

---

# **CREACION DE UN SERVICIO API REST SENCILLO**

**EDUARD LARA**

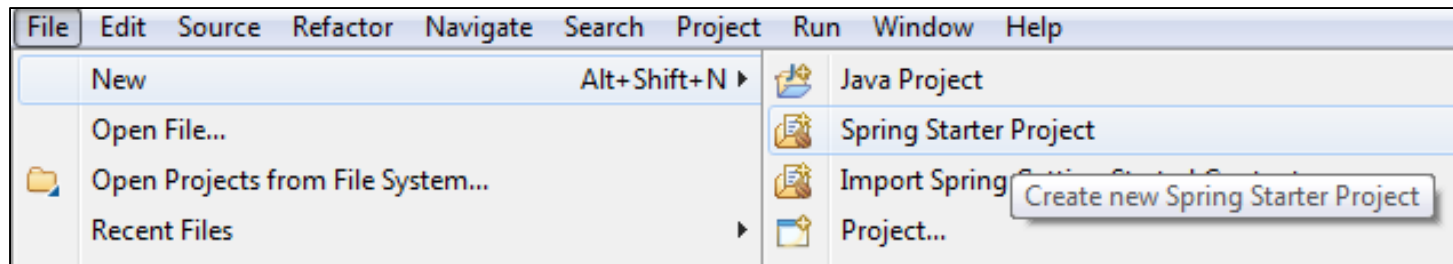
# INDICE

---

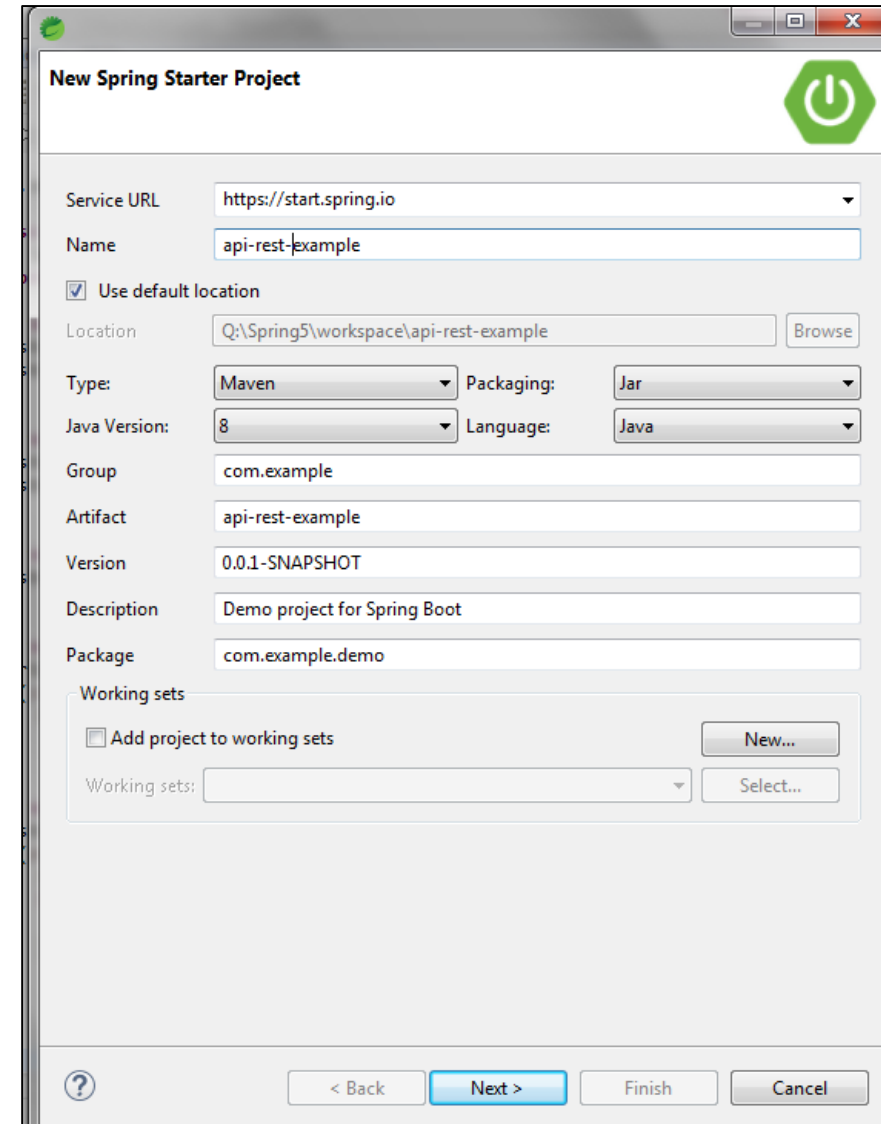
1. Creación proyecto
2. Controlador Rest
3. Modelo de datos
4. Inicialización H2
5. Inicializacion Mysql
6. Clase DAO
7. Servicios API REST
8. Onetomany
9. Métodos derivados JPA

# 1. CREACION PROYECTO

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



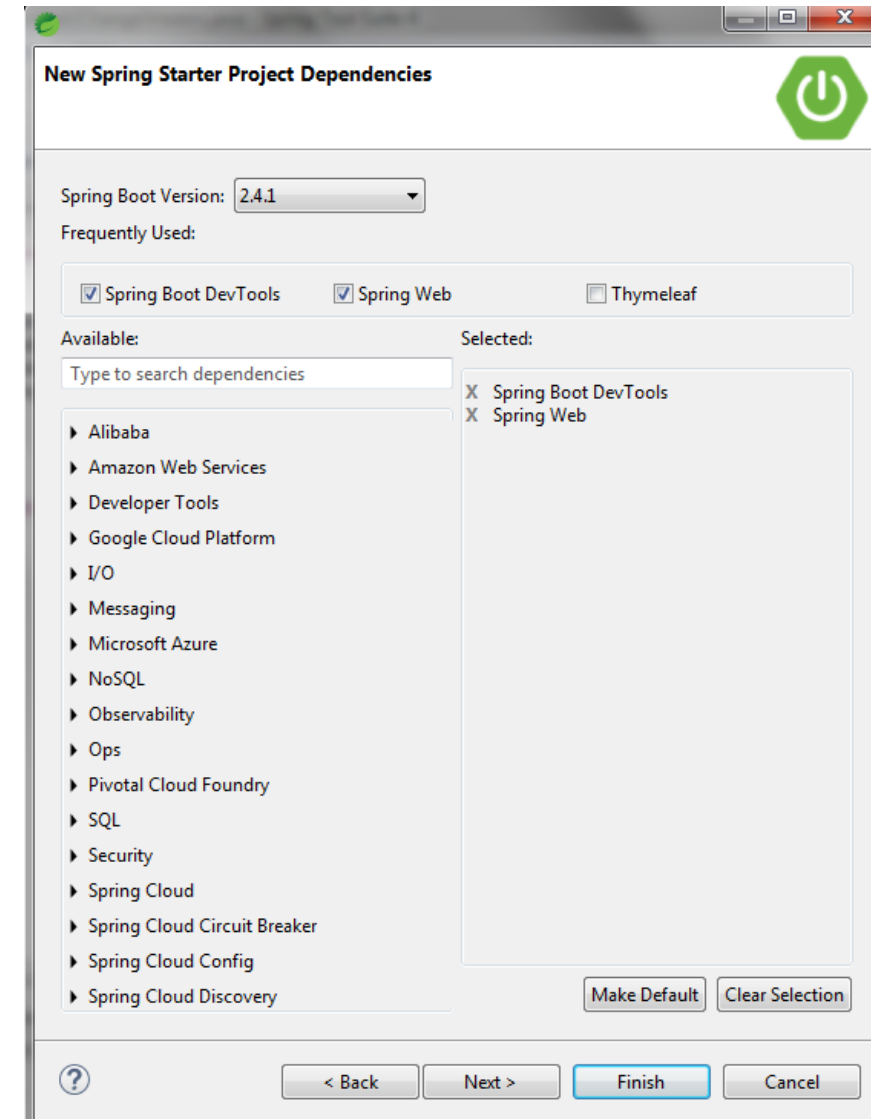
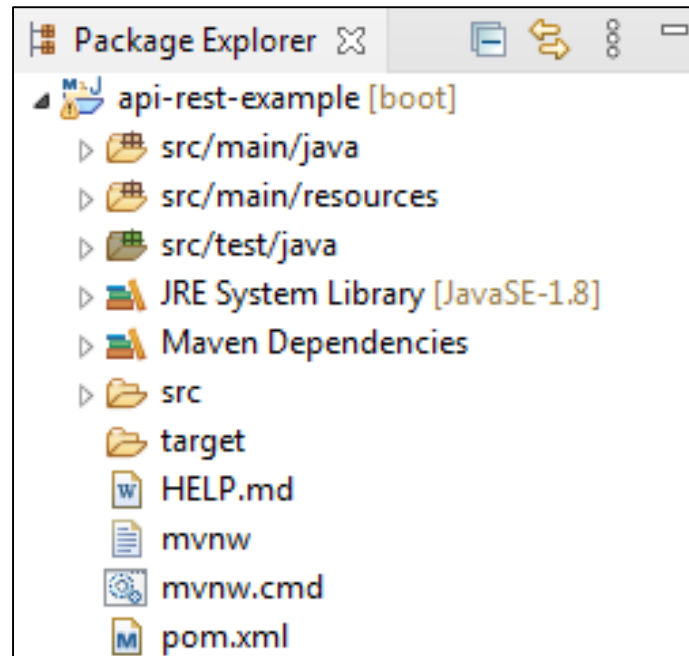
Podemos dejar por defecto los valores que nos presenta el wizard. Si se desea se puede cambiar el nombre de proyecto, el package raíz, el tipo de proyecto (Maven o Gradle) y/o la versión de Java.



# 1. CREACION PROYECTO

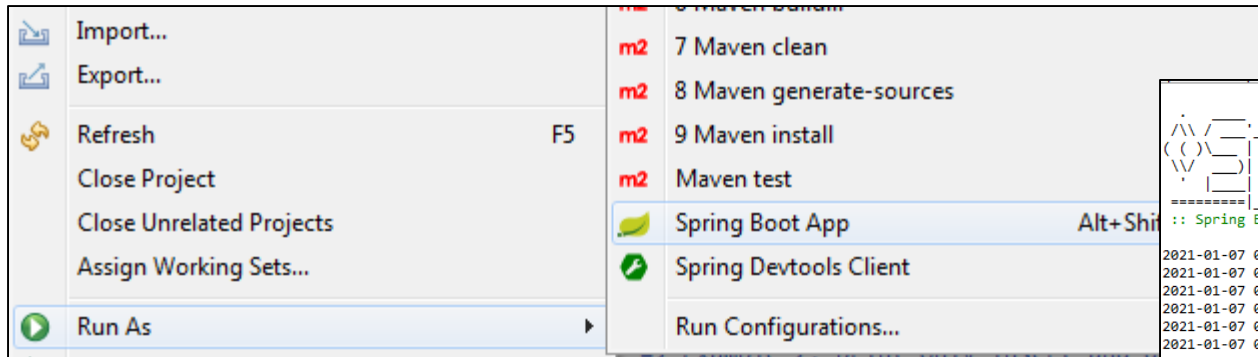
**Paso 2)** Agregamos las librerías:

- 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

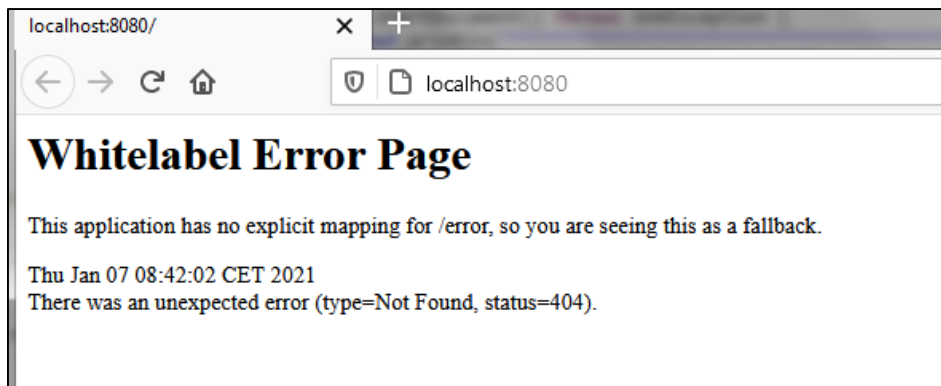
**Paso 3)** Probamos de ejecutar el proyecto, para ello levantamos el servidor Tomcat haciendo Run As/Spring Boot App. Una vez vemos que ha arrancado correctamente el servidor, vamos a un navegador y ponemos **localhost:8080**. Nos da error porque no tenemos ninguna página de inicio. Pero también significa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



```

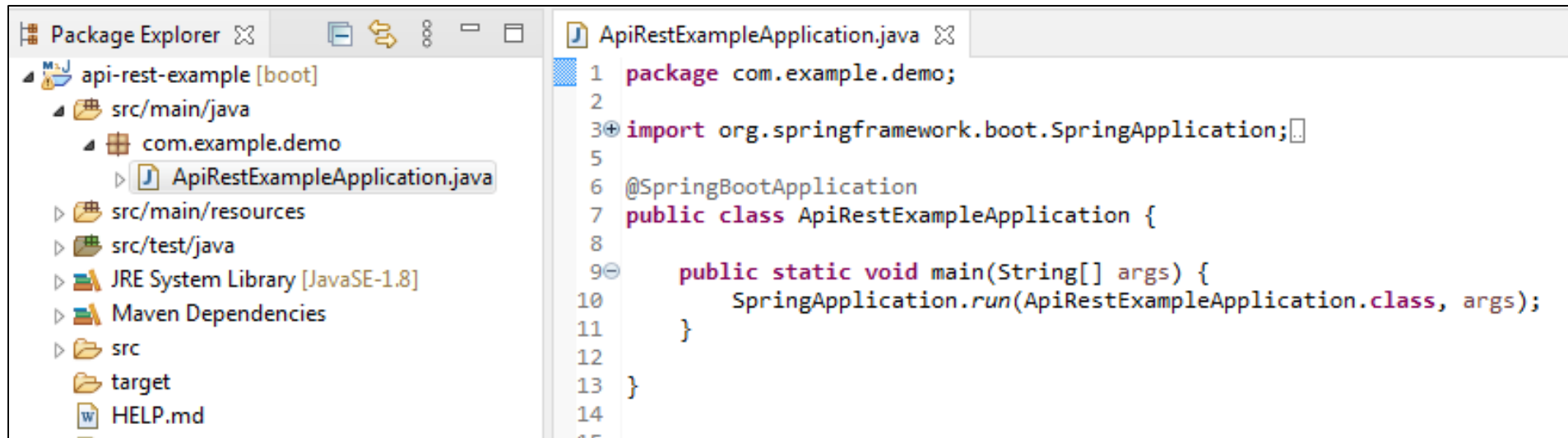
:: Spring Boot ::
(v2.4.1)

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 pro
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] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2021-01-07 08:41:47.375 INFO 16080 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization complet
2021-01-07 08:41:47.569 INFO 16080 --- [main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecu
2021-01-07 08:41:47.788 INFO 16080 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with contex
2021-01-07 08:41:47.799 INFO 16080 --- [main] 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 'dispatcherS
2021-01-07 08:42:02.577 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
```



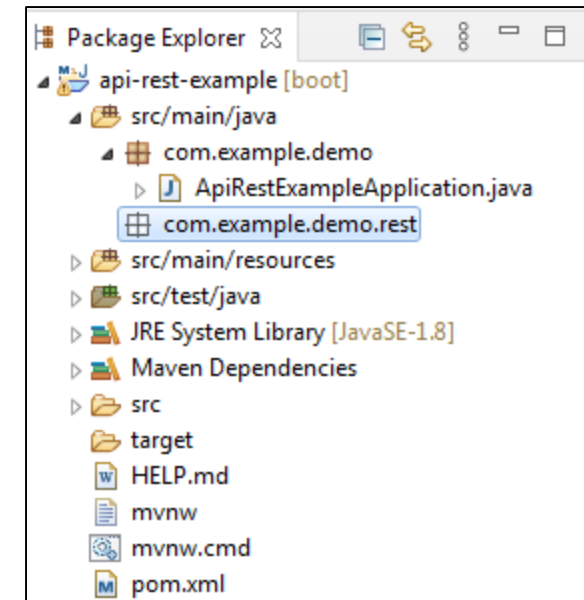
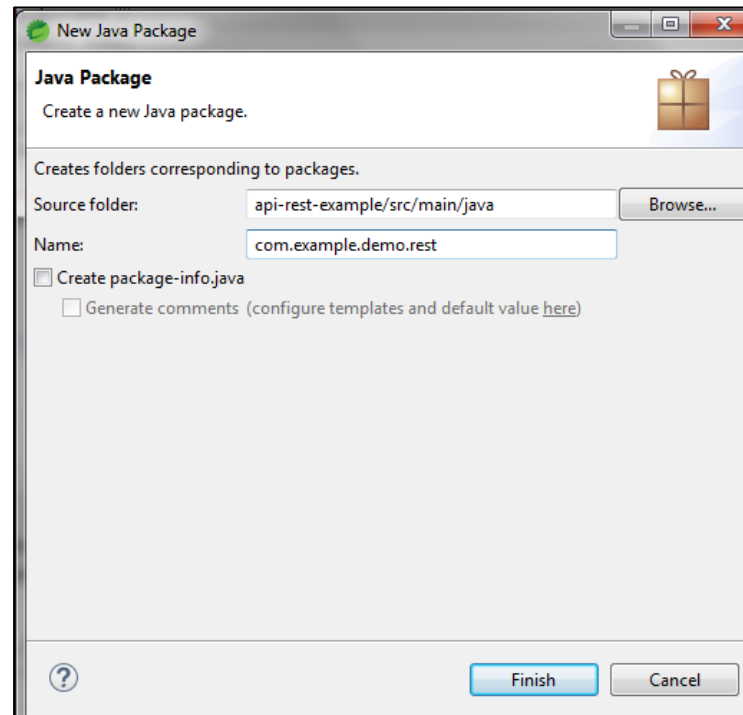
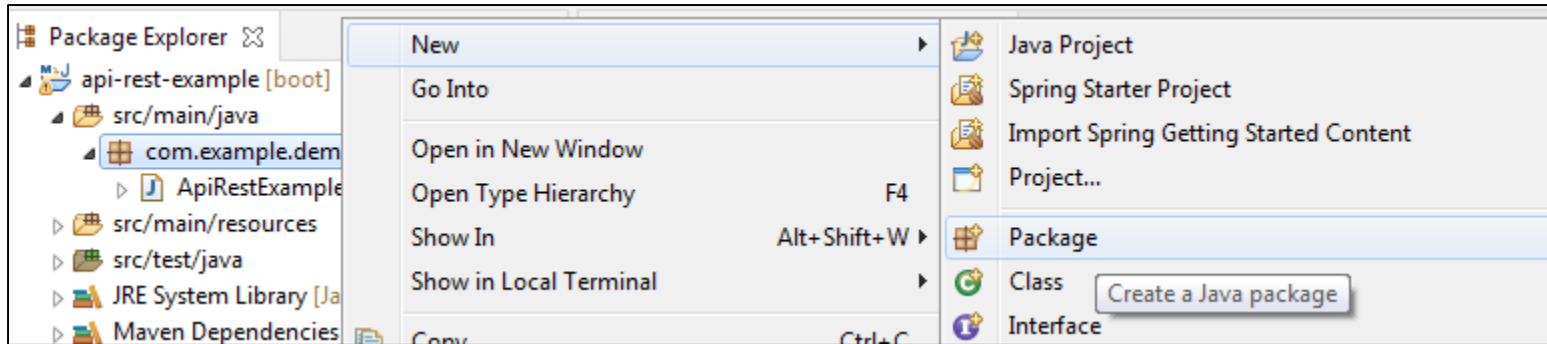
# 1. CREACION PROYECTO

**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:



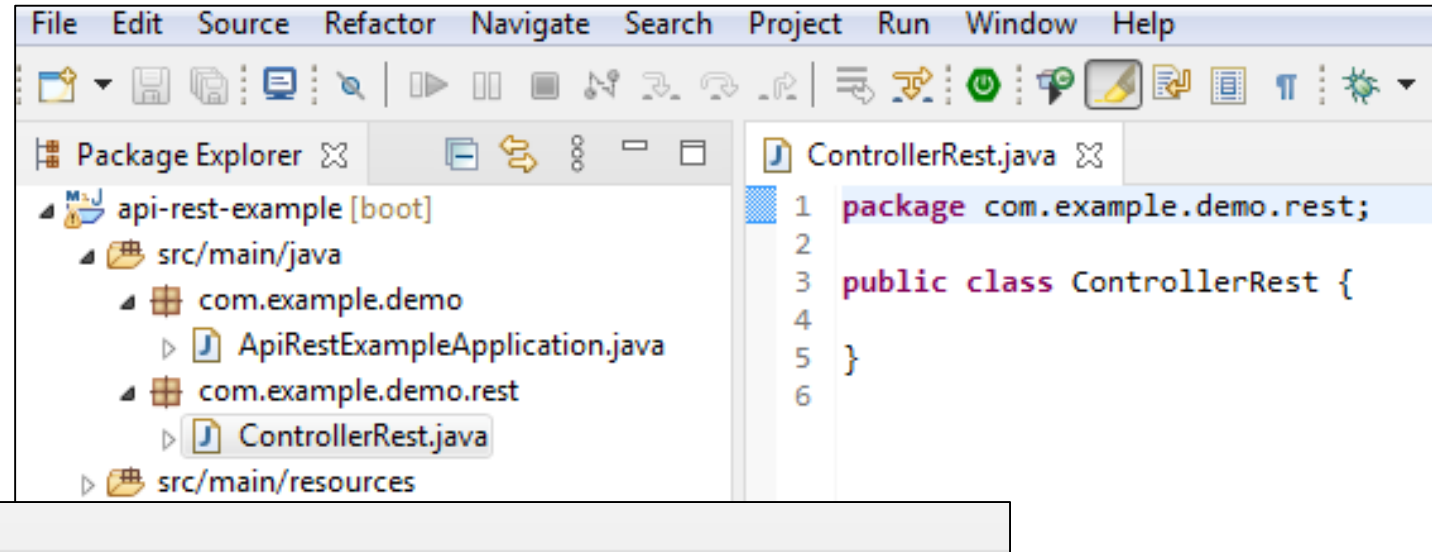
## 2. CONTROLADOR REST

**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:



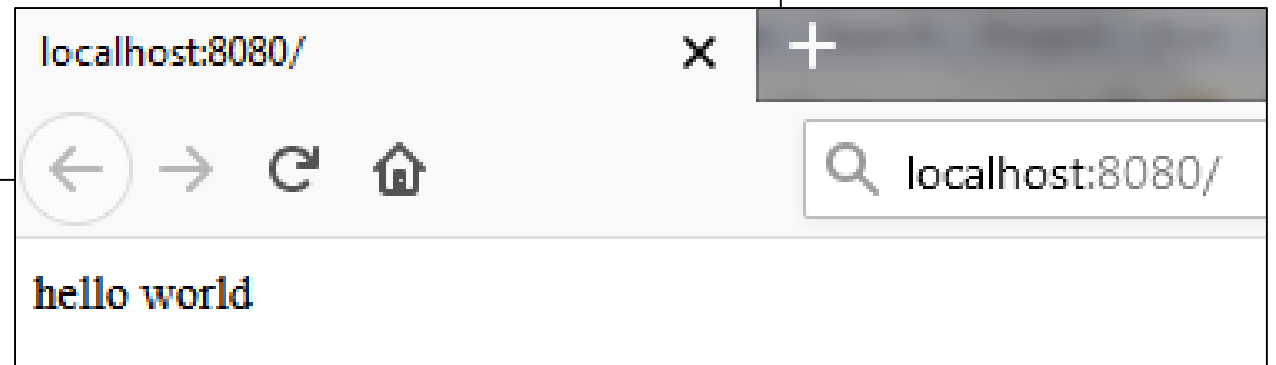
```
1 package com.example.demo.rest;
2
3 import org.springframework.web.bind.annotation.RequestMapping;
4
5
6 @RestController //Indica que esta clase va a ser un servicio REST
7 @RequestMapping("/") //En que URL se va a exponer los servicios de esta clase
8 public class ControllerRest {
9
10 }
```



## 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/

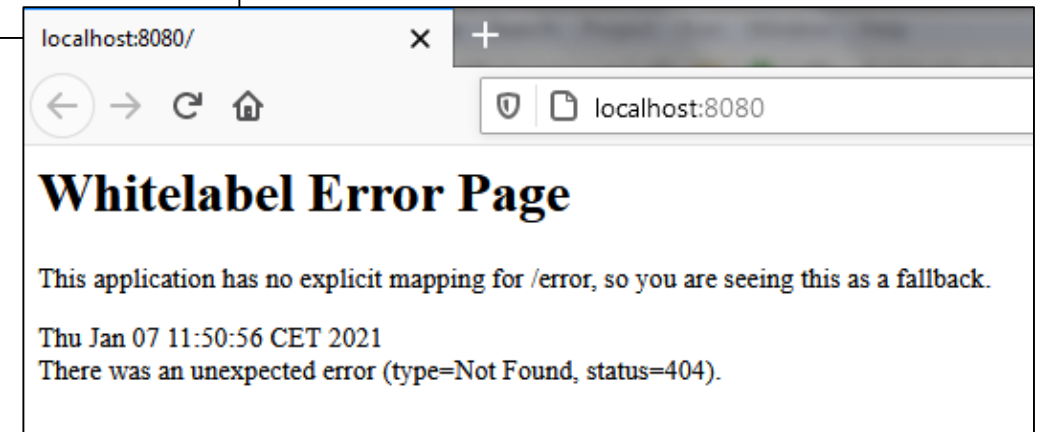
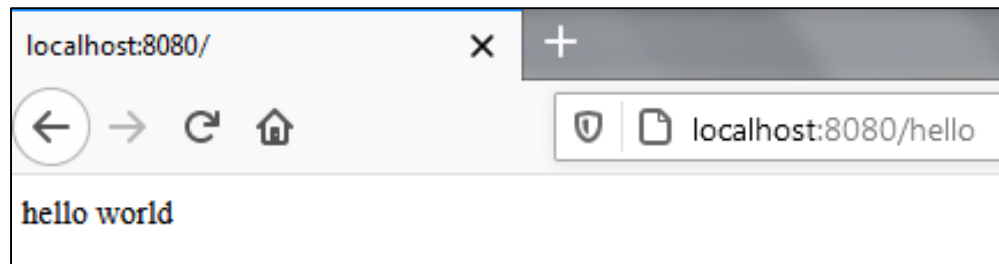
```
ControllerRest.java
1 package com.example.demo.rest;
2
3 import org.springframework.web.bind.annotation.GetMapping;
4
5
6
7
8 @RestController           //Indica que esta clase va a ser un servicio REST
9 @RequestMapping("/")     //En que URL se va a exponer los servicios de esta clase
10 public class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



## 2. CONTROLADOR REST

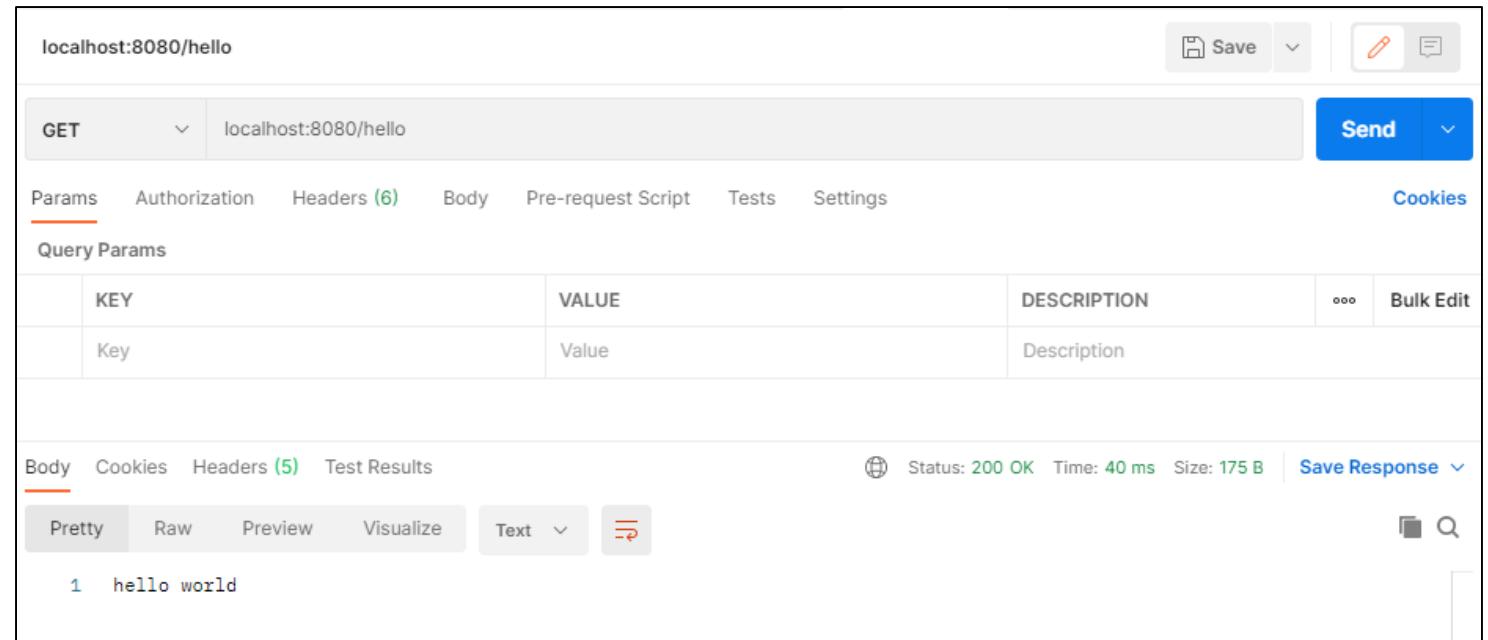
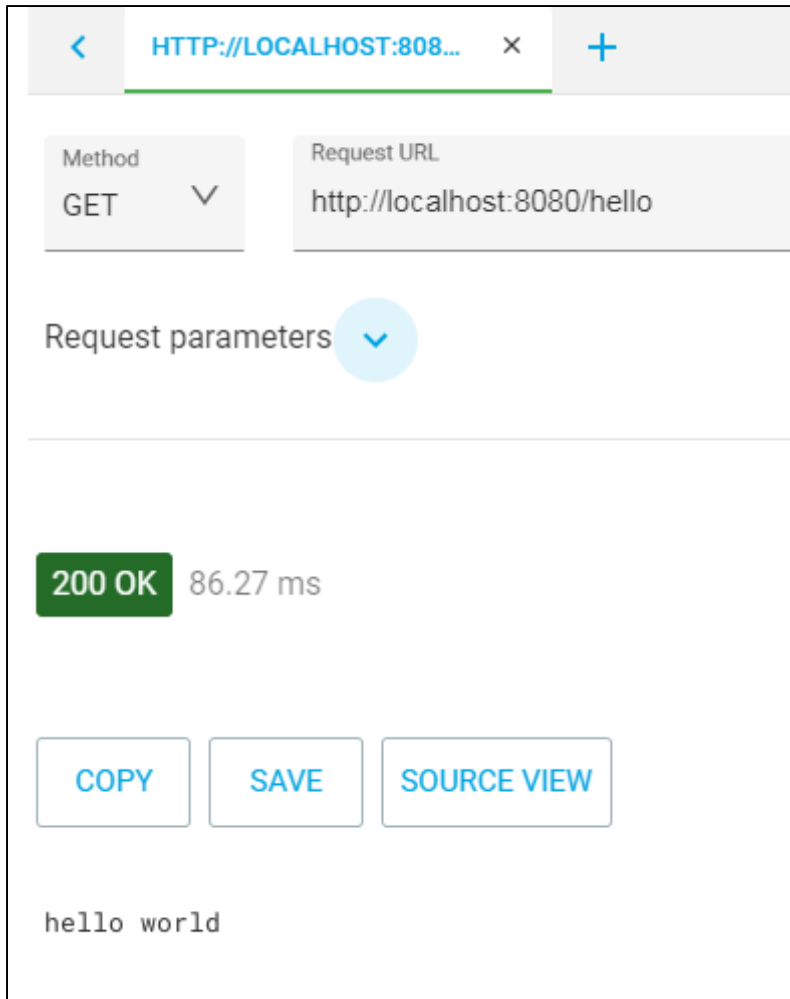
**Paso 4)** Si añadimos en el GetMapping el path “hello”, entonces la función daría servicio en la url localhost:8080/hello:

```
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 }
```



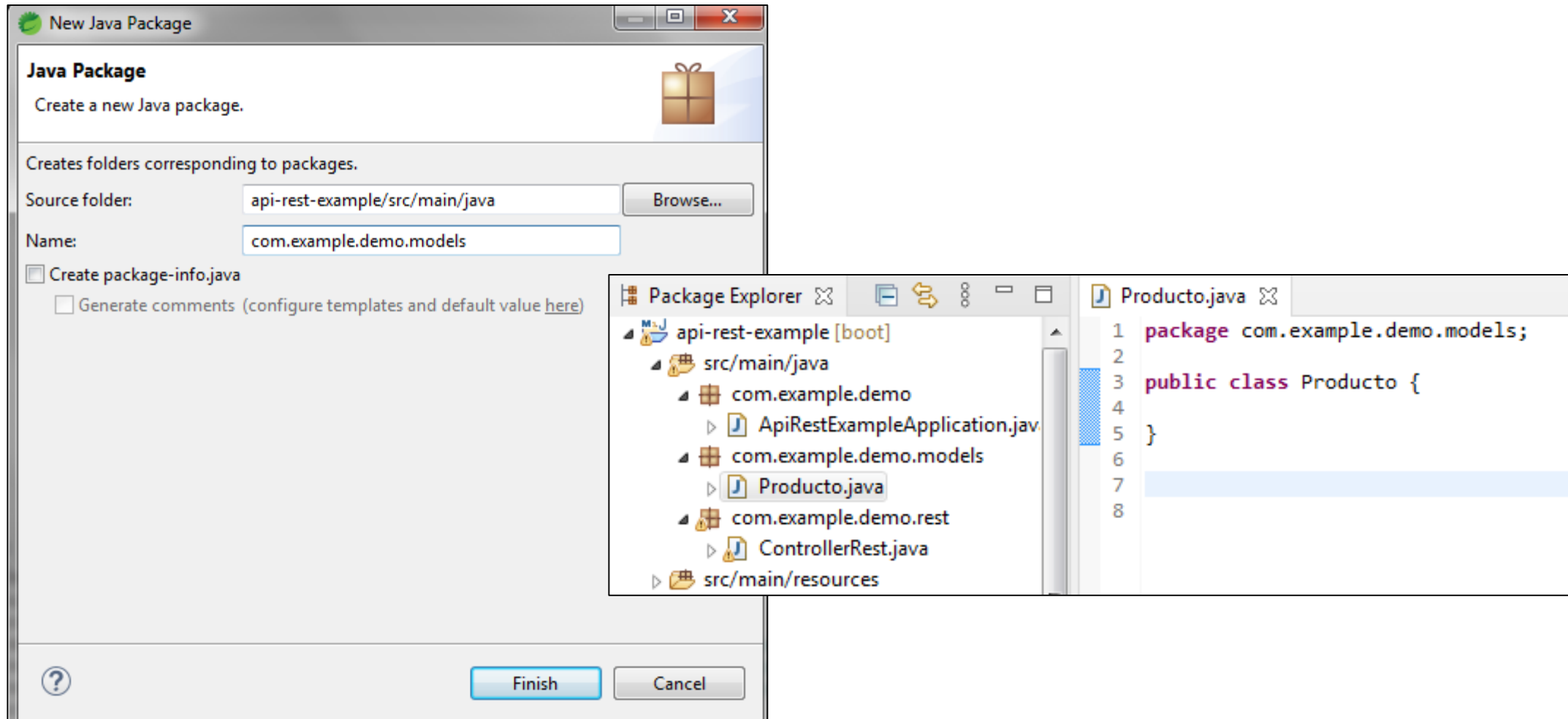
## 2. CONTROLADOR REST

**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

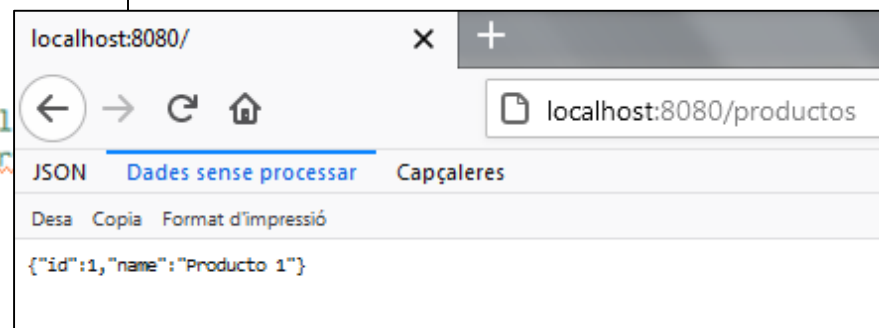
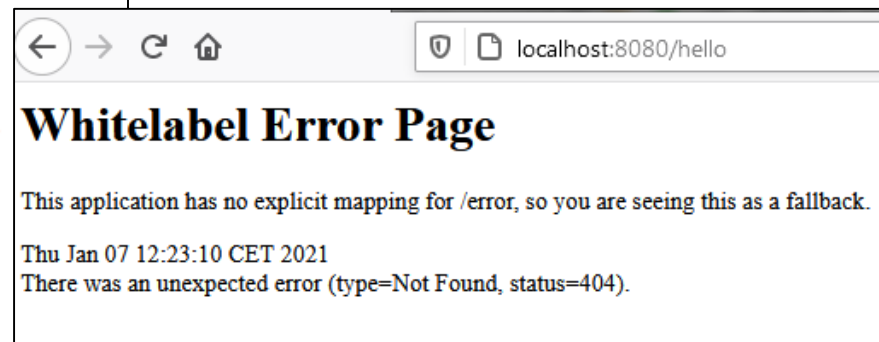
**Paso 2)** Creamos dos atributos simples en la clase POJO Producto, la cual representará el modelo de los datos de la base de datos:

```
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 servicio
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 a JSON object: `{ "id": 1, "name": "Producto 1" }`. At the bottom, there are buttons for **COPY**, **SAVE**, **SOURCE VIEW**, and **DATA TABLE**.

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 a JSON object: `{ "id": 1, "name": "Producto 1" }`. At the bottom, there are buttons for **COPY**, **SAVE**, **SOURCE VIEW**, and **DATA TABLE**.

# 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:

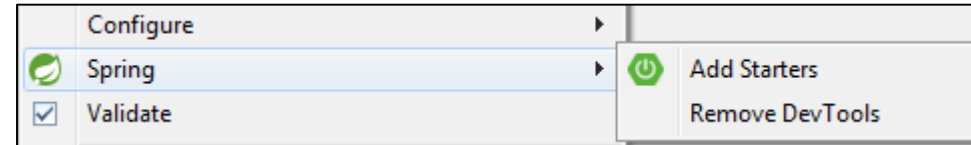
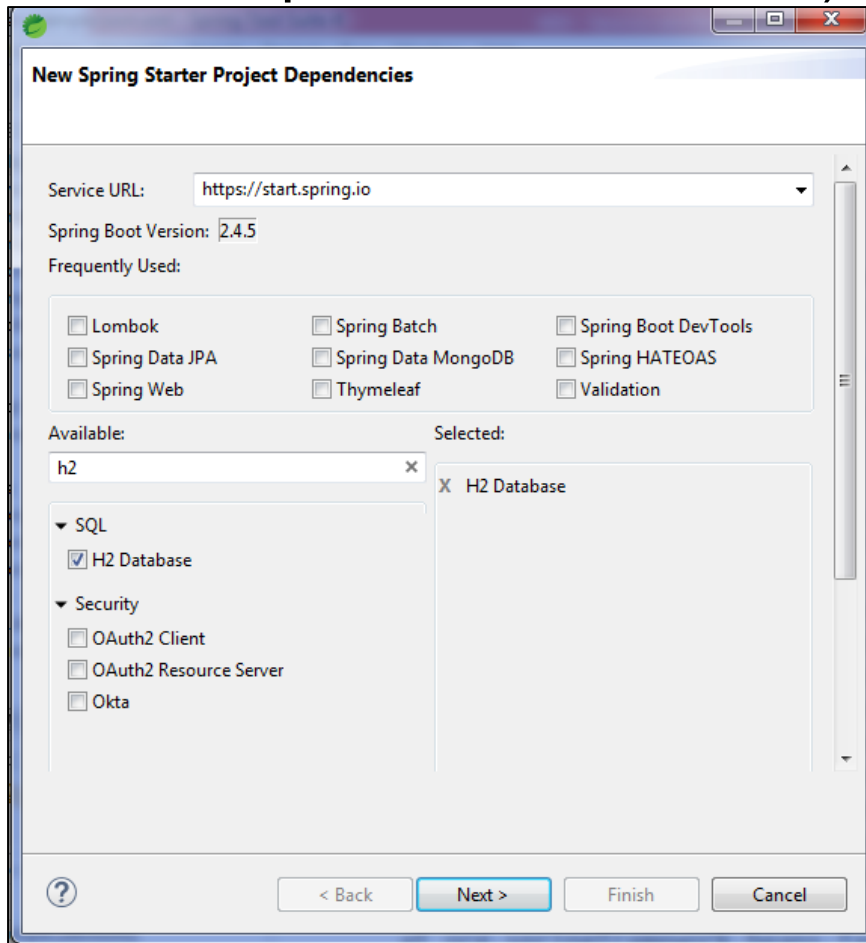
The image illustrates the steps to add Spring Data JPA as a starter dependency in an IDE. It consists of three main parts:

- Context Menu:** A screenshot of a right-click context menu over a project. The 'Spring' option is selected, and a sub-menu is open showing 'Add Starters' and 'Remove DevTools'.
- pom.xml Snippet:** A screenshot of the `api-rest-example/pom.xml` file. The following XML snippet is highlighted with a red box:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```
- New Spring Starter Project Dependencies Dialog:** A screenshot of the 'New Spring Starter Project Dependencies' dialog. The 'Service URL' is set to `https://start.spring.io` and the 'Spring Boot Version' is `2.4.5`. Under 'Frequently Used', 'Spring Data JPA' is checked. In the 'Available' list, 'Spring Data JPA' is selected. The 'Next' button is highlighted.

## 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

---

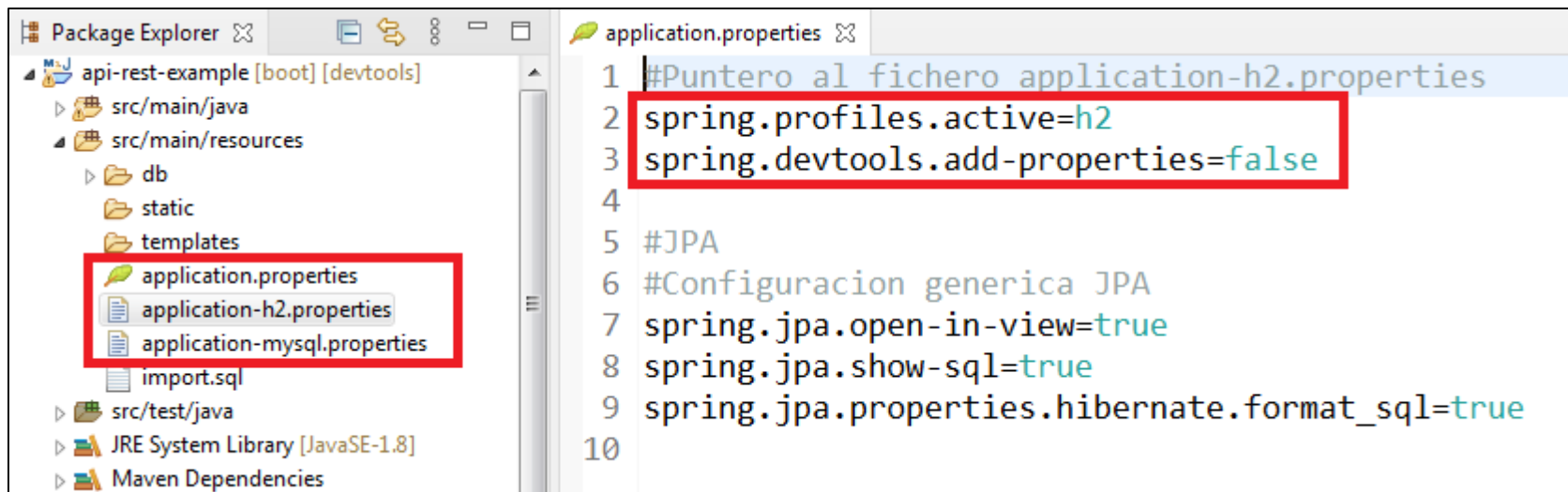
- Spring permite inicializar la base de datos de las siguientes formas:
  - Inicialización DDL → Usando la configuración de las clases entity (Hibernate JPA) o Mediante el fichero externo schema.sql
  - Inicialización DML → Mediante los ficheros import.sql y/o data.sql
- La directiva **spring.jpa.hibernate.ddl-auto** 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

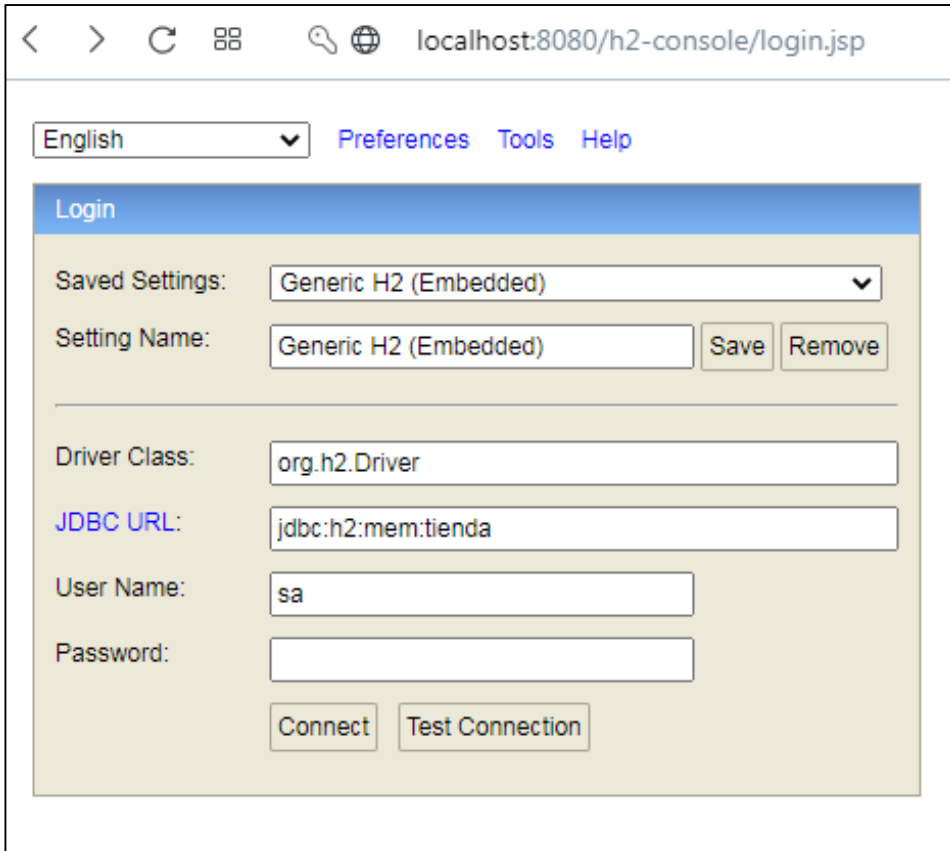
Archivo	Edición	Formato	Ver	Ayuda
INSERT INTO PRODUCTOS VALUES (null, 'Producto 41');				
INSERT INTO PRODUCTOS VALUES (null, 'Producto 42');				
INSERT INTO PRODUCTOS VALUES (null, 'Producto 43');				
INSERT INTO PRODUCTOS VALUES (null, 'Producto 44');				



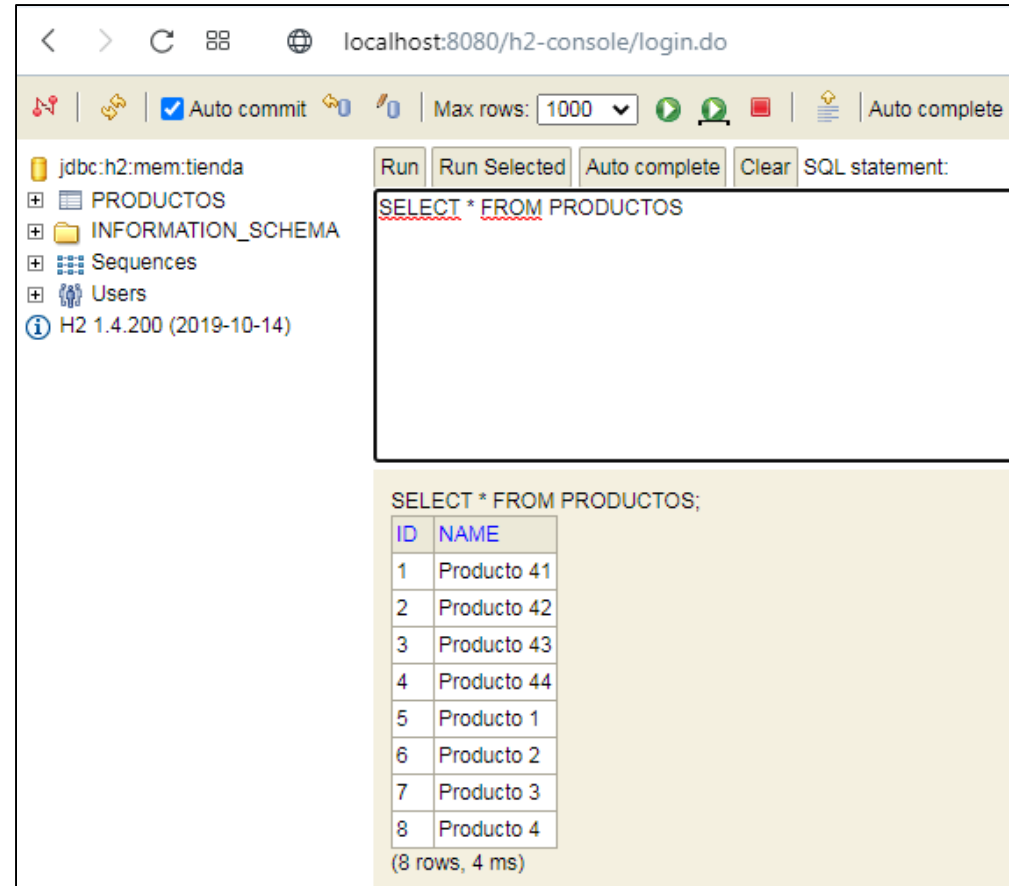
# 4. INICIALIZACION H2

## 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 navigation bar with 'English', 'Preferences', 'Tools', and 'Help'. Below this is a 'Login' section with a 'Saved Settings' dropdown set to 'Generic H2 (Embedded)'. The 'Setting Name' is also 'Generic H2 (Embedded)', with 'Save' and 'Remove' buttons. The 'Driver Class' is 'org.h2.Driver', the 'JDBC URL' is 'jdbc:h2:mem:tienda', the 'User Name' is 'sa', and the 'Password' field is empty. 'Connect' and 'Test Connection' buttons are at the bottom.



The screenshot shows the H2 console main page. The browser address bar displays 'localhost:8080/h2-console/login.do'. The page has a toolbar with 'Auto commit', 'Max rows: 1000', and 'Auto complete'. The left sidebar shows the database structure: 'jdbc:h2:mem:tienda' (expanded), 'PRODUCTOS', 'INFORMATION\_SCHEMA', 'Sequences', and 'Users'. Below this is 'H2 1.4.200 (2019-10-14)'. The main area shows the SQL statement 'SELECT \* FROM PRODUCTOS' and the execution results.

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

(8 rows, 4 ms)

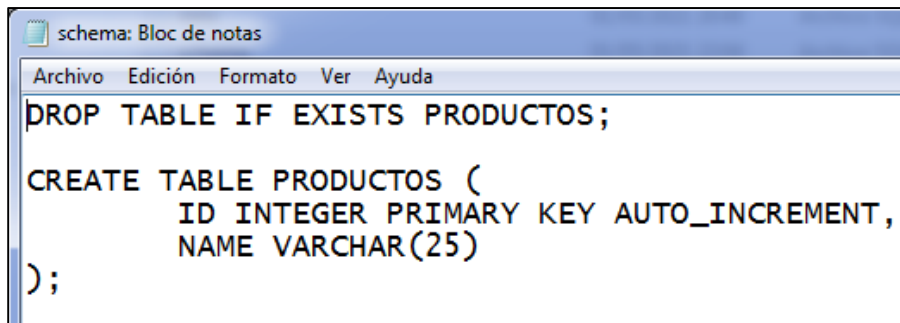


# 4. INICIALIZACION H2

## Configuración H2 para inicialización DDL con schema.sql

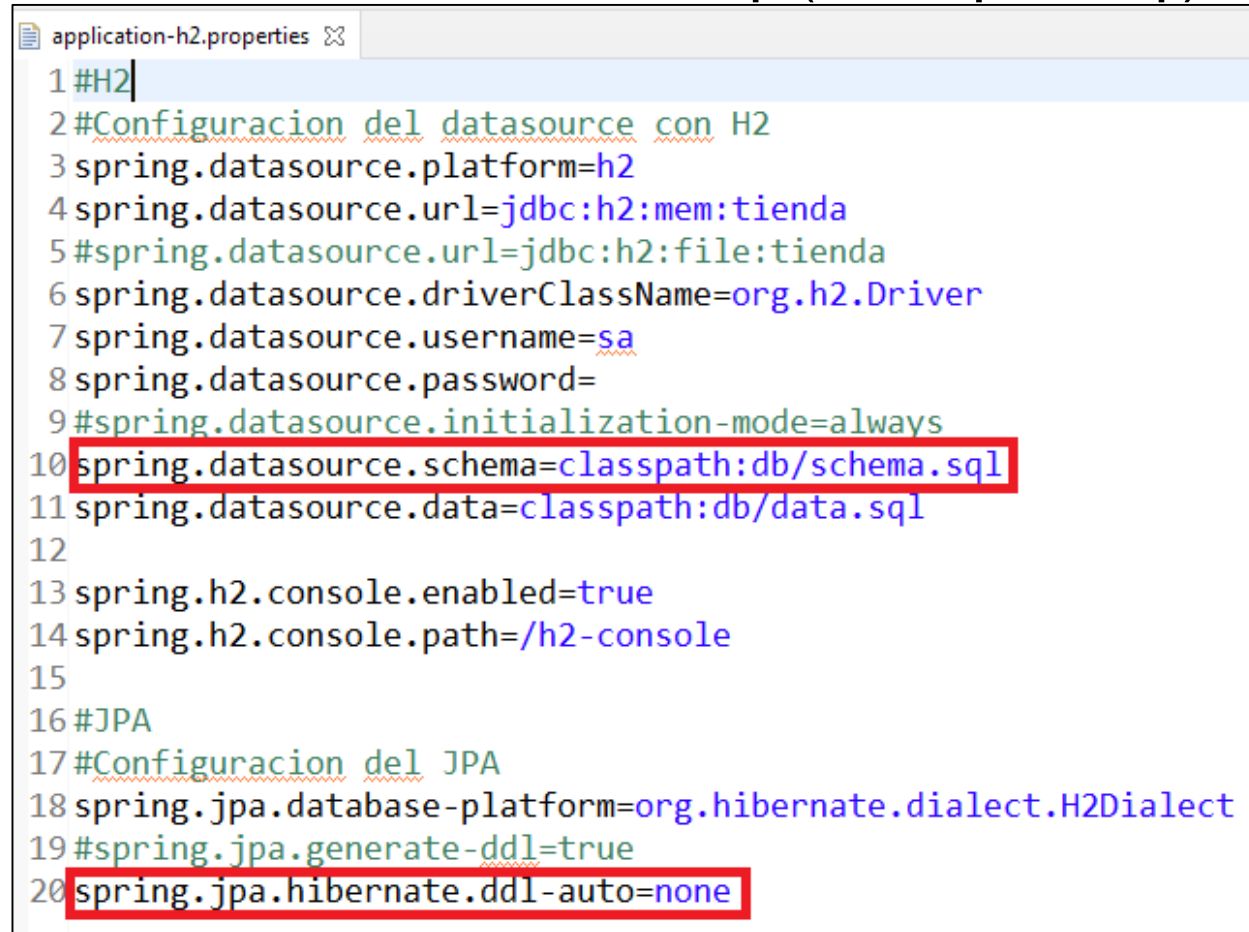
**Paso 4 bis)** En este caso la inicialización DDL se realiza mediante el archivo schema.sql y la carga DML a través únicamente del archivo data.sql (no import.sql).

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)
);
```

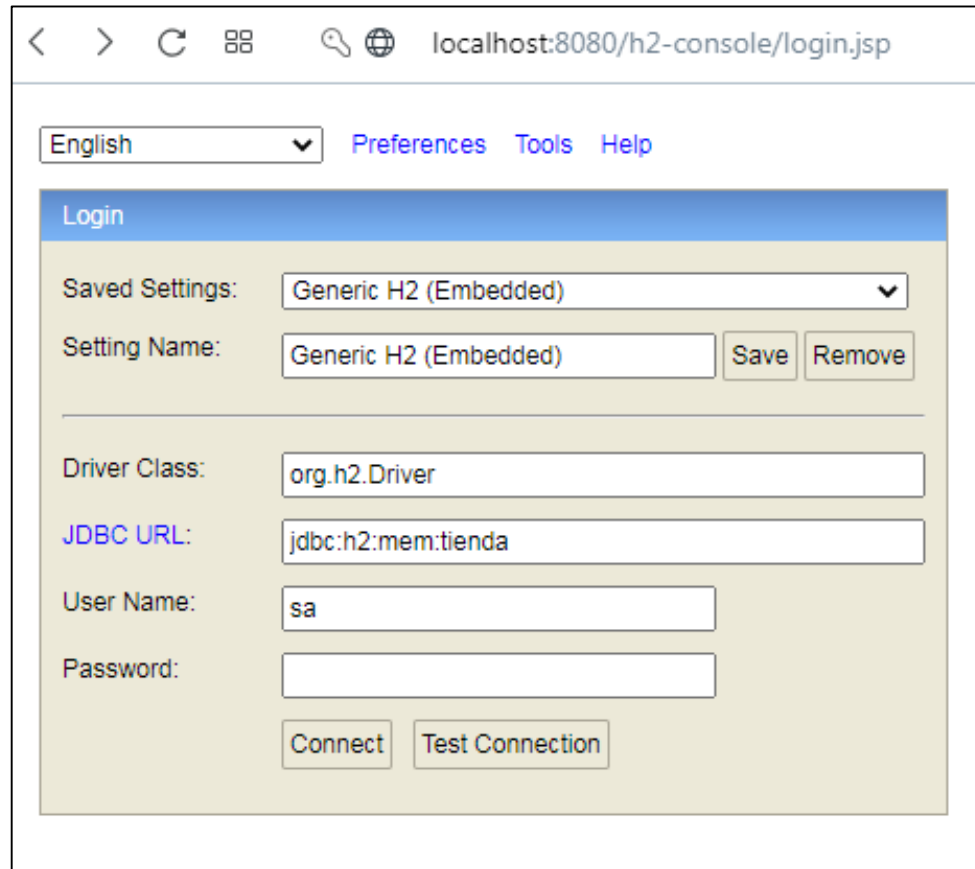


```
application-h2.properties
1 #H2
2 #Configuracion del datasource con H2
3 spring.datasource.platform=h2
4 spring.datasource.url=jdbc:h2:mem:tienda
5 #spring.datasource.url=jdbc:h2:file:tienda
6 spring.datasource.driverClassName=org.h2.Driver
7 spring.datasource.username=sa
8 spring.datasource.password=
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
```

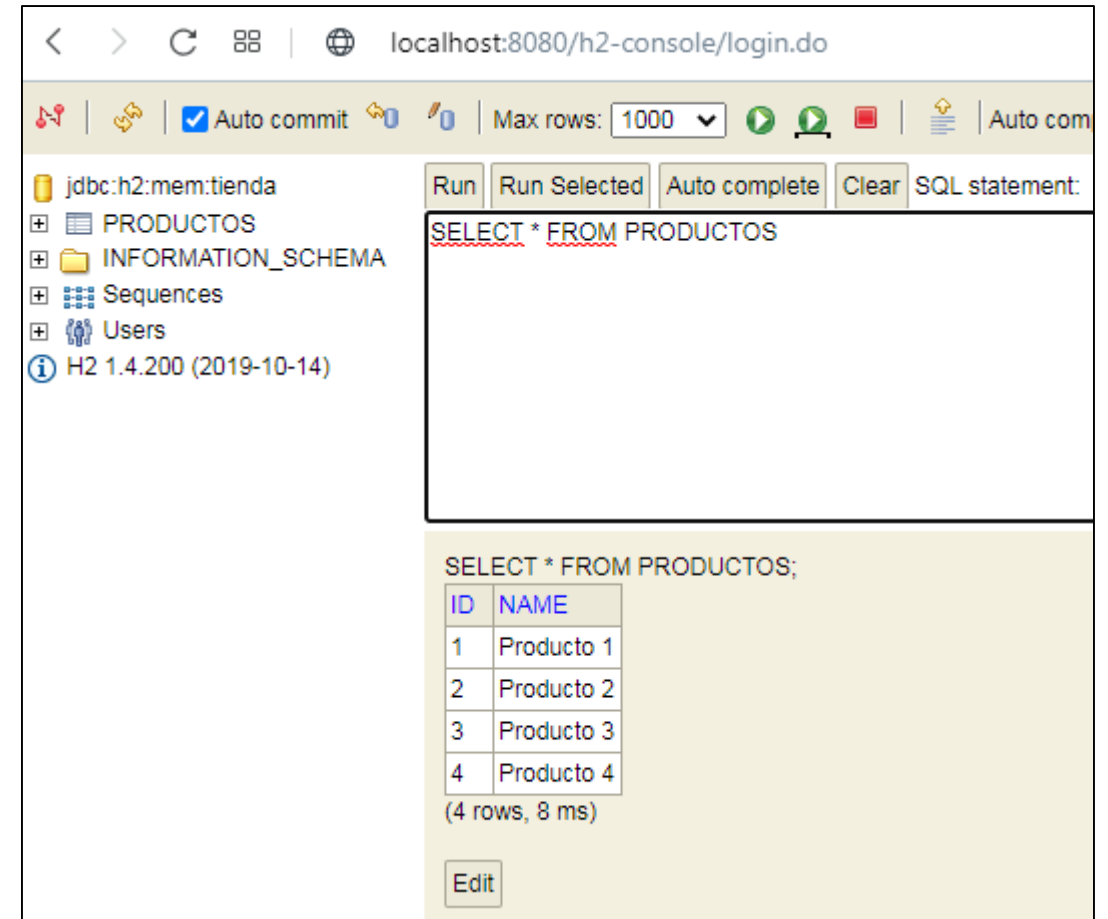
# 4. INICIALIZACION H2

## 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 H2 console login page 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 includes a 'Saved Settings' dropdown set to 'Generic H2 (Embedded)', a 'Setting Name' field with the same value, and 'Save' and 'Remove' buttons. Below this, the 'Driver Class' is 'org.h2.Driver', the 'JDBC URL' is 'jdbc:h2:mem:tienda', the 'User Name' is 'sa', and the 'Password' field is empty. 'Connect' and 'Test Connection' buttons are at the bottom.

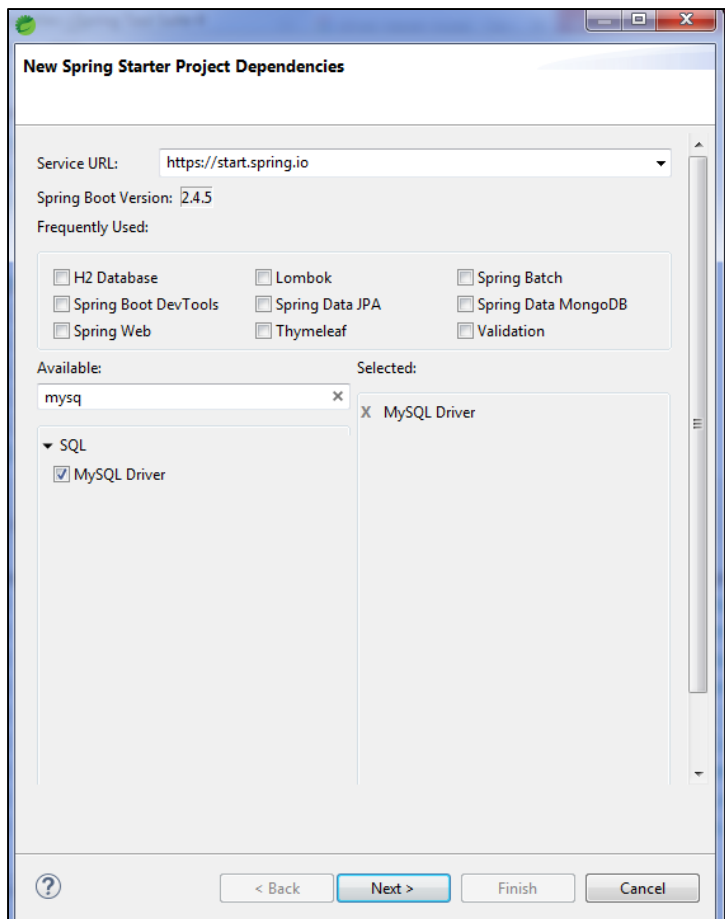


The screenshot shows the H2 console main interface at localhost:8080/h2-console/login.do. The top bar includes navigation icons, a refresh button, and a 'Max rows' dropdown set to '1000'. The left sidebar shows the database structure: 'jdbc:h2:mem:tienda' with folders for 'PRODUCTOS', 'INFORMATION\_SCHEMA', 'Sequences', and 'Users', and a version indicator 'H2 1.4.200 (2019-10-14)'. The main area has buttons for 'Run', 'Run Selected', 'Auto complete', and 'Clear', followed by an 'SQL statement:' input field containing 'SELECT \* FROM PRODUCTOS'. Below the input field, the query result is displayed as a table with 4 rows and 2 columns (ID, NAME). The table shows products 1 through 4. The execution time is '(4 rows, 8 ms)'. An 'Edit' button is at the bottom.

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

# 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 a MySQL database 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**

```
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');
```

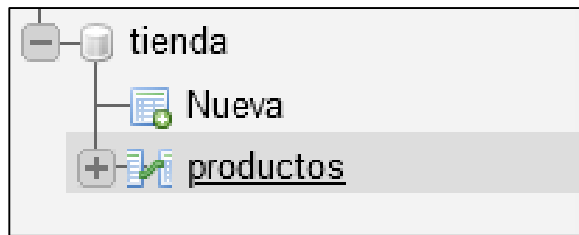
**import: Bloc de notas**

```
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)

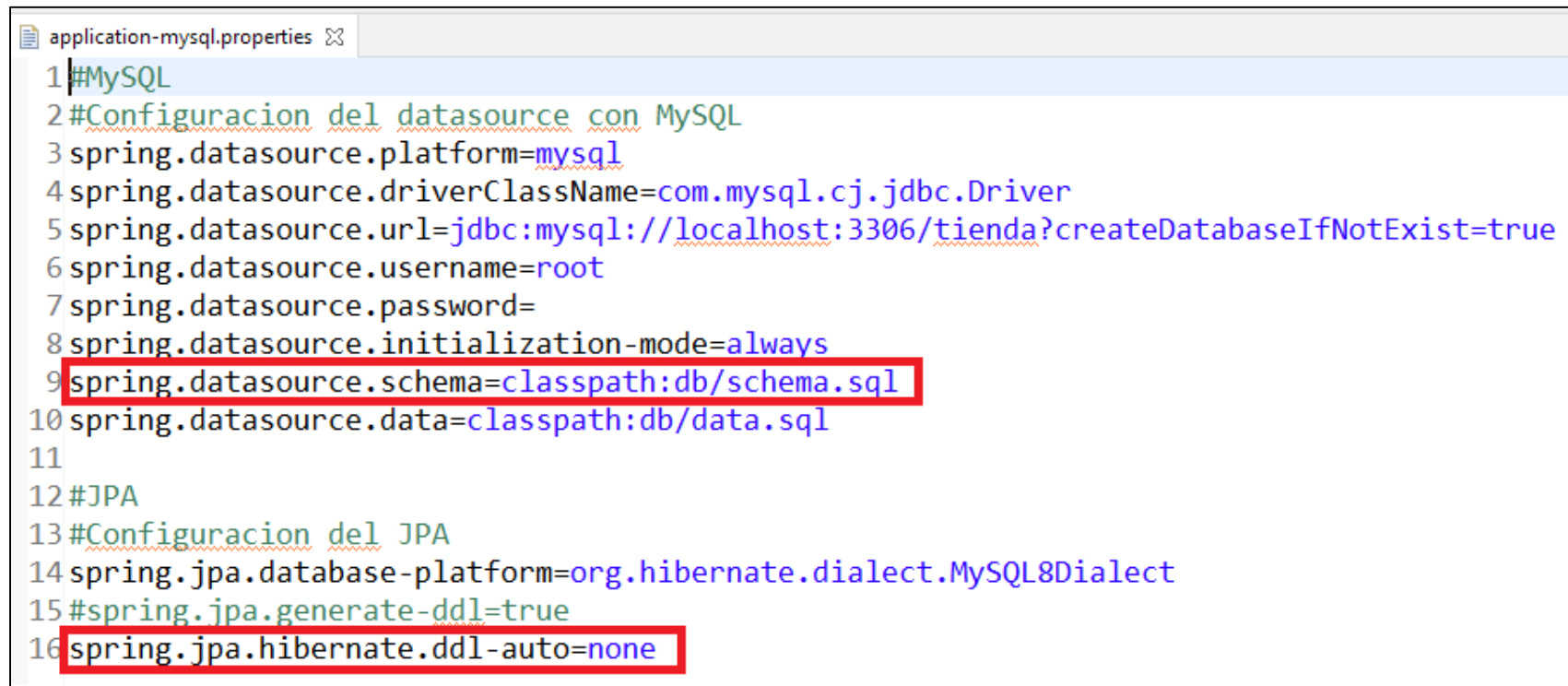
La imagen es una captura de pantalla de la interfaz de usuario de phpMyAdmin. La barra superior indica la conexión al servidor 127.0.0.1, la base de datos 'tienda' y la tabla seleccionada 'productos'. El menú de navegación incluye opciones como Examinar, Estructura, SQL, Buscar, Insertar, Exportar, Importar, Privilegios y Operaciones. Un mensaje de estado verde confirma que se están mostrando 8 filas. El editor de SQL contiene la consulta 'SELECT \* FROM `productos`'. Debajo, hay una barra de herramientas para la consulta, incluyendo opciones de perfilado y edición. La sección de visualización muestra 'Mostrar todo', 'Número de filas' configurado en 25, un campo de filtro y una opción de ordenamiento. La tabla de datos presenta 8 filas, cada una con botones de acción (Editar, Copiar, Borrar) y los campos 'id' y 'name'.

				id	name
<input type="checkbox"/>	Editar	Copiar	Borrar	1	Producto 41
<input type="checkbox"/>	Editar	Copiar	Borrar	2	Producto 42
<input type="checkbox"/>	Editar	Copiar	Borrar	3	Producto 43
<input type="checkbox"/>	Editar	Copiar	Borrar	4	Producto 44
<input type="checkbox"/>	Editar	Copiar	Borrar	5	Producto 1
<input type="checkbox"/>	Editar	Copiar	Borrar	6	Producto 2
<input type="checkbox"/>	Editar	Copiar	Borrar	7	Producto 3
<input type="checkbox"/>	Editar	Copiar	Borrar	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**)

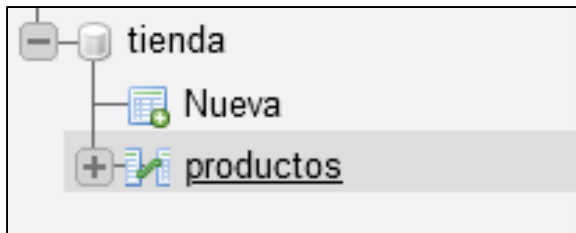


```
application-mysql.properties
1 #MySQL
2 #Configuracion del datasource con MySQL
3 spring.datasource.platform=mysql
4 spring.datasource.driverClassName=com.mysql.cj.jdbc.Driver
5 spring.datasource.url=jdbc:mysql://localhost:3306/tienda?createDatabaseIfNotExist=true
6 spring.datasource.username=root
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=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');
```

Captura de pantalla de la interfaz de MySQL Workbench. La barra superior muestra: Servidor: 127.0.0.1 » Base de datos: tienda » Tabla: productos. El menú principal incluye: Examinar, Estructura, SQL, Buscar, Insertar, Exportar, Importar. El panel central muestra un mensaje de éxito: 'Mostrando filas 0 - 3 (total de 4, La consulta tardó 0,0010 segundos.)' y la consulta SQL: 'SELECT \* FROM `productos`'. Debajo de la consulta hay botones: Perfilando, Editar en línea, Editar, Explicar SQL, Crear código PHP, Actualizar. El panel inferior muestra una tabla con 4 filas y 2 columnas: ID y NAME. Cada fila tiene botones de acción: Editar, Copiar, Borrar.

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.

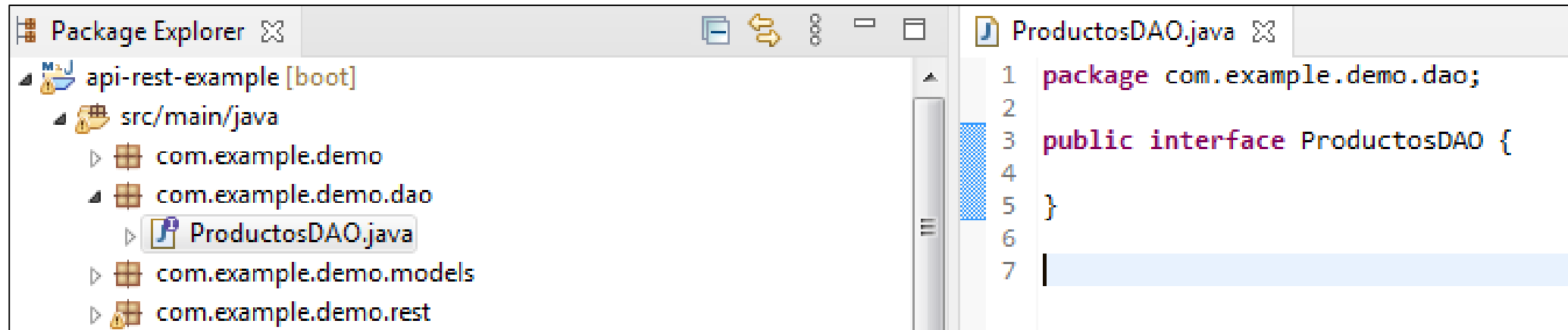
Según el Patron DAO, una vez tenemos las clases que representan nuestros datos (en nuestro caso Producto), se debe de crear una interface con los métodos necesarios para obtener y almacenar Productos. No debe tener nada que la relacione con la base de datos (sin parámetro Connection).

```
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);  
    ...  
}
```

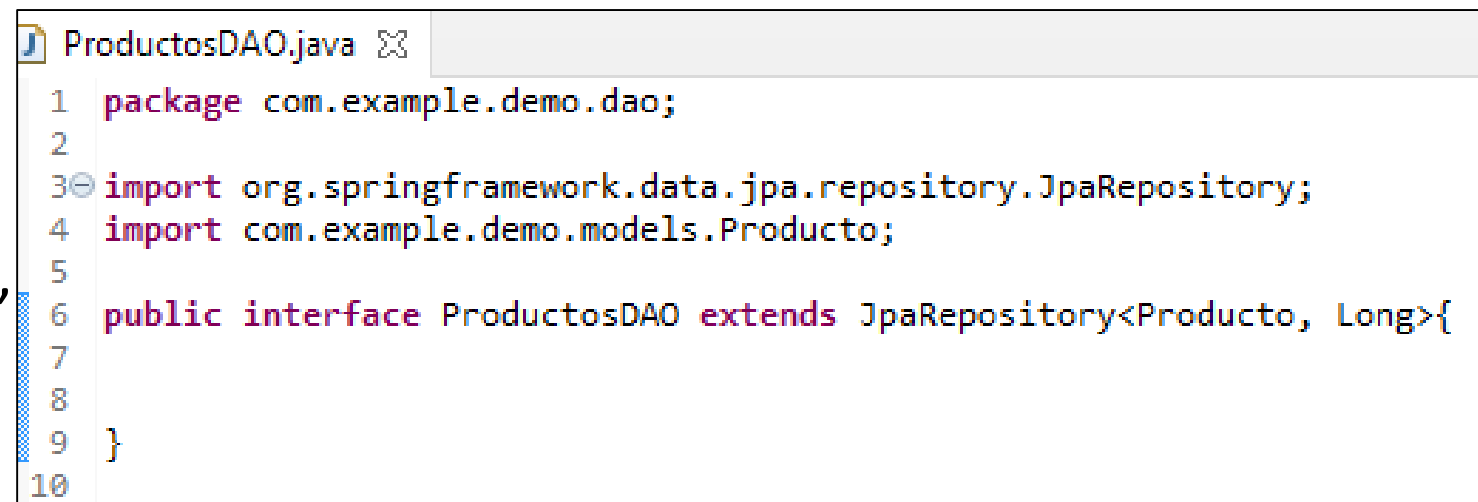


## 6. CLASE DAO

**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

## Interfaces CrudRepository vs JpaRepository en Spring Data jpa

- **CrudRepository** proporciona principalmente funciones CRUD .
- **PagingAndSortingRepository** proporcionan métodos para hacer la paginación y ordenar los registros.
- **JpaRepository** proporciona algunos métodos relacionados con JPA, como el vaciado del contexto de persistencia y la eliminación de registros en un lote. JpaRepository tendrá todas las funciones de CrudRepository y PagingAndSortingRepository (es la mas completa() )

Si no necesita que el repositorio tenga las funciones proporcionadas por JpaRepository y PagingAndSortingRepository utilizar CrudRepository

## 6. CLASE DAO

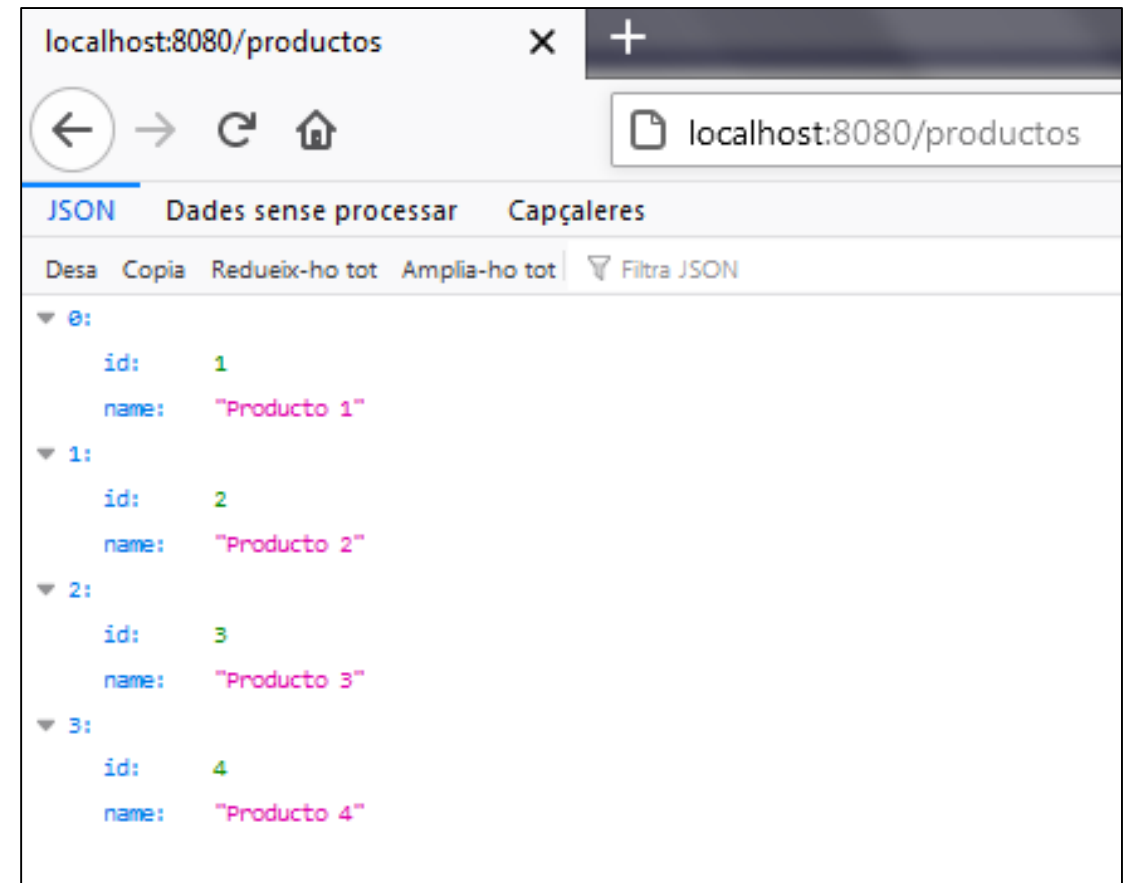
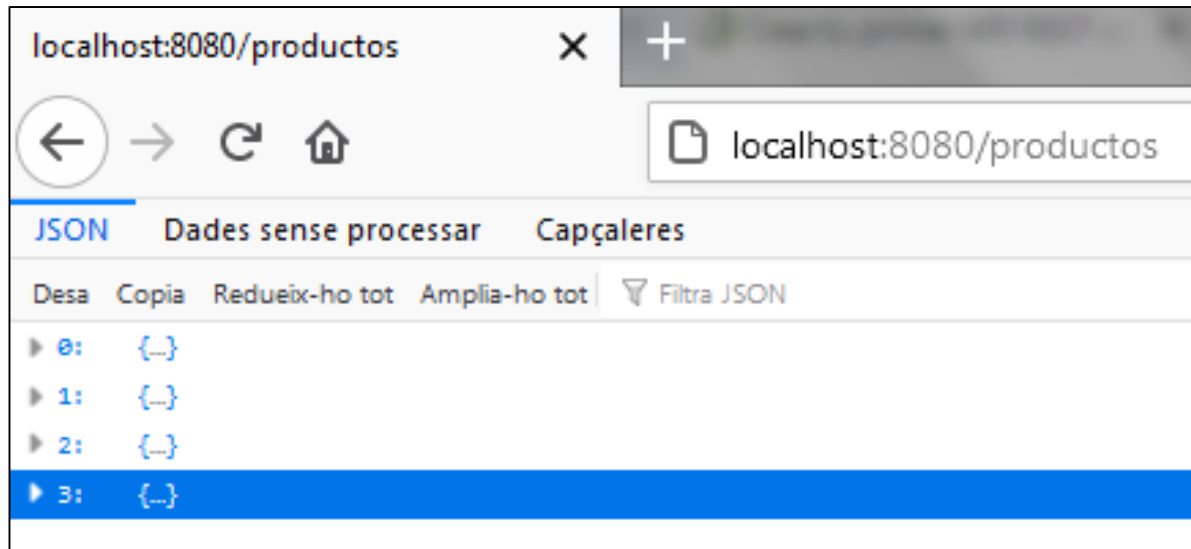
**Paso 3)** Una vez creada la interfaz DAO, creamos el atributo productosDAO en el controlador, sin new ProductosDAO(), sólo con la anotación @Autowired.

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

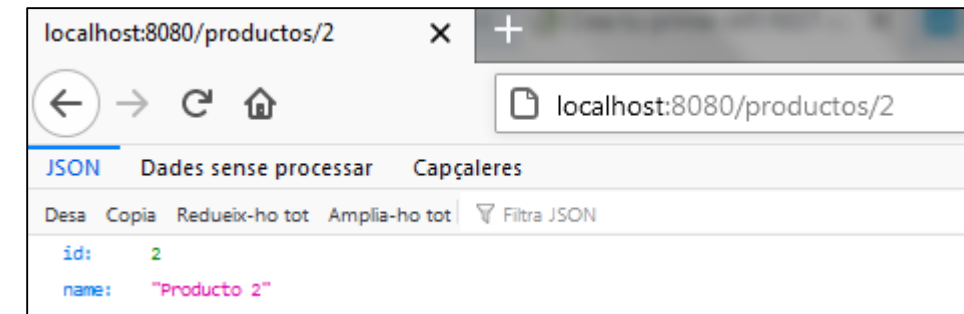
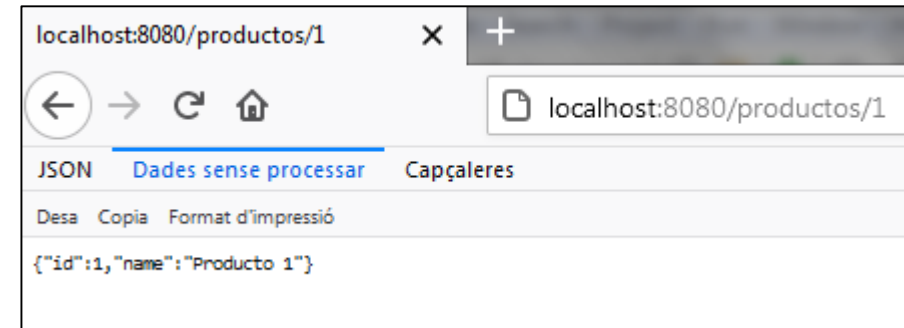
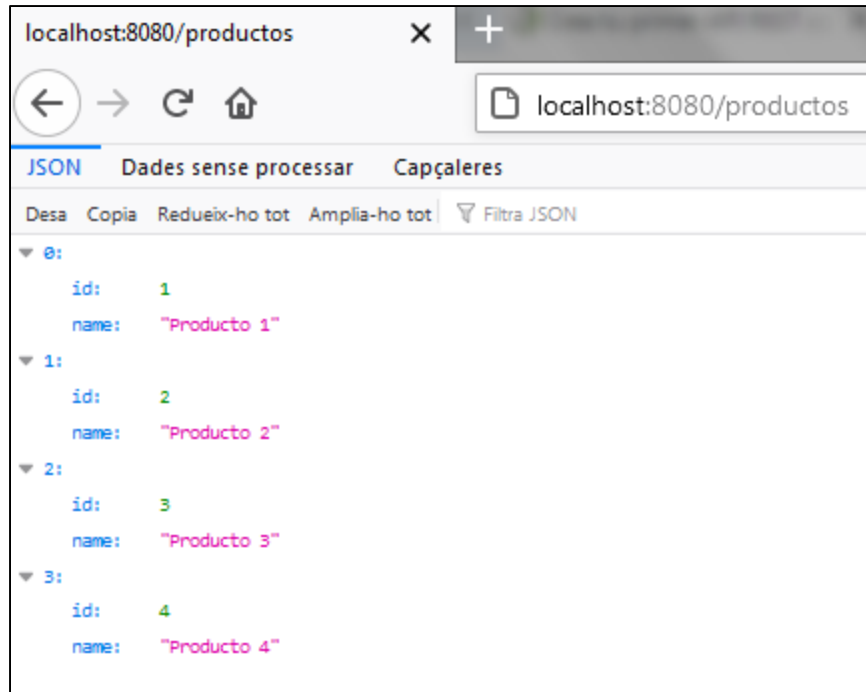
**Paso 2)** Creamos el servicio que nos permita leer un producto en concreto usando la anotación `@PathVariable` (GET):

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
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     @Autowired //Inyeccion de dependencias
10     private ProductosDAO productosDAO;
11
12     @GetMapping
13     public ResponseEntity<List<Producto>> getProducto() {
14         List<Producto> productos = productosDAO.findAll();
15         return ResponseEntity.ok(productos);
16     }
17
18     @RequestMapping(value="{productId}") //productos/{productId} --> productos/1
19     public ResponseEntity<Producto> getProductoById(@PathVariable("productId") Long productId) {
20         Optional<Producto> optionalProducto = productosDAO.findById(productId);
21         if (optionalProducto.isPresent()) {
22             return ResponseEntity.ok(optionalProducto.get());
23         } else {
24             return ResponseEntity.noContent().build();
25         }
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 }
```

# 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
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     @RequestMapping(value="{productId}") //productos/{productId} --> productos/1
40     public ResponseEntity<Producto> getProductoById(@PathVariable("productId") Long productId) {
41         Optional<Producto> optionalProducto = productosDAO.findById(productId);
42         if (optionalProducto.isPresent()) {
43             return ResponseEntity.ok(optionalProducto.get());
44         } else {
45             return ResponseEntity.noContent().build();
46         }
47     }
48 }
```

# 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 blue 'SEND' button is located to the right. 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 shown as a JSON object: `{ "name": "Producto n" }`. To the right of the main interface, a response panel shows a status of '200 OK' and a response time of '119.71 ms'. Below this, there are buttons for 'COPY', 'SAVE', 'SOURCE VIEW', and 'DATA TABLE'. The response body is displayed as 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"
}
```



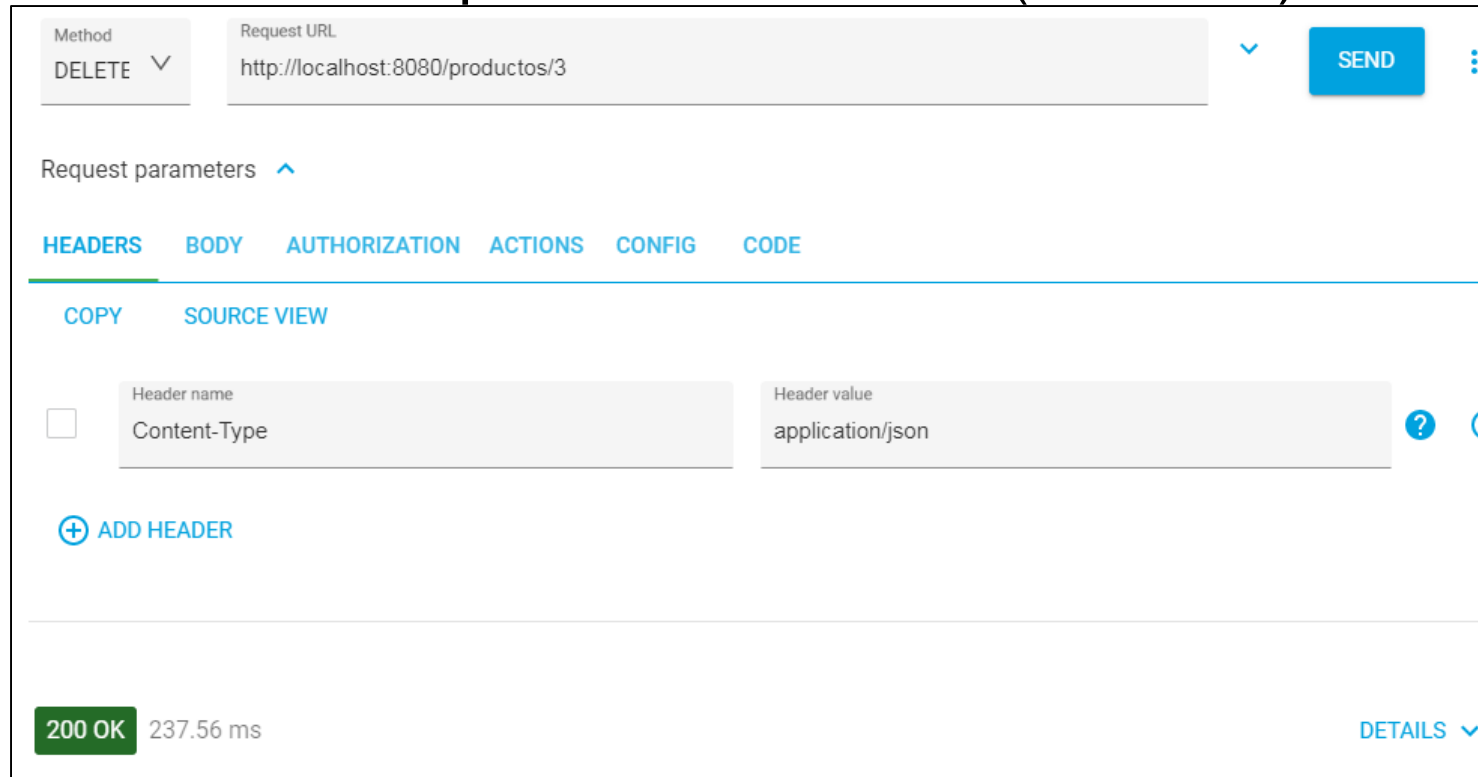
# 7. SERVICIOS API REST

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

```
ControllerRest.java
1 package com.example.demo.rest;
2
3 import java.util.List;
4
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 newProducto = productosDAO.save(producto);
36         return ResponseEntity.ok(newProducto);
37     }
38
39     @DeleteMapping(value="{productoId}") //productos/{productId} (DELETE)
40     public ResponseEntity<Void> deleteProducto(@PathVariable("productoId") Long productId) {
41         productosDAO.deleteById(productId);
42         return ResponseEntity.ok(null);
43     }
44 }
```

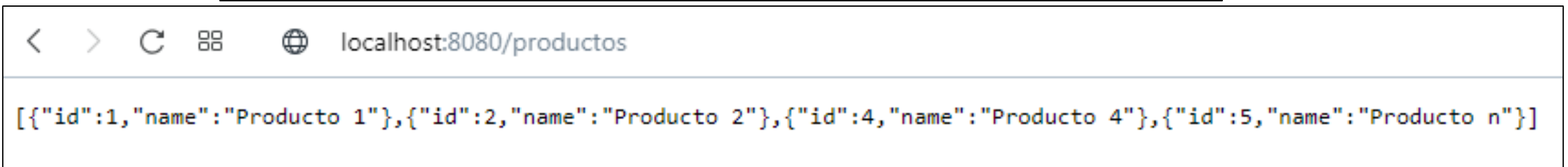
# 7. SERVICIOS API REST

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



The screenshot shows the REST Client interface with the following details:

- Method:** DELETE
- Request URL:** http://localhost:8080/productos/3
- Buttons:** SEND, and a menu icon (three dots).
- Request parameters:** Expandable section.
- Tabs:** HEADERS (selected), BODY, AUTHORIZATION, ACTIONS, CONFIG, CODE.
- Header List:**
  - ☐ Header name: Content-Type, Header value: application/json. Includes a help icon (?) and a minus icon (-).
- Actions:** COPY, SOURCE VIEW, and a blue button with a plus icon and the text "ADD HEADER".
- Response:** 200 OK, 237.56 ms. Includes a "DETAILS" link with a downward arrow.



The screenshot shows a web browser with the address bar displaying "localhost:8080/productos". The main content area displays a JSON array of product objects:

```
[{"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 productoId) {
42     productosDAO.deleteById(productoId);
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"}]
```

## 8. ONETOMANY

---

<https://www.adictosaltrabajo.com/2020/04/02/hibernate-onetoone-onetomany-manytoone-y-manytomany/>

<https://www.oscarblancarteblog.com/2018/12/20/relaciones-onetomany/>

# 9. METODOS DERIVADOS JPA

## Búsquedas por nombre de método

- **findBy**LastName**And**Firstname, **findBy**LastName**Or**Firstname
- **findBy**Firstname, **findBy**Firstname**Is**, **findBy**Firstname**Equals**
- **findBy**StartDate**Between**
- **findBy**Age**LessThan**, **findBy**Age**LessThanEqual**, **findBy**Age**GreaterThan**, **findBy**Age**GreaterThanEqual**
- **findBy**StartDate**After**, **findBy**StartDate**Before**
- **findBy**Age**IsNull**, **findBy**Age**NotNull**, **findBy**Age**IsNotNull**
- **findBy**Firstname**Like**, **findBy**Firstname**NotLike**
- **findBy**Firstname**StartingWith**, **findBy**Firstname**EndingWith**
- **findBy**Firstname**Containing**
- **findBy**Age**OrderBy**LastName**Desc**
- **findBy**LastName**Not**
- **findBy**Age**In**(Collection<Age> ages), **findBy**Age**NotIn**(Collection<Age> ages)
- **findBy**Active**True**(), **findBy**Active**False**()
- **findBy**Firstname**IgnoreCase**
- List<Entity> **findBy**Atr1(String atr1);
- int **deleteBy**Atr1**GreaterThan**(int value);

```
public interface PicturesDAO extends JpaRepository<Picture, Long> {  
    List<Picture> findPicturesByShopId(Long shopId);  
    void deletePicturesByShopId(Long shopId);  
    List<Picture> findPicturesByShop(Shop shop);  
}
```

find  
delete

- First
- First\*
- Distinct

By

- Is, Equals, Between, LessThan...
- IgnoreCase

?And,Or

?OrderBy

- <attribute>Desc
- <attribute>Asc

<https://www.baeldung.com/spring-data-derived-queries>