Institut Joan d'Àustria

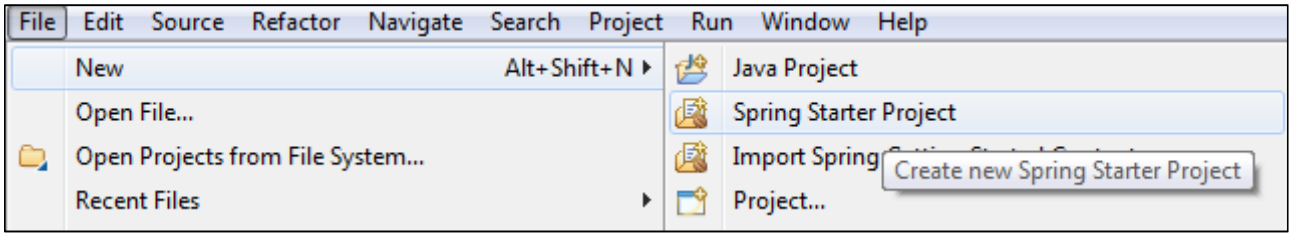
CREACION DE UN SERVICIO API REST SENCILLO

Eduard Lara

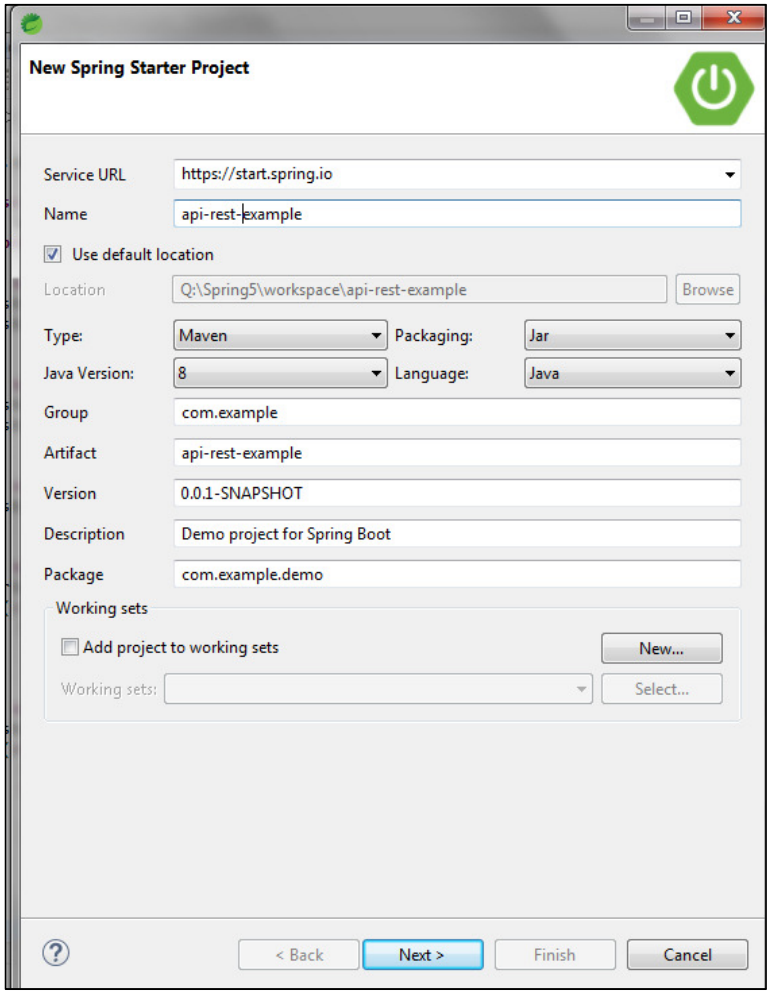
INDICE

1. CREACION PROYECTO

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



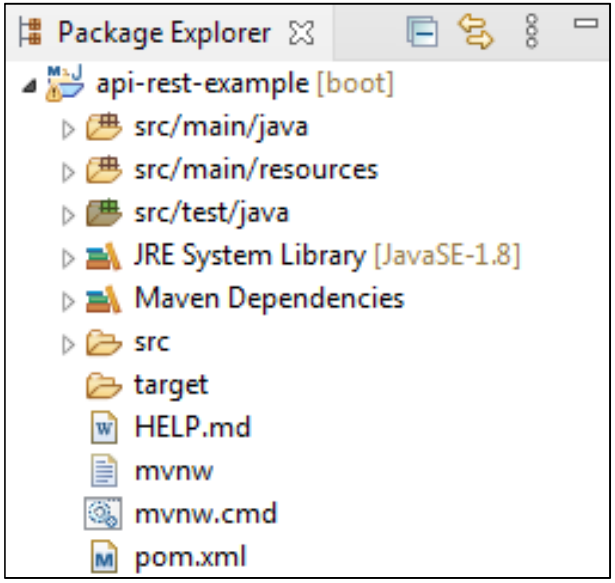
Podemos dejar por defecto los valores que nos presenta el wizard. Si se desea se puede cambiar el nombre de proyecto, el package raíz, el tipo de proyecto (Maven o Gradle) y/o la versión de Java.

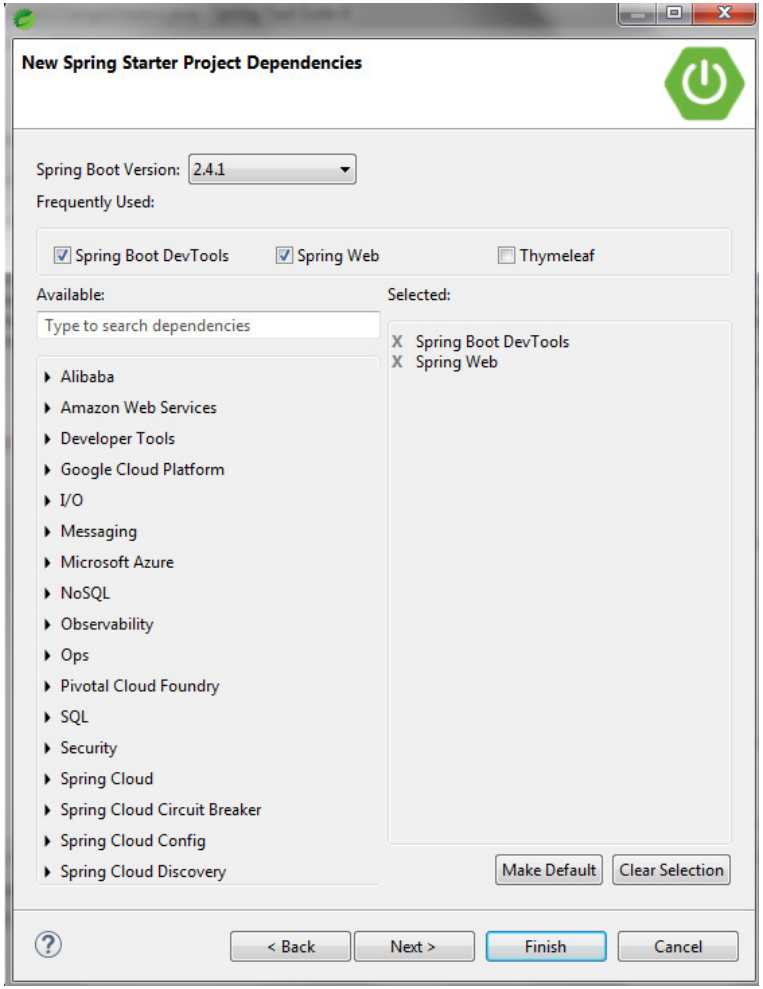


1. CREACION PROYECTO

Paso 2) Agregamos las librerías:

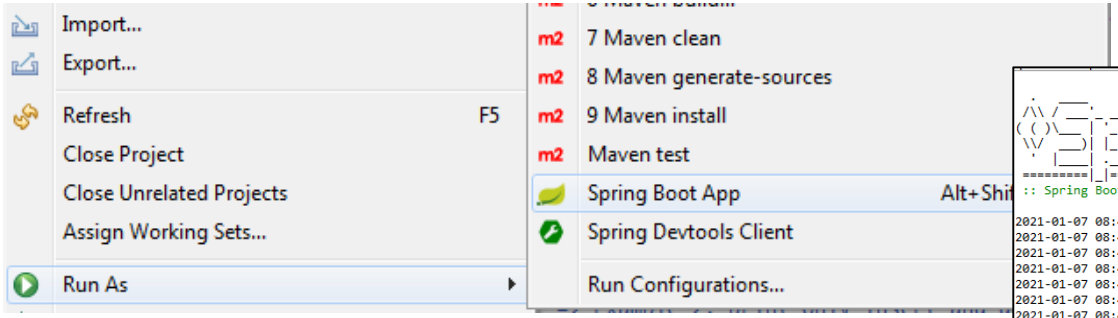
- Spring Web (necesaria)
- Spring Boot Dev Tools (muy importante ya que cualquier cambio que hagamos en nuestro código java, de forma automática se va a actualizar en el despliegue sin tener que reiniciar el servidor)

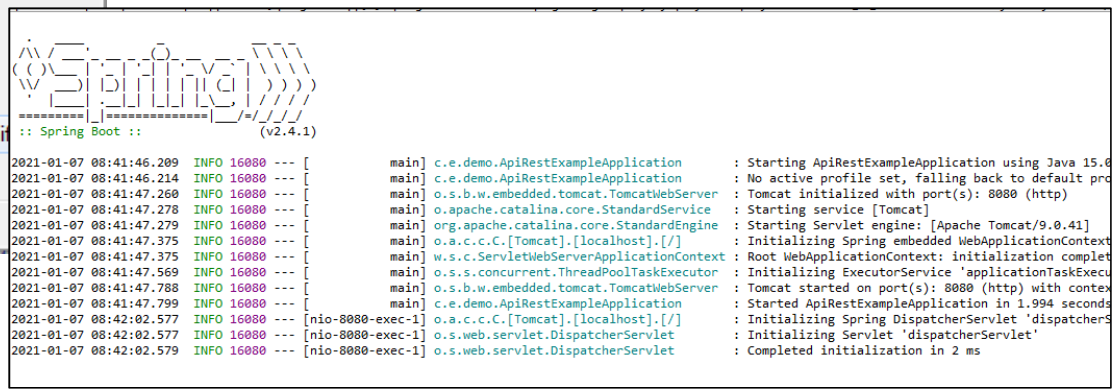


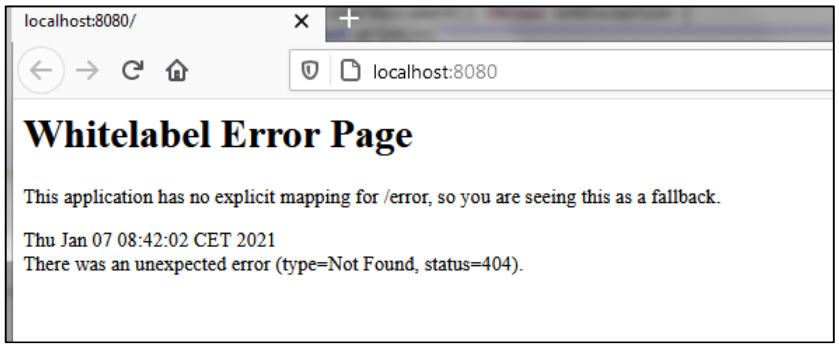


1. CREACION PROYECTO

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.

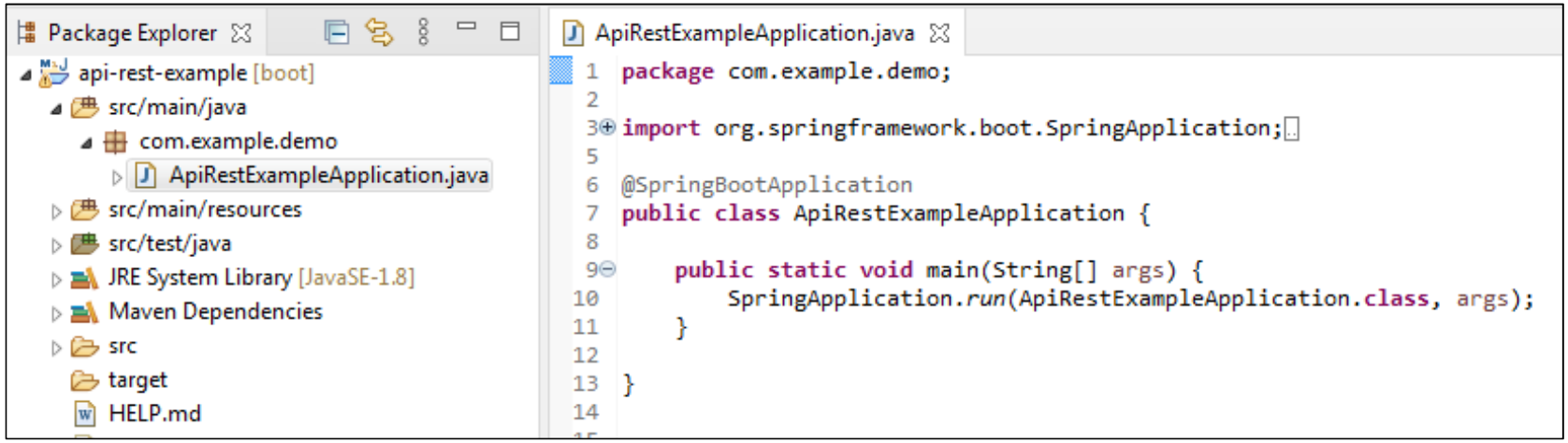






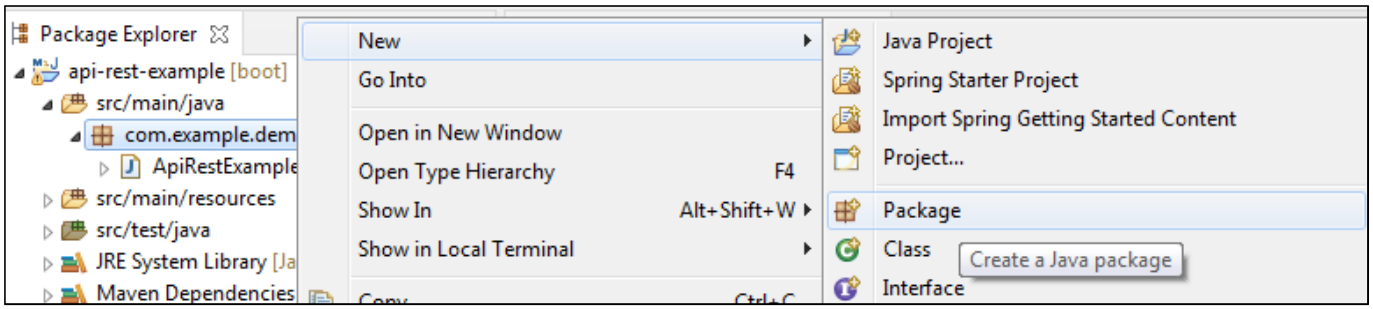
1. CREACION PROYECTO

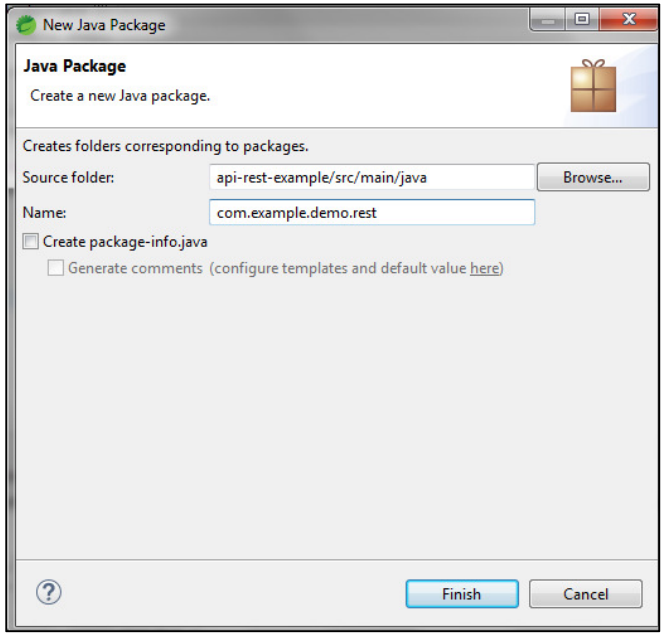
Paso 4) Podemos observar en el package raíz indicado al principio en la creación del proyecto, la clase generada automáticamente que inicia nuestro servidor y la aplicación:

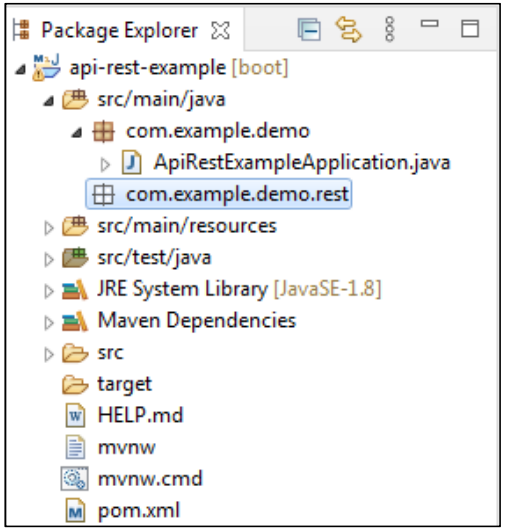


2. CONTROLADOR REST

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

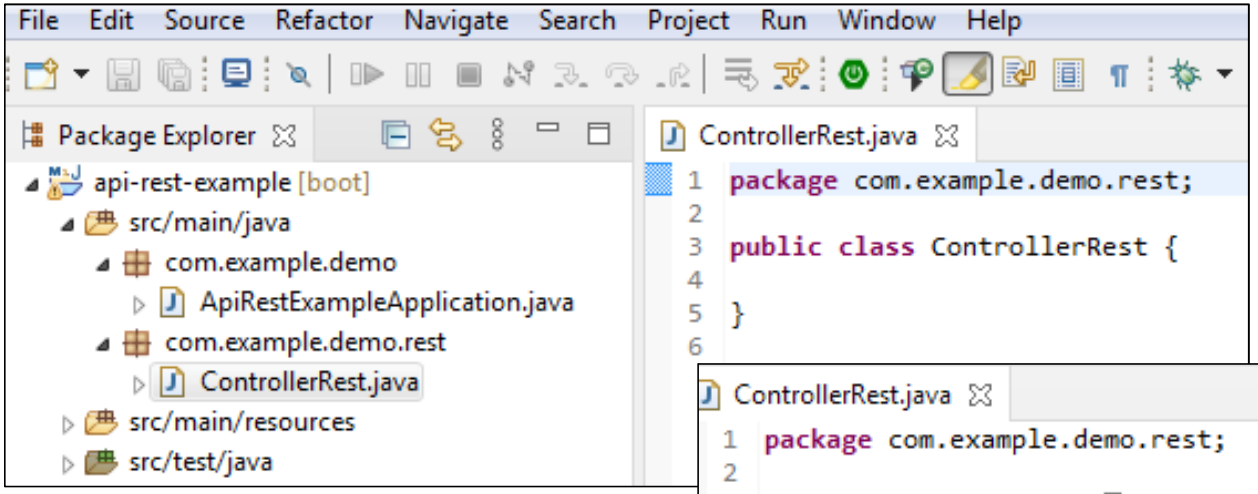






2. CONTROLADOR REST

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:



ControllerRest.java

```
1  package com.example.demo.rest;
2
3  import java.util.List;
20
21 @RestController                  //Indica que esta clase va a ser un servicio REST
22 @RequestMapping("/productos")    //En que URL se va a exponer los servicios de esta clase
23 public class ControllerRest {
24
25
26 }
27
```

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

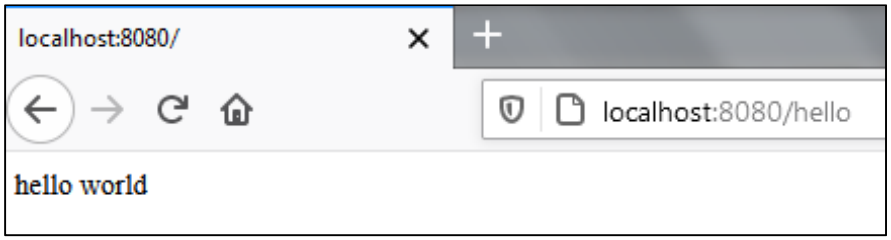
```java
package com.example.demo.rest;

import org.springframework.web.bind.annotation.GetMapping;

@RestController             //Indica que esta clase va a ser un servicio REST
@RequestMapping("/")        //En que URL se va a exponer los servicios de esta clase
public class ControllerRest {

    @GetMapping("")         //Servicio disponible mediante GET (localhost:8080/)
    //@RequestMapping(value="hello", method=RequestMethod.GET)  //En que url esta el servicio
    public String hello() {
        return "hello world";
    }
}

```

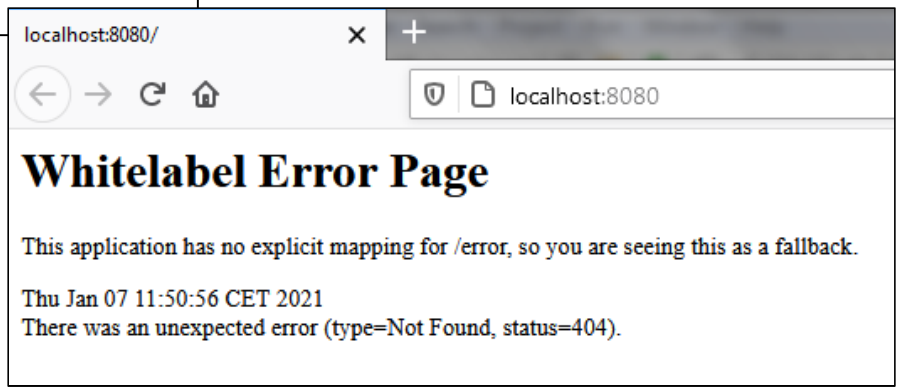
2. CONTROLADOR REST

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

ControllerRest.java

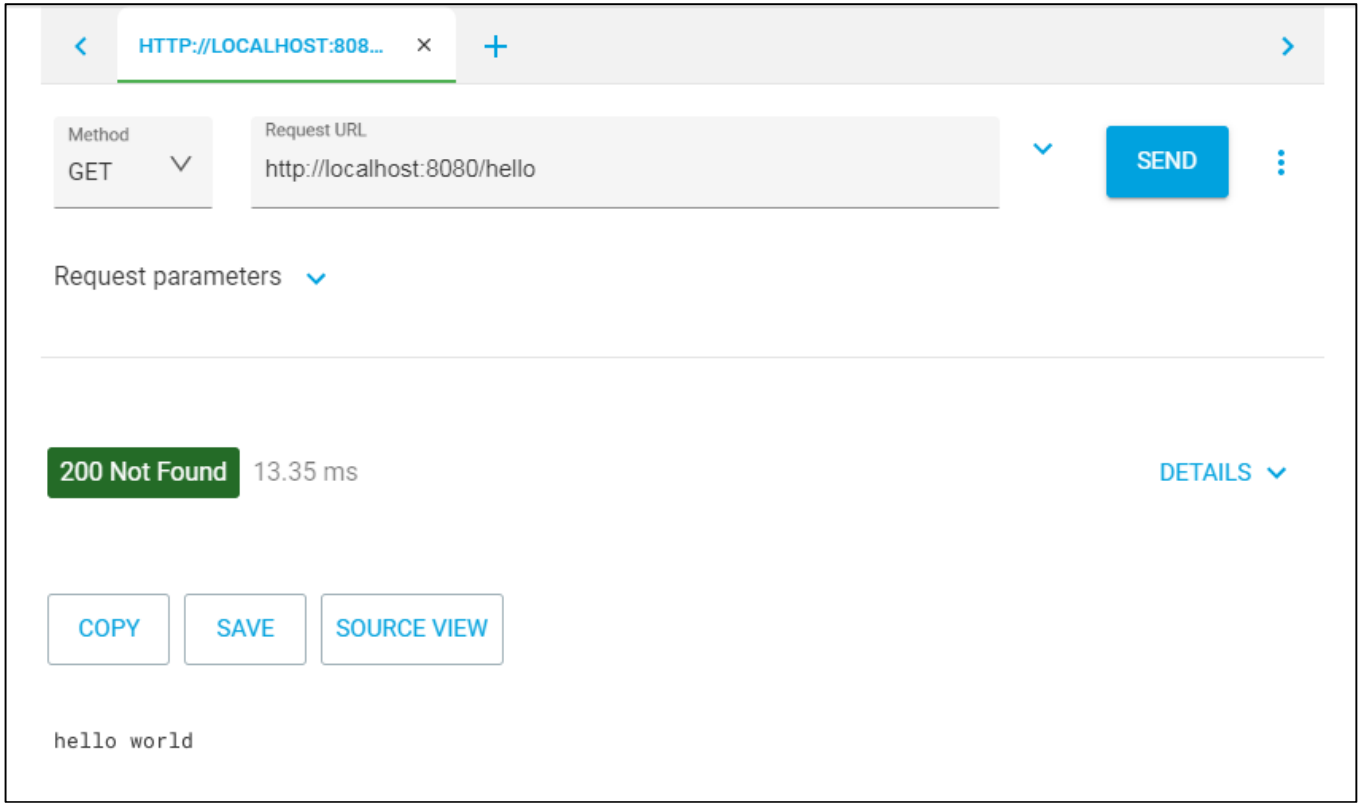
```java
package com.example.demo.rest;

import org.springframework.web.bind.annotation.GetMapping;

@RestController            //Indica que esta clase va a ser un servicio REST
@RequestMapping("/")       //En que URL se va a exponer los servicios de esta clase
public class ControllerRest {

    @GetMapping("hello")        //Servicio disponible mediante GET (localhost:8080/hello)
    //@RequestMapping(value="hello", method=RequestMethod.GET)  //En que url esta el servicio
    public String hello() {
        return "hello world";
    }
}
```





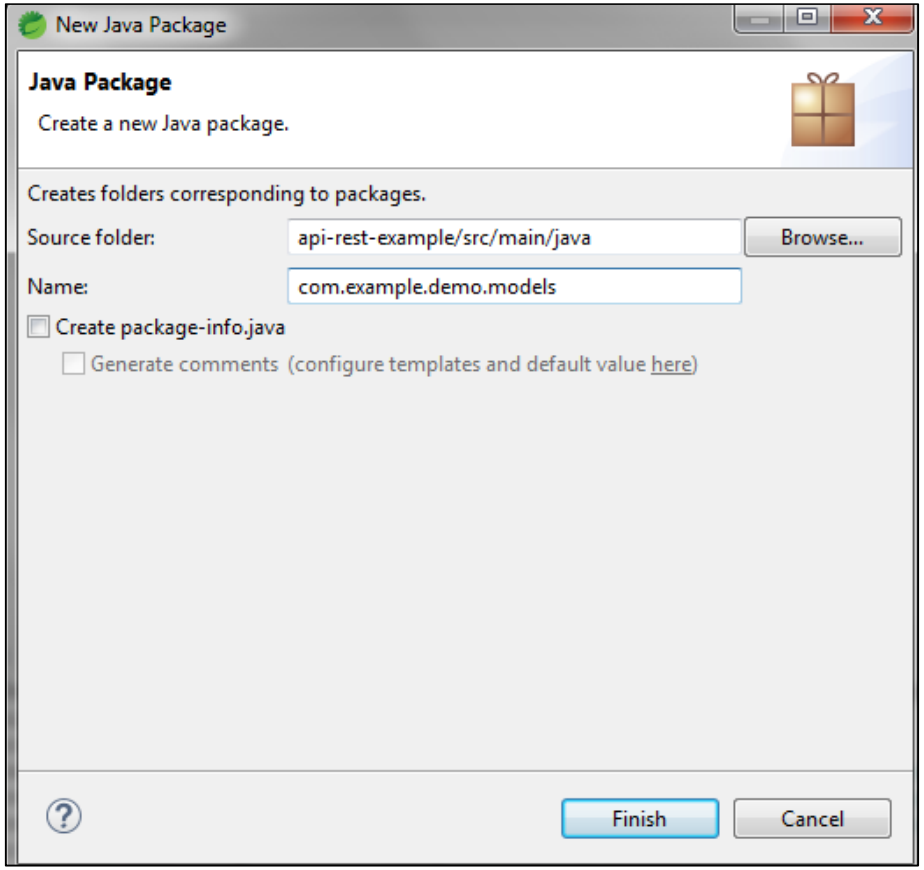
2. CONTROLADOR REST

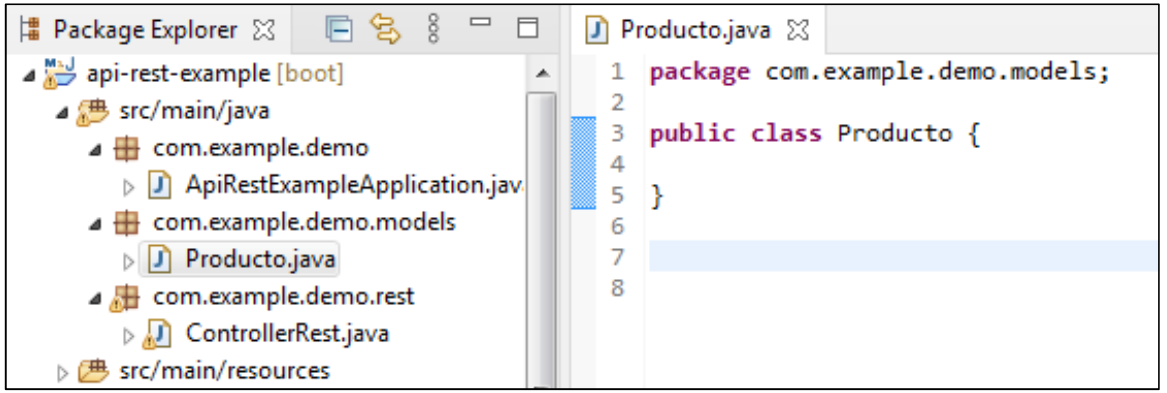
Paso 5) Podemos probar el servicio con la aplicación Advanced Rest Client:



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

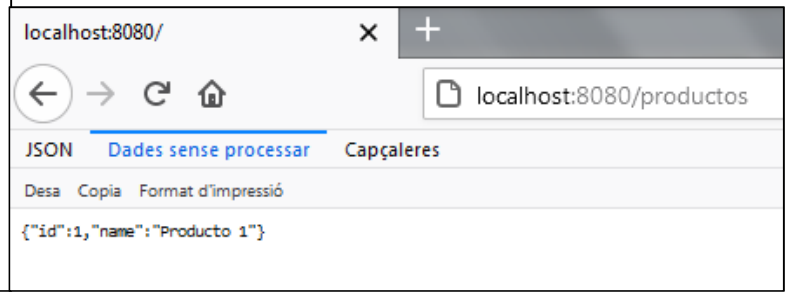
```java
package com.example.demo.models;

public class Producto {
    private long id;
    private String name;

    public long getId() {
        return id;
    }
    public void setId(long id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

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

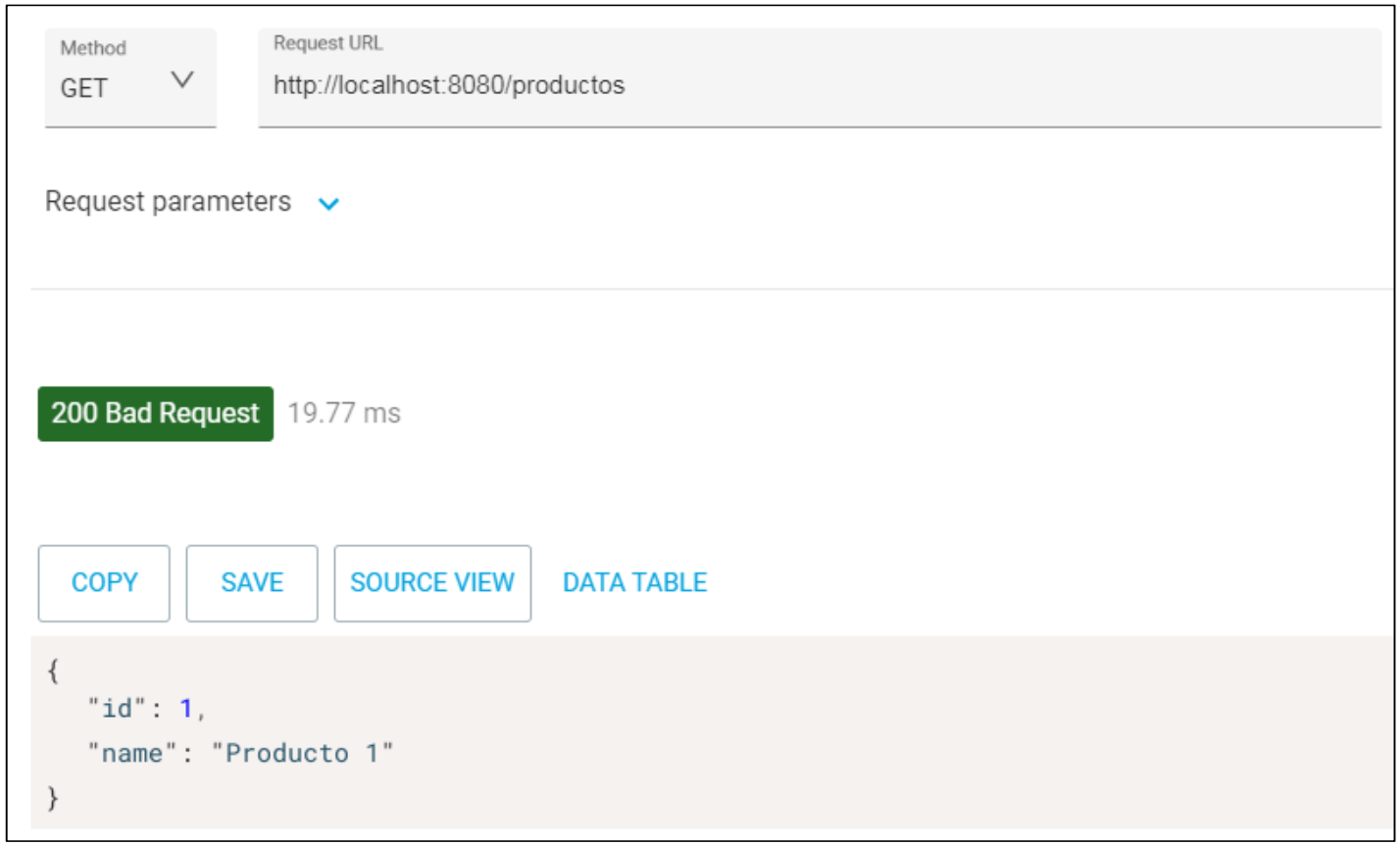
ControllerRest.java

```java
package com.example.demo.rest;

import org.springframework.http.ResponseEntity;

@RestController                 //Indica que esta clase va a ser un servicio REST
@RequestMapping("/productos")   //En que URL se va a exponer los servicios de esta clase
public class ControllerRest {

    @GetMapping
    public ResponseEntity<Producto> getProducto() {
        Producto producto = new Producto();
        producto.setId(1);
        producto.setName("Producto 1");
        return ResponseEntity.ok(producto);
    }

    //@GetMapping("hello")          //Servicio disponible mediante GET (localhost:8080/hello)
    //@RequestMapping(value="hello", method=RequestMethod.GET)  //En que url esta el servicio
    public String hello() {
        return "hello world";
    }
}
```



3. MODELO DE DATOS

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



4. JDBC Y JPA

Paso 1) Para poder conectarnos a la base de datos mysql, necesitamos el driver mysql-connector de java. Y para implementar persistencia con la base de datos necesitamos el driver de JPA. Mediante estas anotaciones en el fichero pom.xml de Maven, conseguimos ambos drivers:

api-rest-example/pom.xml

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

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

    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.6</version>
    </dependency>

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

</dependencies>
```

JPA o Java Persistence API es la API de persistencia desarrollada para la plataforma Java EE.

Decimos que tenemos persistencia con una tabla de una base de datos cuando existe una clase entity de java que equivale a un elemento - fila de esa tabla de la base de datos

4. JDBC Y JPA

Paso 2) Transformamos la clase Producto en una clase Entity, mediante las anotaciones @Entity y @Table.

Con @Id indicamos el atributo que en la base de datos es primary key.
@Column mapeamos el atributo con la columa de la tabla indicada en @Table.

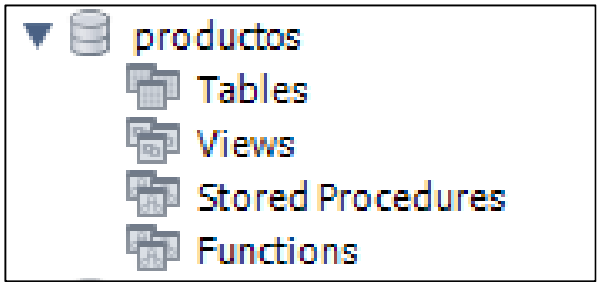
De esta forma un objeto de la clase Producto representará un registro de la tabla productos, donde una de las cosas mas importantes, es que no utilizaremos el típico lenguaje DML de SQL para acceder a la base de datos, sino una API mas sencilla y orientada a objetos

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. JDBC Y JPA

Paso 3) Creamos una base de datos nueva en Mysql, de nombre productos:




```
CREATE DATABASE `productos`
DEFAULT CHARACTER SET utf8 COLLATE utf8_spanish_ci;
```

4. JDBC Y JPA

Paso 4) Comentamos parte de la función getProducto (ahora devolverá una lista de productos) e iniciamos el servidor Tomcat. Nos indica que no hemos configurado la url del datasource:

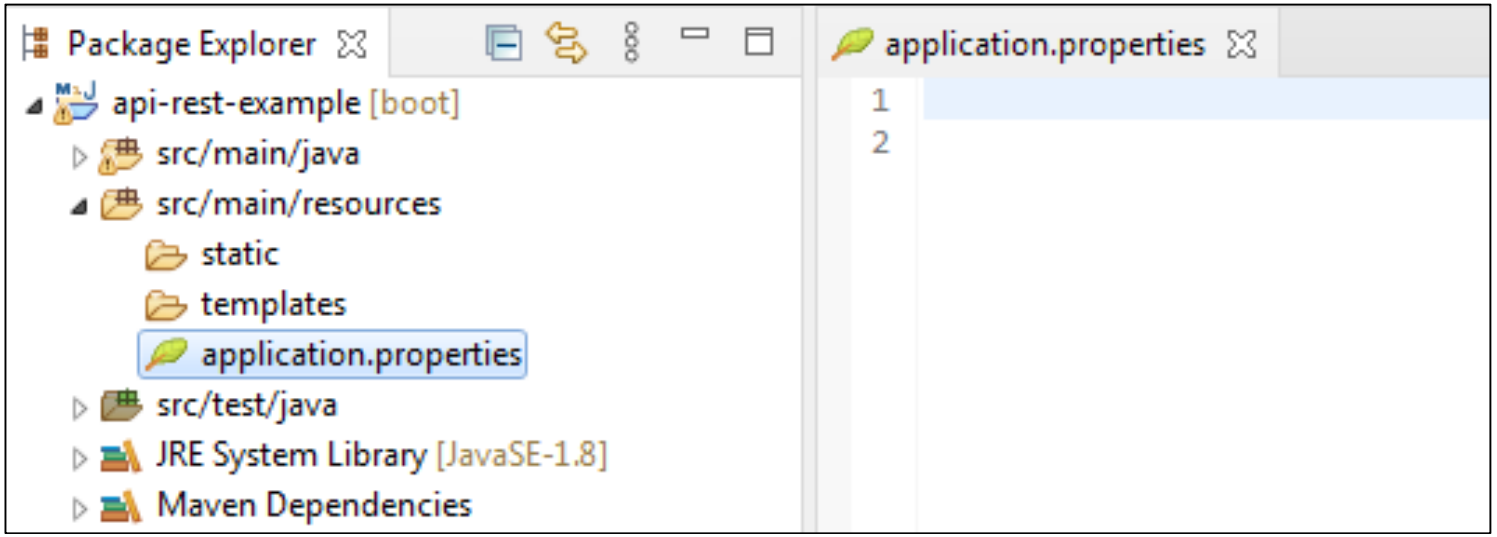
ControllerRest.java

```
1  package com.example.demo.rest;
2
3  import java.util.List;
12
13 @RestController                //Indica que esta clase va a ser un servicio REST
14 @RequestMapping("/productos")  //En que URL se va a exponer los servicios de esta clase
15 public class ControllerRest {
16
17     @GetMapping
18     public ResponseEntity<List<Producto>> getProducto() {
19         //Producto producto = new Producto();
20         //producto.setId(1);
21         //producto.setName("Produ
22         return null;
23         //return ResponseEntity.o
24     }
```

```
***************************
APPLICATION FAILED TO START
***************************

Description:

Failed to configure a DataSource: 'url' attribute is not specified and no embedded datasource could be configured.

Reason: Failed to determine a suitable driver class


Action:

Consider the following:
        If you want an embedded database (H2, HSQL or Derby), please put it on the classpath.
        If you have database settings to be loaded from a particular profile you may need to activate it (no profiles are currently active).
```

4. JDBC Y JPA

Paso 5) Debemos poner toda la configuración necesaria (el clásico connectString de java con mysql) en el fichero application.properties:

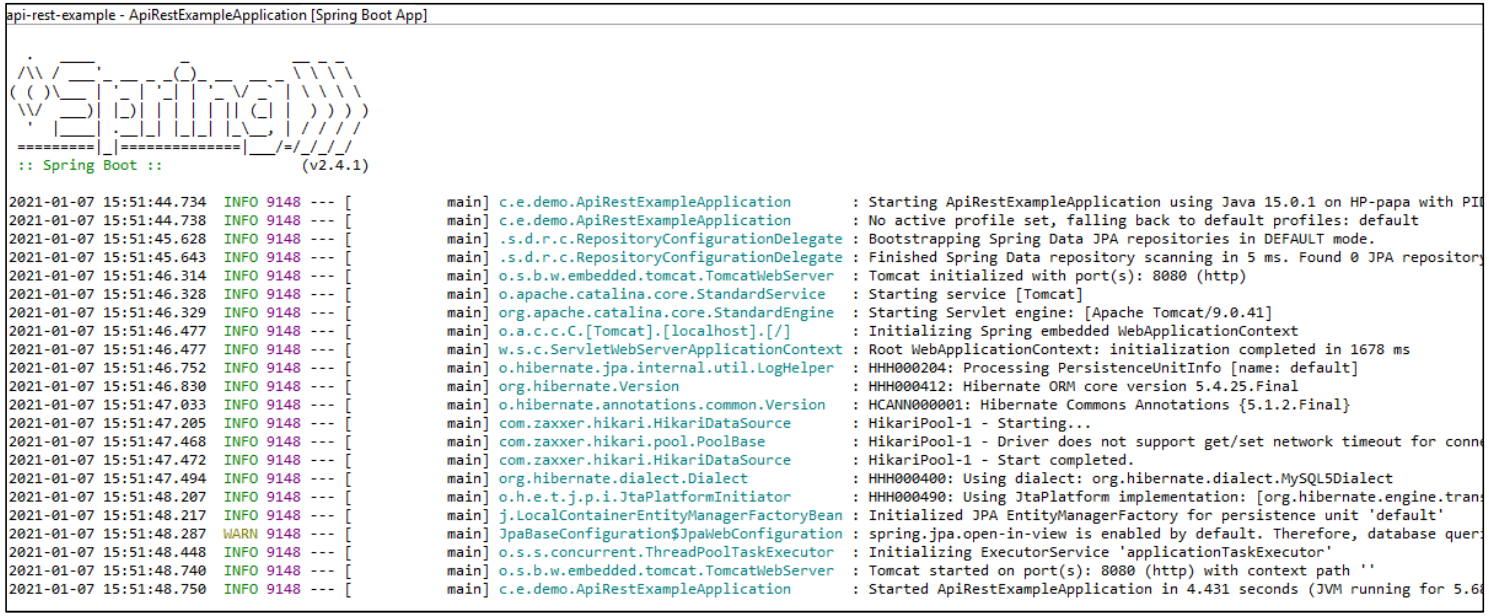


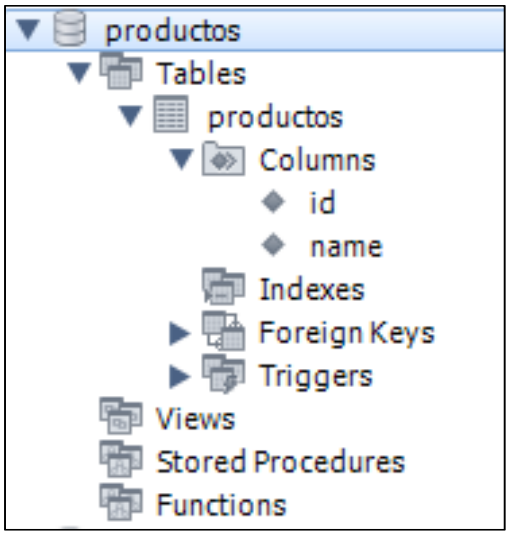
application.properties

```
1 spring.jpa.hibernate.ddl-auto=update
2 spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect
3 spring.datasource.driverClassName=com.mysql.jdbc.Driver
4 spring.datasource.url=jdbc:mysql://localhost:3306/productos
5 spring.datasource.username=root
6 spring.datasource.password=
```

4. JDBC Y JPA

Paso 6) Volvemos a arrancar el servidor y vemos que esto de la persistencia va en serio. El servidor al inicio ha conectado con la base de datos y ha creado la tabla productos que necesitaba para poder establecer la relación de persistencia. La tabla podría haber estado creada de antemano, no hubiera pasado nada. En situaciones de mas datos es preferible que el mapeo de datos mysql-java lo haga JPA





5. CLASE DAO

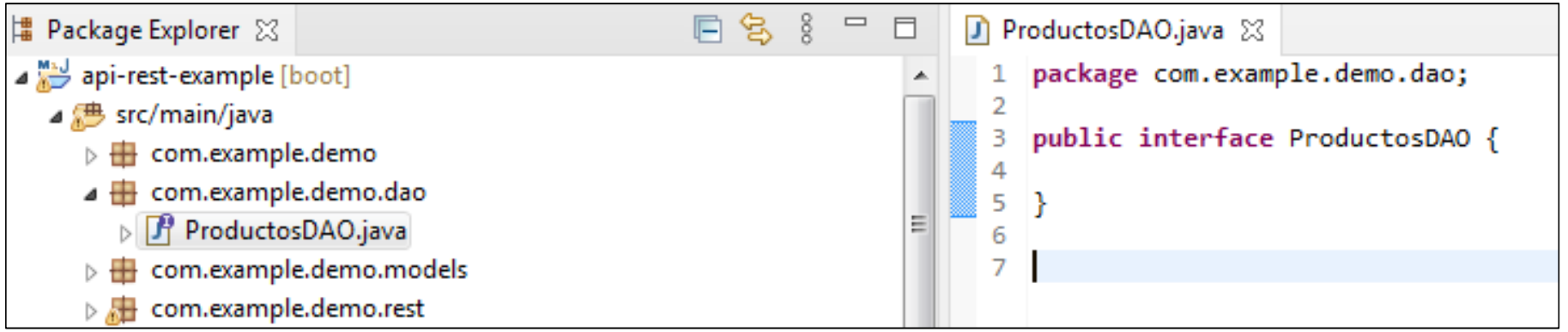
Paso 1) La clase DAO (Data Access Object) es un componente de software que suministra una interfaz común entre la aplicación y uno o más dispositivos de almacenamiento de datos, tales como una Base de datos o un archivo.

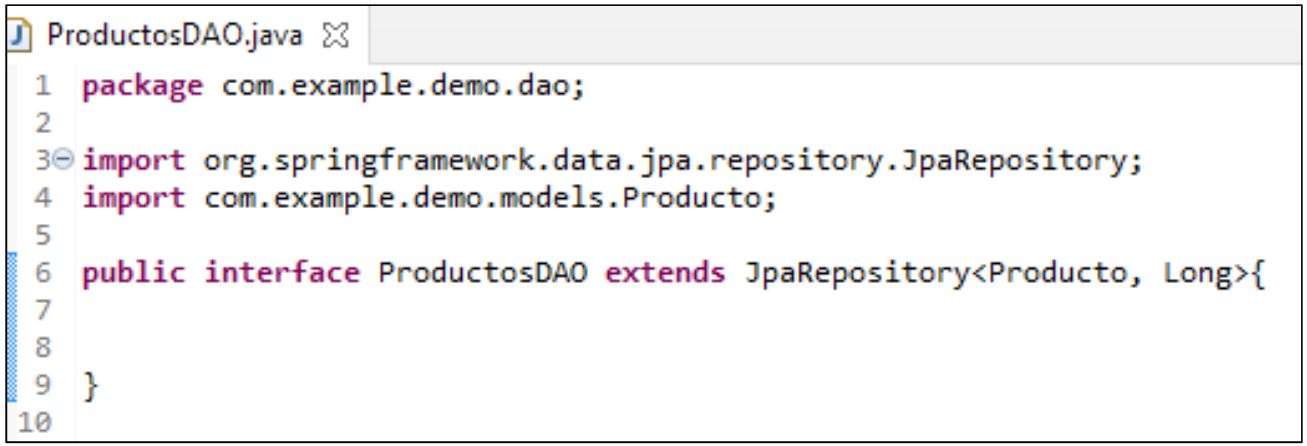
Según el Patron DAO, una vez tenemos las clases que representan nuestros datos (en nuestro caso Producto), se debe de crear una interface con los métodos necesarios para obtener y almacenar Productos. No debe tener nada que la relacione con la base de datos (sin parámetro Connection).

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

5. CLASE DAO

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





5. 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

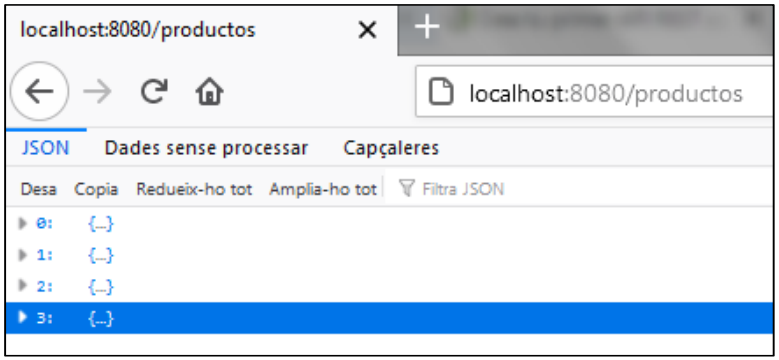
ControllerRest.java

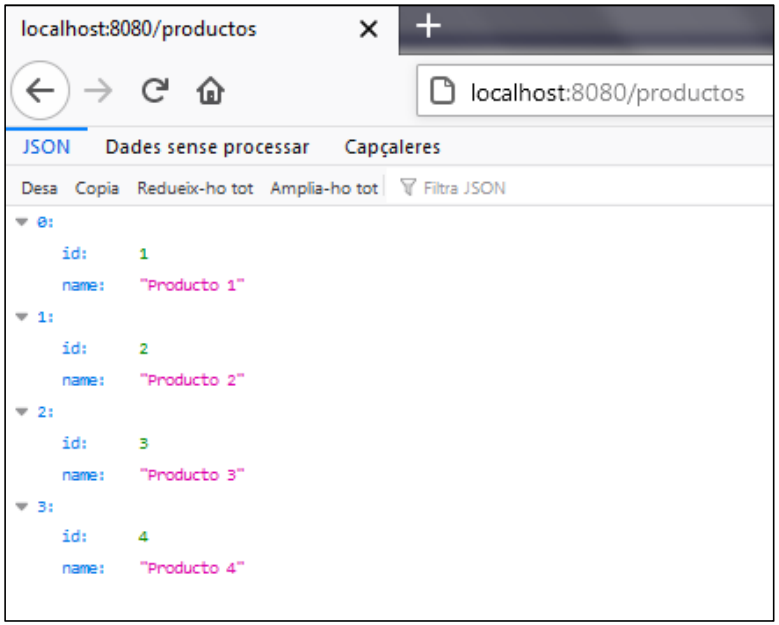
```
1  package com.example.demo.rest;
2
3  import java.util.List;
14
15 @RestController                    //Indica que esta clase va a ser un servicio REST
16 @RequestMapping("/productos")      //En que URL se va a exponer los servicios de esta clase
17 public class ControllerRest {
18
19     @Autowired                    //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
```

6. API REST GET

Paso 1) Insertamos 4 productos desde el Workbench y comprobamos el servicio desde un navegador mediante la url localhost:8080/productos:

```
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');
```





6. API REST GET

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

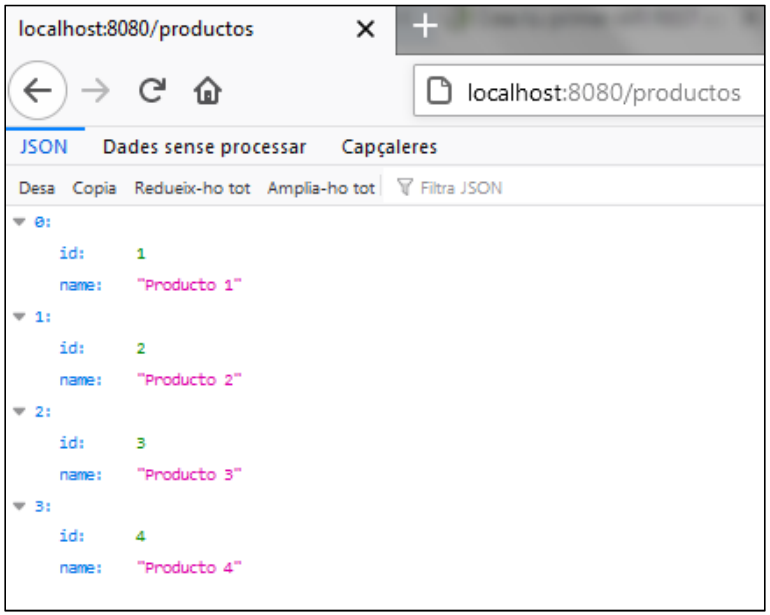
Se deberá usar la url localhost:8080/productos/{productId} de manera que el se información sobre el producto que se desea lo extaera de rvicio extraerá eñ valor por el que buscara de la propia url

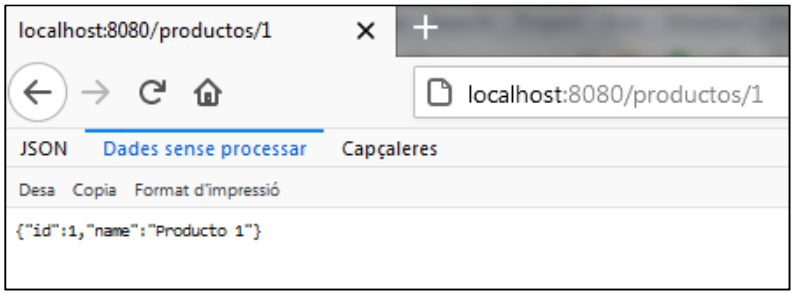
ControllerRest.java

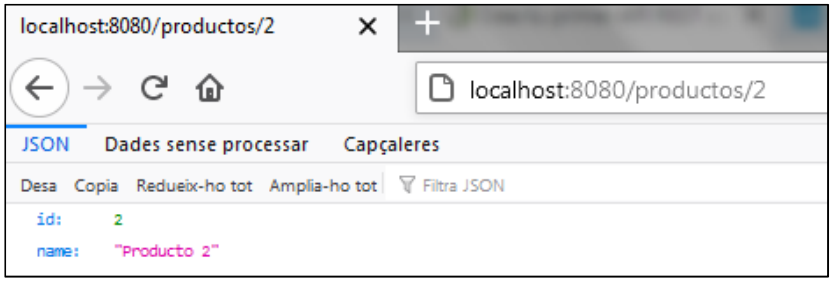
```
1  package com.example.demo.rest;
2
3  import java.util.List;
16
17 @RestController                //Indica que esta clase va a ser un servicio REST
18 @RequestMapping("/productos")  //En que URL se va a exponer los servicios de esta clase
19 public class ControllerRest {
20
21     @Autowired           //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
```

6. API REST GET

Paso 24) Ahora buscamos uno en concreto.







6. API REST POST

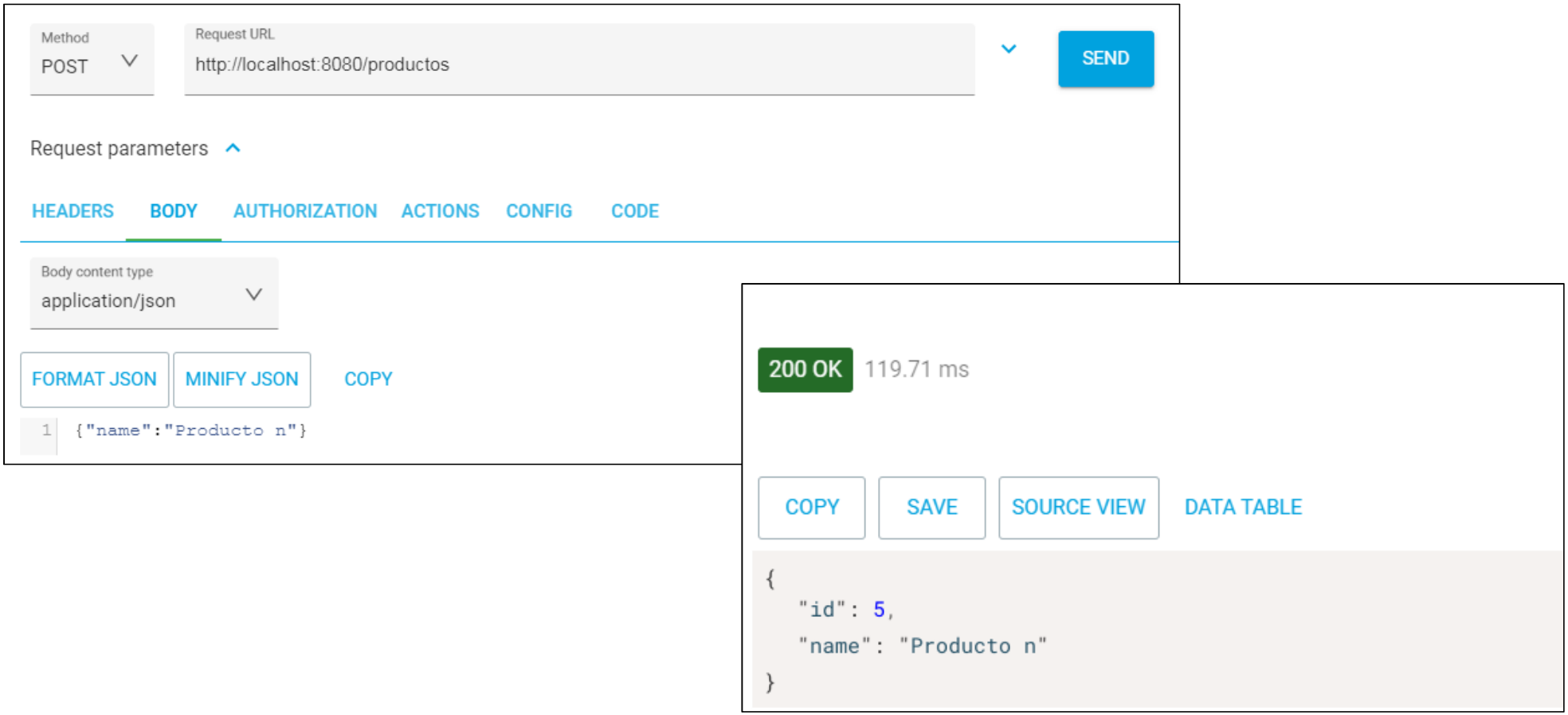
Paso 25) Ahora vamos a ver la inserción de producto a través de post

ControllerRest.java

```java
package com.example.demo.rest;

import java.util.List;

@RestController                 //Indica que esta clase va a ser un servicio REST
@RequestMapping("/productos")   //En que URL se va a exponer los servicios de esta clase
public class ControllerRest {

    @Autowired              //Inyeccion de dependencias
    private ProductosDAO productosDAO;

    @GetMapping
    public ResponseEntity<List<Producto>> getProducto() {
        List<Producto> productos = productosDAO.findAll();
        return ResponseEntity.ok(productos);
    }

    @PostMapping    //productos (POST)
    public ResponseEntity<Producto> crearProducto(@RequestBody Producto producto) {
        Producto newProduct = productosDAO.save(producto);
        return ResponseEntity.ok(newProduct);
    }

    @RequestMapping(value="{productId}")   //productos/{productId} --> productos/1
    public ResponseEntity<Producto> getProductoById(@PathVariable("productId") Long productId) {
        Optional<Producto> optionalProducto = productosDAO.findById(productId);
        if (optionalProducto.isPresent()) {
            return ResponseEntity.ok(optionalProducto.get());
        }else {
            return ResponseEntity.noContent().build();
        }
    }
```

6. API REST POST

Paso 25) Haremos la inserción desde el plugin advanced rest client



6. API REST DELETE

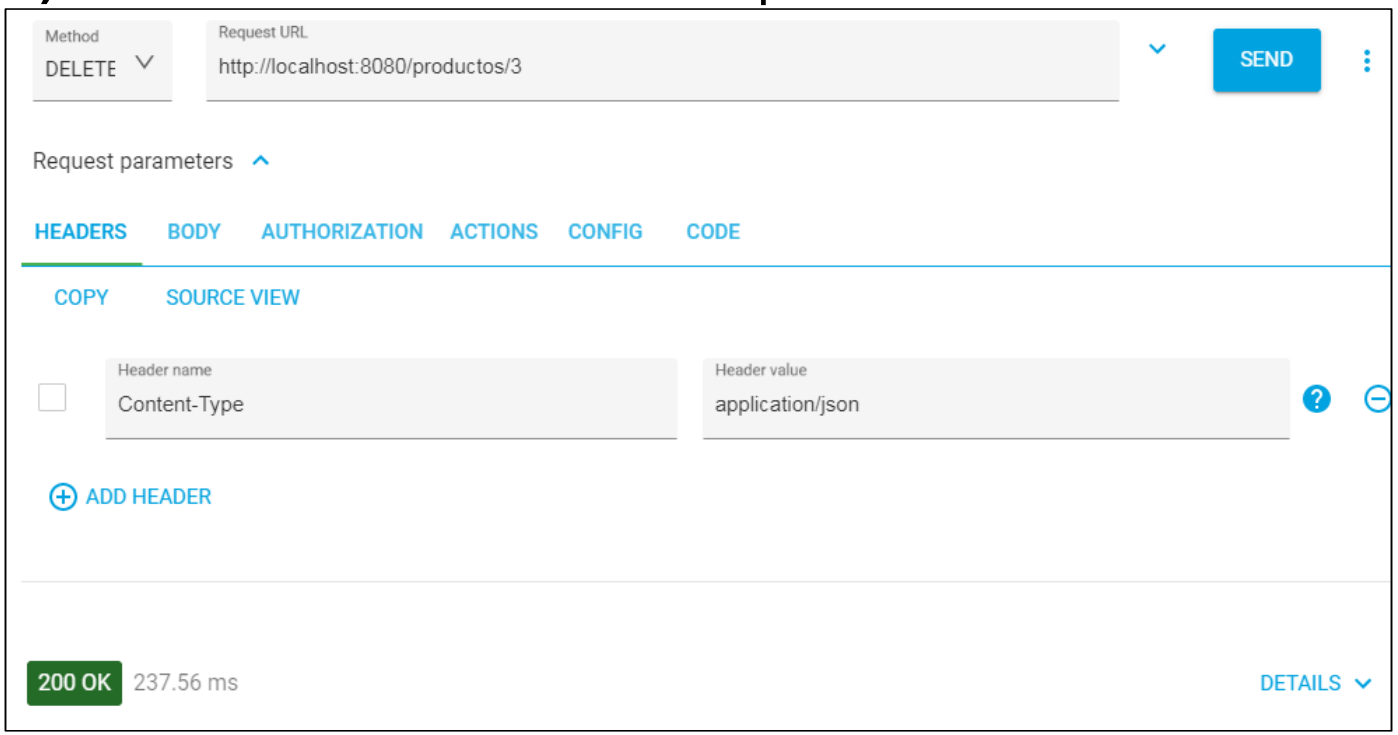
Paso 25) Haremos el borrado de un producto

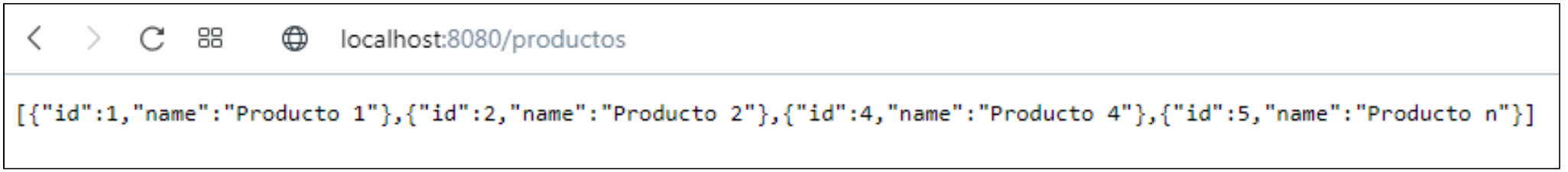
ControllerRest.java

```java
package com.example.demo.rest;

import java.util.List;

@RestController                 //Indica que esta clase va a ser un servicio REST
@RequestMapping("/productos")   //En que URL se va a exponer los servicios de esta clase
public class ControllerRest {

    @Autowired          //Inyeccion de dependencias
    private ProductosDAO productosDAO;

    @GetMapping
    public ResponseEntity<List<Producto>> getProducto() {
        List<Producto> productos = productosDAO.findAll();
        return ResponseEntity.ok(productos);
    }

    @PostMapping    //productos (POST)
    public ResponseEntity<Producto> crearProducto(@RequestBody Producto producto) {
        Producto newProduct = productosDAO.save(producto);
        return ResponseEntity.ok(newProduct);
    }

    @DeleteMapping(value="{productoId}")    //productos/{productId}  (DELETE)
    public ResponseEntity<Void> deleteProducto(@PathVariable("productoId") Long productoId) {
        productosDAO.deleteById(productoId);
        return ResponseEntity.ok(null);
    }

}
```

6. API REST DELTET

Paso 25) Haremos el borrado de un producto





6. API REST PUT

Paso 25) Haremos el borrado de un producto

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
```

6. API REST PUT

Paso 25) Haremos el borrado de un producto