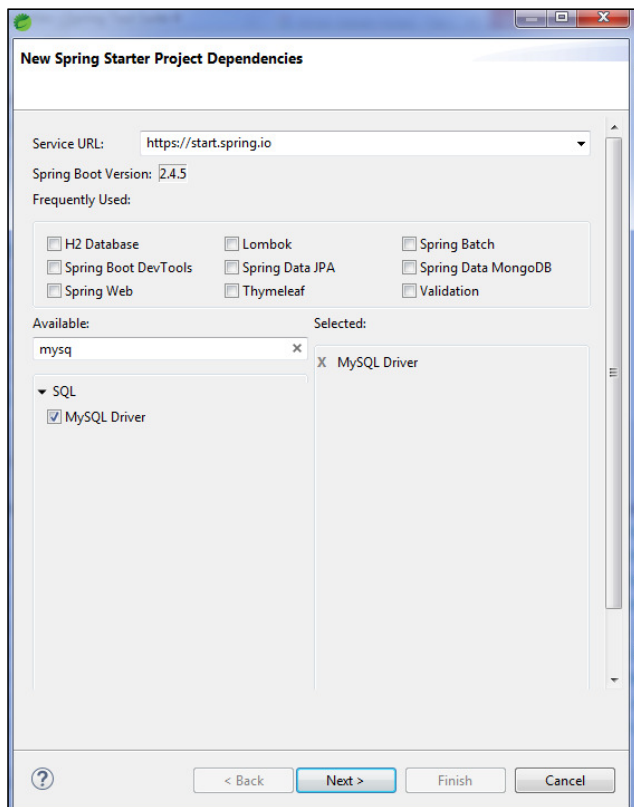
The SQL statement 'SELECT * FROM PRODUCTOS;' is entered and executed, resulting in the following table:

ID	NAME
1	Producto 1
2	Producto 2
3	Producto 3
4	Producto 4

Below the table, it indicates '(4 rows, 8 ms)' and an 'Edit' button.

5. INICIALIZACION MYSQL

Paso 1) Para tener acceso a la base de datos Mysql, se debe de agregar la dependencia del driver mysql en el fichero pom.xml del repositorio Maven:



```
api-rest-example/pom.xml
43     <dependency>
44         <groupId>com.h2database</groupId>
45         <artifactId>h2</artifactId>
46         <scope>runtime</scope>
47     </dependency>
48     <dependency>
49         <groupId>mysql</groupId>
50         <artifactId>mysql-connector-java</artifactId>
51         <scope>runtime</scope>
52     </dependency>
53 </dependencies>
```

5. INICIALIZACION MYSQL

Configuración Mysql para inicialización DDL con JPA entity

Paso 2) Debemos activar la clase producto como entity y seguir la siguiente configuración en el application-mysql.properties.

The image displays the configuration for a Spring application to connect to MySQL and generate DDL using JPA. It includes two Notepad windows showing SQL data for a 'PRODUCTOS' table.

application-mysql.properties

```
1 #MySQL
2 #Configuracion del datasource con MySQL
3 spring.datasource.platform=mysql
4 spring.datasource.driverClassName=com.mysql.cj.jdbc.Driver
5 spring.datasource.url=jdbc:mysql://localhost:3306/tienda?createDatabaseIfNotExist=true
6 spring.datasource.username=root
7 spring.datasource.password=
8 spring.datasource.initialization-mode=always
9 #spring.datasource.schema=classpath:db/schema.sql
10 spring.datasource.data=classpath:db/data.sql
11
12 #JPA
13 #Configuracion del JPA
14 spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect
15 #spring.jpa.generate-ddl=true
16 spring.jpa.hibernate.ddl-auto=create
```

application.properties

```
1 #Puntero al fichero application-mysql.properties
2 spring.profiles.active=mysql
3 spring.devtools.add-properties=false
```

data: Bloc de notas

```
Archivo Edición Formato Ver Ayuda
INSERT INTO PRODUCTOS VALUES (null, 'Producto 1');
INSERT INTO PRODUCTOS VALUES (null, 'Producto 2');
INSERT INTO PRODUCTOS VALUES (null, 'Producto 3');
INSERT INTO PRODUCTOS VALUES (null, 'Producto 4');
```

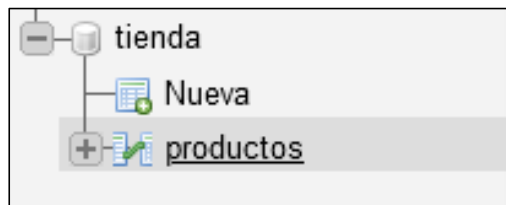
import: Bloc de notas

```
Archivo Edición Formato Ver Ayuda
INSERT INTO PRODUCTOS VALUES (null, 'Producto 41');
INSERT INTO PRODUCTOS VALUES (null, 'Producto 42');
INSERT INTO PRODUCTOS VALUES (null, 'Producto 43');
INSERT INTO PRODUCTOS VALUES (null, 'Producto 44');
```

5. INICIALIZACION MYSQL

Configuración Mysql para inicialización DDL con JPA entity

Paso 3) Al arrancar el servidor, automáticamente se crea la base de datos tienda en mysql, junto con la tabla productos. Podemos acceder ver el resultado accediendo con **<http://localhost/phpmyadmin/>**



Se han insertado los datos de import.sql y de data.sql (gracias a la clausula always de initialization-mode)

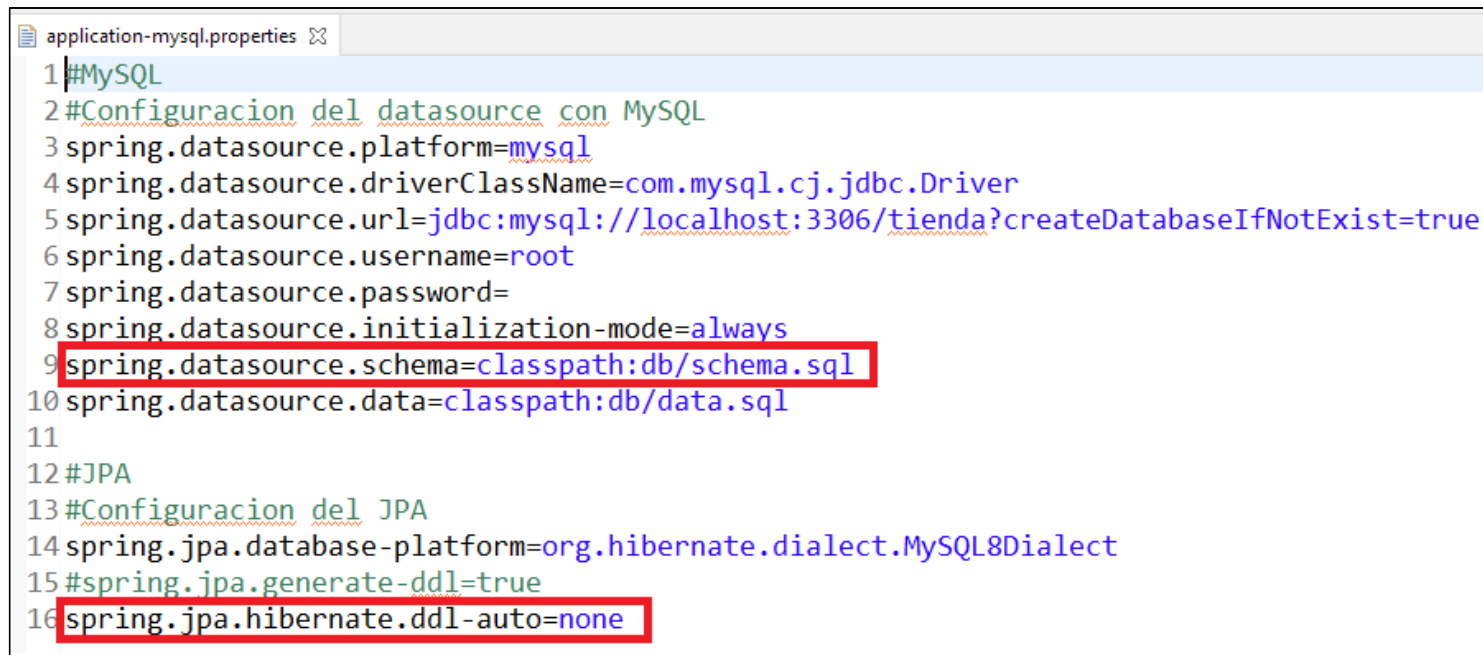
La imagen es una captura de pantalla de la interfaz de phpMyAdmin. En la parte superior, se indica el servidor (127.0.0.1), la base de datos (tienda) y la tabla seleccionada (productos). Hay una barra de herramientas con opciones como Examinar, Estructura, SQL, Buscar, Insertar, Exportar, Importar, Privilegios y Operaciones. Debajo de la barra, se muestra un mensaje de éxito: 'Mostrando filas 0 - 7 (total de 8, La consulta tardó 0,0010 segundos.)'. A continuación, se muestra la consulta SQL: 'SELECT * FROM `productos`'. Hay una barra de filtros con 'Mostrar todo', 'Número de filas' (25), 'Filtrar filas' (Buscar en esta tabla) y 'Sort by key' (Ninguna). En la parte inferior, se muestra la tabla de datos con columnas 'id' y 'name'. Cada fila tiene botones para 'Editar' y 'Borrar'.

	id	name
<input type="checkbox"/>	1	Producto 41
<input type="checkbox"/>	2	Producto 42
<input type="checkbox"/>	3	Producto 43
<input type="checkbox"/>	4	Producto 44
<input type="checkbox"/>	5	Producto 1
<input type="checkbox"/>	6	Producto 2
<input type="checkbox"/>	7	Producto 3
<input type="checkbox"/>	8	Producto 4

5. INICIALIZACION MYSQL

Configuración Mysql para inicialización DDL con schema.sql

Paso 2 bis) En este caso la inicialización DDL se realiza mediante el archivo schema.sql y la carga DML a través únicamente del archivo data.sql (no import.sql). Desactivamos la generación DDL con JPA (**spring.jpa.hibernate.ddl-auto=none**) y activamos schema.sql (**spring.datasource.schema=classpath:db/schema.sql**)

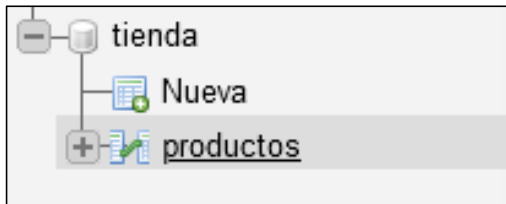
A screenshot of a text editor window titled 'application-mysql.properties'. The file contains configuration for a MySQL database. Lines 9 and 16 are highlighted with red boxes. Line 9 is 'spring.datasource.schema=classpath:db/schema.sql' and line 16 is 'spring.jpa.hibernate.ddl-auto=none'.

```
1 #MySQL
2 #Configuracion del datasource con MySQL
3 spring.datasource.platform=mysql
4 spring.datasource.driverClassName=com.mysql.cj.jdbc.Driver
5 spring.datasource.url=jdbc:mysql://localhost:3306/tienda?createDatabaseIfNotExist=true
6 spring.datasource.username=root
7 spring.datasource.password=
8 spring.datasource.initialization-mode=always
9 spring.datasource.schema=classpath:db/schema.sql
10 spring.datasource.data=classpath:db/data.sql
11
12 #JPA
13 #Configuracion del JPA
14 spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect
15 #spring.jpa.generate-ddl=true
16 spring.jpa.hibernate.ddl-auto=none
```

5. INICIALIZACION MYSQL

Configuración Mysql para inicialización DDL con schema.sql

Paso 3) Al arrancar el servidor, automáticamente se crea la base de datos tienda en mysql, junto con la tabla productos a partir del script schema.sql y con los datos de data.sql



```
schema: Bloc de notas
Archivo Edición Formato Ver Ayuda
DROP TABLE IF EXISTS PRODUCTOS;

CREATE TABLE PRODUCTOS (
  ID INTEGER PRIMARY KEY AUTO_INCREMENT,
  NAME VARCHAR(25)
);
```

```
data: Bloc de notas
Archivo Edición Formato Ver Ayuda
INSERT INTO PRODUCTOS VALUES (null, 'Producto 1');
INSERT INTO PRODUCTOS VALUES (null, 'Producto 2');
INSERT INTO PRODUCTOS VALUES (null, 'Producto 3');
INSERT INTO PRODUCTOS VALUES (null, 'Producto 4');
```

La interfaz de administración de MySQL muestra la base de datos 'tienda' y la tabla 'productos'. Se visualiza la consulta SQL: `SELECT * FROM `productos``. La consulta devuelve 4 filas de datos.

ID	NAME
1	Producto 1
2	Producto 2
3	Producto 3
4	Producto 4

6. CLASE DAO

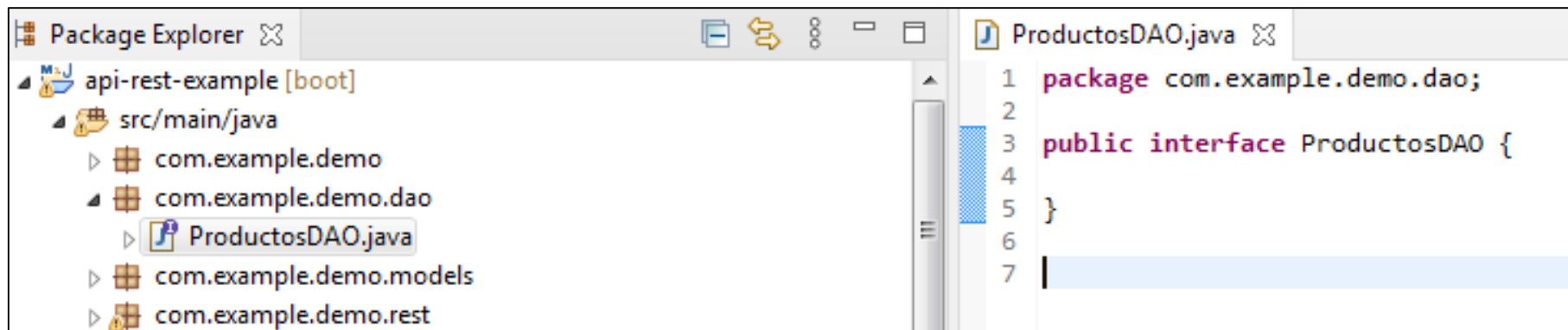
Paso 1) La clase DAO (Data Access Object) es un componente de software que suministra una interfaz común entre la aplicación y uno o más dispositivos de almacenamiento de datos, tales como una Base de datos o un archivo.

Según el Patron DAO, una vez tenemos las clases que representan nuestros datos (en nuestro caso Producto), se debe de crear una interface con los métodos necesarios para obtener y almacenar Productos. No debe tener nada que la relacione con la base de datos (sin parámetro Connection).

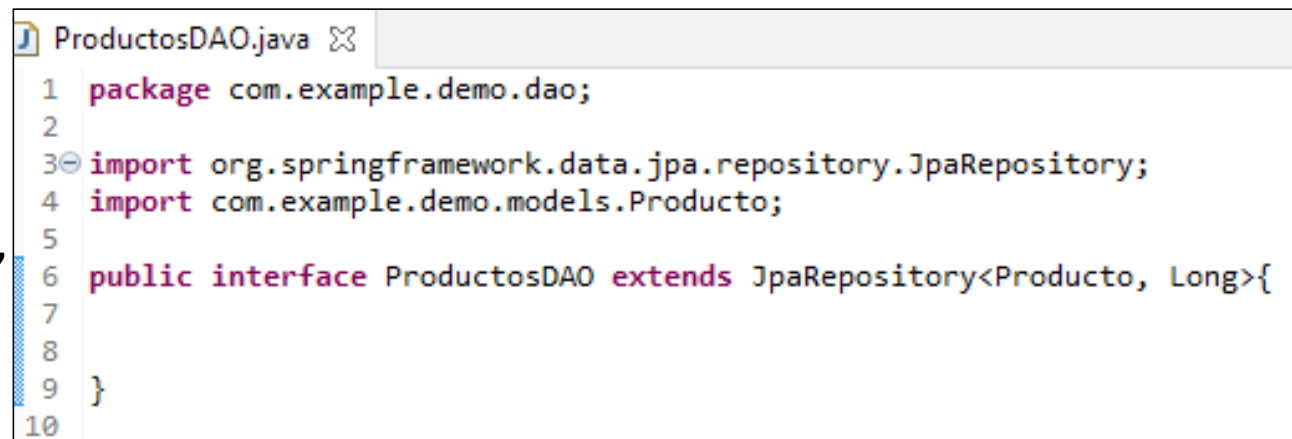
```
public interface InterfaceDAO {  
    public List<Persona> getPersonas();  
    public Persona getPersonaPorNombre (String nombre);  
    public void salvaPersona (Persona persona);  
    public void modificaPersona (Persona persona);  
    public void borraPersonaPorNombre (String nombre);  
    ...  
}
```


6. CLASE DAO

Paso 2) Crearemos un nuevo package con extensión dao y dentro de él nuestra clase dao, llamada ProductosDAO.java que hereda de JpaRepository



JpaRepository <Producto, Long>
Indica que la clase Entity es
Producto y Long es el tipo de la
clave Primaria de la tabla producto,
o que apunta la clase Producto



6. CLASE DAO

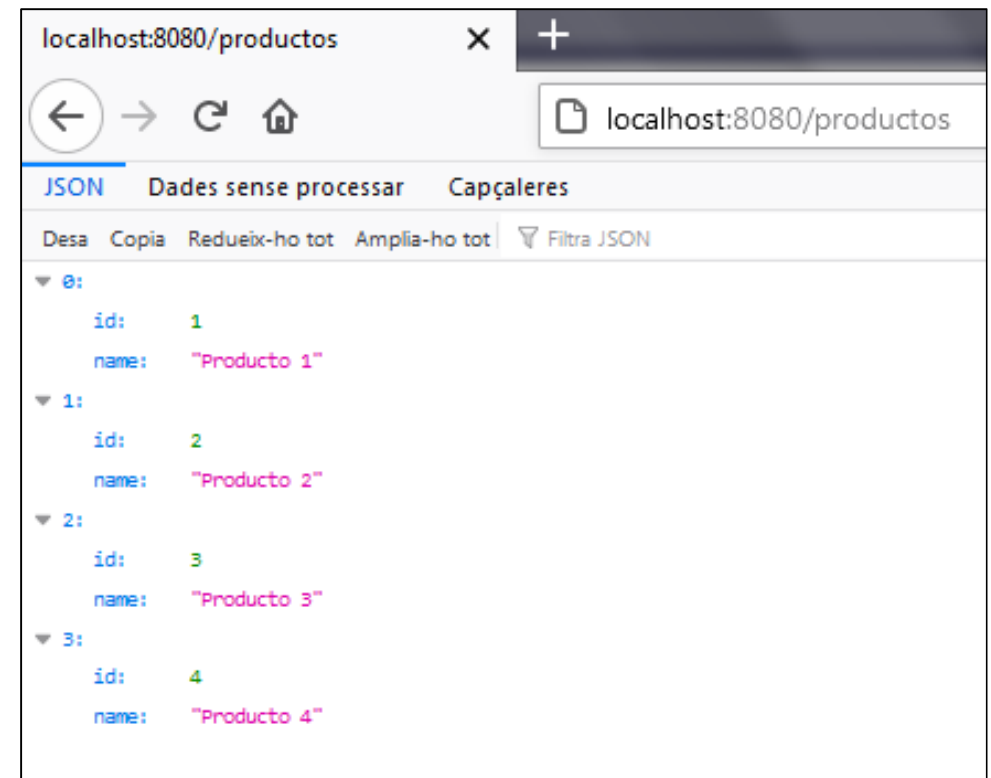
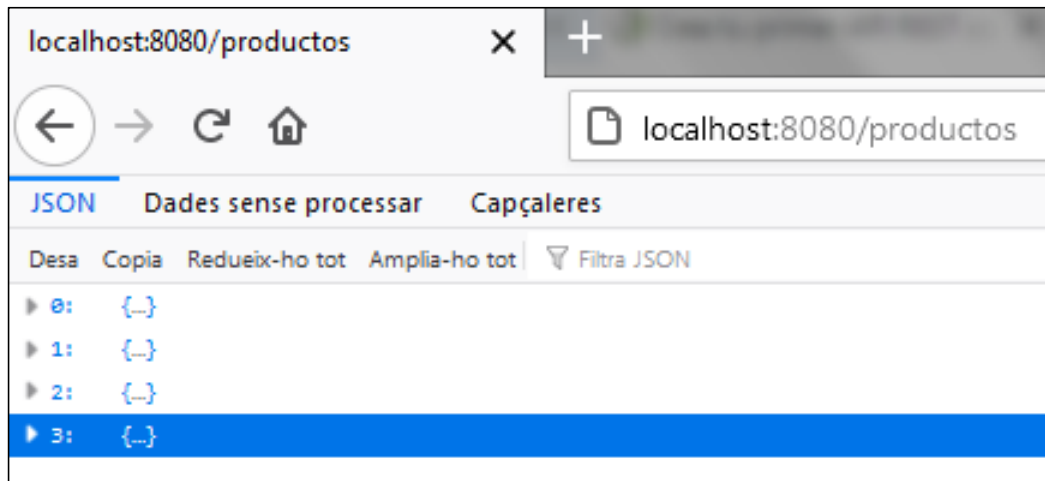
Paso 3) Una vez creada la interfaz DAO, creamos el atributo productosDAO en el controlador, sin new ProductosDAO(), sólo con la anotación @Autowired.

Esto recibe el nombre de inyección de dependencias: dejo que el sistema llame a una clase que implemente dicha interfaz y de esta manera ya podemos utilizar las funciones de dicha interfaz que se corresponde con las funciones de JpaRepository

```
ControllerRest.java
1 package com.example.demo.rest;
2
3 import java.util.List;
4
14
15 @RestController //Indica que esta clase va a ser un servicio REST
16 @RequestMapping("/productos") //En que URL se va a exponer los servicios de esta clase
17 public class ControllerRest {
18
19     @Autowired //Inyeccion de dependencias
20     private ProductosDAO productosDAO;
21
22     @GetMapping
23     public ResponseEntity<List<Producto>> getProducto() {
24         List<Producto> productos = productosDAO.findAll();
25         return ResponseEntity.ok(productos);
26     }
27
28     // @GetMapping("hello") //Servicio disponible mediante GET (localhost:8080/hello)
29     // @RequestMapping(value="hello", method=RequestMethod.GET) //En que url esta el servicio
30     public String hello() {
31         return "hello world";
32     }
33 }
34
```

7. SERVICIOS API REST

Paso 1) Podemos comprobar el servicio desde un navegador mediante la url **localhost:8080/productos** (GET):



7. SERVICIOS API REST

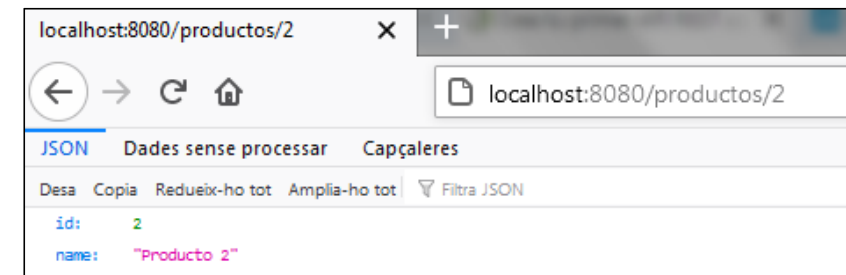
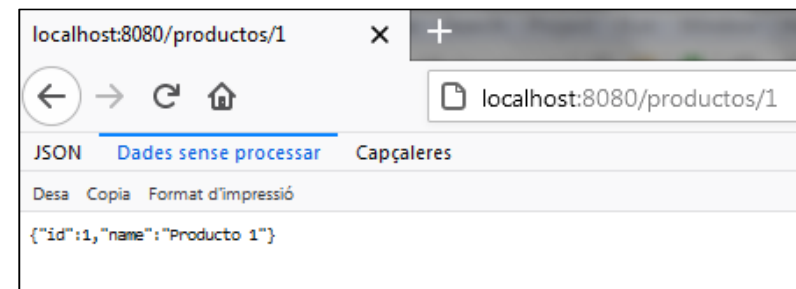
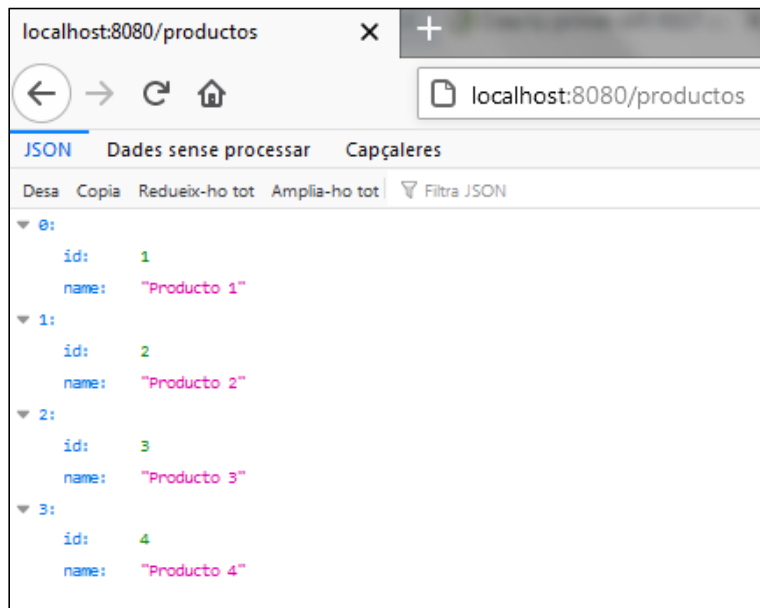
Paso 2) Creamos el servicio que nos permita leer un producto en concreto usando la anotación `@PathVariable` (GET):

Se deberá usar la url `localhost:8080/productos/{productId}` de manera que el servicio extraerá de la propia url la información sobre el producto que se desea mostrar

```
ControllerRest.java
1 package com.example.demo.rest;
2
3 import java.util.List;
4
16
17 @RestController //Indica que esta clase va a ser un servicio REST
18 @RequestMapping("/productos") //En que URL se va a exponer los servicios de esta clase
19 public class ControllerRest {
20
21     @Autowired //Inyeccion de dependencias
22     private ProductosDAO productosDAO;
23
24     @GetMapping
25     public ResponseEntity<List<Producto>> getProducto() {
26         List<Producto> productos = productosDAO.findAll();
27         return ResponseEntity.ok(productos);
28     }
29
30     @RequestMapping(value="{productId}") //productos/{productId} --> productos/1
31     public ResponseEntity<Producto> getProductoById(@PathVariable("productId") Long productId) {
32         Optional<Producto> optionalProducto = productosDAO.findById(productId);
33         if (optionalProducto.isPresent()) {
34             return ResponseEntity.ok(optionalProducto.get());
35         } else {
36             return ResponseEntity.noContent().build();
37         }
38     }
39
40     // @GetMapping("hello") //Servicio disponible mediante GET (localhost:8080/hello)
41     // @RequestMapping(value="hello", method=RequestMethod.GET) //En que url esta el servicio
42     public String hello() {
43         return "hello world";
44     }
45 }
46
```

7. SERVICIOS API REST

Paso 3) Ahora buscamos uno en concreto (GET):



7. SERVICIOS API REST

Paso 4) Ahora vamos a ver la inserción de producto (POST):

```
ControllerRest.java
1 package com.example.demo.rest;
2
3 import java.util.List;
4
19 @RestController //Indica que esta clase va a ser un servicio REST
20 @RequestMapping("/productos") //En que URL se va a exponer los servicios de esta clase
21 public class ControllerRest {
22
23     @Autowired //Inyeccion de dependencias
24     private ProductosDAO productosDAO;
25
26     @GetMapping
27     public ResponseEntity<List<Producto>> getProducto() {
28         List<Producto> productos = productosDAO.findAll();
29         return ResponseEntity.ok(productos);
30     }
31
32     @PostMapping //productos (POST)
33     public ResponseEntity<Producto> crearProducto(@RequestBody Producto producto) {
34         Producto newProduct = productosDAO.save(producto);
35         return ResponseEntity.ok(newProduct);
36     }
37
38     @RequestMapping(value="{productId}") //productos/{productId} --> productos/1
39     public ResponseEntity<Producto> getProductoById(@PathVariable("productId") Long productId) {
40         Optional<Producto> optionalProducto = productosDAO.findById(productId);
41         if (optionalProducto.isPresent()) {
42             return ResponseEntity.ok(optionalProducto.get());
43         } else {
44             return ResponseEntity.noContent().build();
45         }
46     }
47 }
```

7. SERVICIOS API REST

Paso 5) Haremos la inserción desde el plugin advanced rest client (POST):

The screenshot displays the Advanced REST Client interface. At the top, the 'Method' is set to 'POST' and the 'Request URL' is 'http://localhost:8080/productos'. A 'SEND' button is visible. Below this, the 'Request parameters' section is collapsed. The 'BODY' tab is selected, showing a 'Body content type' of 'application/json'. Below the body type, there are buttons for 'FORMAT JSON', 'MINIFY JSON', and 'COPY'. The body content is a JSON object: `{ "name": "Producto n" }`. On the right side, a response panel shows a status of '200 OK' with a response time of '119.71 ms'. Below this, there are buttons for 'COPY', 'SAVE', 'SOURCE VIEW', and 'DATA TABLE'. The response body is a JSON object: `{ "id": 5, "name": "Producto n" }`.

Method: POST
Request URL: http://localhost:8080/productos
SEND

Request parameters

HEADERS BODY AUTHORIZATION ACTIONS CONFIG CODE

Body content type: application/json

FORMAT JSON MINIFY JSON COPY

1 { "name": "Producto n" }

200 OK 119.71 ms

COPY SAVE SOURCE VIEW DATA TABLE

```
{
  "id": 5,
  "name": "Producto n"
}
```

7. SERVICIOS API REST

Paso 6) Haremos el borrado de un producto (DELETE):

```
ControllerRest.java
1 package com.example.demo.rest;
2
3 import java.util.List;
19
20 @RestController //Indica que esta clase va a ser un servicio REST
21 @RequestMapping("/productos") //En que URL se va a exponer los servicios de esta clase
22 public class ControllerRest {
23
24     @Autowired //Inyeccion de dependencias
25     private ProductosDAO productosDAO;
26
27     @GetMapping
28     public ResponseEntity<List<Producto>> getProducto() {
29         List<Producto> productos = productosDAO.findAll();
30         return ResponseEntity.ok(productos);
31     }
32
33     @PostMapping //productos (POST)
34     public ResponseEntity<Producto> crearProducto(@RequestBody Producto producto) {
35         Producto newProduct = productosDAO.save(producto);
36         return ResponseEntity.ok(newProduct);
37     }
38
39     @DeleteMapping(value="{productoId}") //productos/{productId} (DELETE)
40     public ResponseEntity<Void> deleteProducto(@PathVariable("productoId") Long productoId) {
41         productosDAO.deleteById(productoId);
42         return ResponseEntity.ok(null);
43     }
44 }
```


7. SERVICIOS API REST

Paso 7) Realizamos la comprobación via ARC (DELETE):

Method: DELETE Request URL: http://localhost:8080/productos/3

Request parameters

HEADERS BODY AUTHORIZATION ACTIONS CONFIG CODE

COPY SOURCE VIEW

	Header name	Header value
<input type="checkbox"/>	Content-Type	application/json

+ ADD HEADER

200 OK 237.56 ms

DETAILS

localhost:8080/productos

```
[{"id":1,"name":"Producto 1"}, {"id":2,"name":"Producto 2"}, {"id":4,"name":"Producto 4"}, {"id":5,"name":"Producto n"}]
```

7. SERVICIOS API REST

Paso 8) Haremos la actualización de un producto (PUT):

```
ControllerRest.java
29 public ResponseEntity<List<Producto>> getProducto() {
30     List<Producto> productos = productosDAO.findAll();
31     return ResponseEntity.ok(productos);
32 }
33
34 @PostMapping //productos (POST)
35 public ResponseEntity<Producto> crearProducto(@RequestBody Producto producto) {
36     Producto newProduct = productosDAO.save(producto);
37     return ResponseEntity.ok(newProduct);
38 }
39
40 @DeleteMapping(value="{productoId}") //productos/{productId} (DELETE)
41 public ResponseEntity<Void> deleteProducto(@PathVariable("productoId") Long productId) {
42     productosDAO.deleteById(productId);
43     return ResponseEntity.ok(null);
44 }
45
46 @PutMapping //productos/{productId} --> productos/1
47 public ResponseEntity<Producto> updateProducto(@RequestBody Producto producto) {
48     Optional<Producto> optionalProducto = productosDAO.findById(producto.getId());
49     if (optionalProducto.isPresent()) {
50         Producto updateProducto = optionalProducto.get();
51         updateProducto.setName(producto.getName());
52         productosDAO.save(updateProducto);
53         return ResponseEntity.ok(updateProducto);
54     } else {
55         return ResponseEntity.notFound().build();
56     }
57 }
58 }
```

7. SERVICIOS API REST

Paso 9) Realizamos la actualización de un producto via ARC (PUT):

Method
PUT

Request URL
http://localhost:8080/productos

SEND

Request parameters

HEADERS

BODY

AUTHORIZATION

ACTIONS

CONFIG

CODE

Body content type
application/json

FORMAT JSON

MINIFY JSON

COPY

```
1 {  
2   "id": 1,  
3   "name": "Producto Iphone"  
4 }
```

< > ↺ ☐ | 🌐 localhost:8080/productos

```
[{"id":1,"name":"Producto Iphone"}, {"id":2,"name":"Producto 2"}, {"id":4,"name":"Producto 4"}, {"id":5,"name":"Producto n"}]
```

9. PRACTICA

Tenim una franquícia de botiga de quadres il·legals que fa veure que ven collarets blancs, per això es diu “white collar”. Aquesta franquícia té varies botigues, cadascuna amb un nom i una capacitat màxima de quadres (o collars^^). Hi ha quadres que tenen un nom d'autor i d'altres que són anònims. Això sí, tots tenen un nom, un preu i una data d'entrada (és la data del moment en el que entra a la botiga). El client ens demana implementar una API amb Spring amb les següents funcionalitats:

- Crear botiga: li direm el nom i la capacitat (POST /shops/).
- Llistar botigues: retorna la llista de botigues amb el seu nom i la capacitat (GET /shops/).
- Afegir quadre: li donarem el nom del quadre i el del autor (POST /shops/{ID}/pictures)
- Llistar els quadres de la botiga (GET /shops/{ID}/pictures).
- Incendiar quadres: per si ve la policia, es poden eliminar tots els quadres de la botiga sense deixar rastre (DELETE /shops/{ID}/pictures).

9. PRACTICA

NOTES

Has de tindre en compte els següents detalls de construcció:

- Dissenya la base de dades com a primer pas. Utilitza JPA per accedir-hi.
- Inclou els scripts sql de creació de taules en el projecte, o, si ho prefereixes, configura JPA per crear automàticament les taules a l'arrencar el microservicio
- Inclou en un directori del projecte la col·lecció Postman per a les invocacions http
- La configuració del projecte ha d'estar en un fitxer application.properties
- Per a consultes a mysql, utilitzeu MysqlWorkbench.

- Fase 1

Has de persistir les dades en una base de dades en memòria h2 (inclosa en Spring boot)

- Fase 2

Utilitza mysql per persistir les dades, en lloc de h2

Ambdós orígens de dades hauran de modificar-se en application.properties.