

CREACION DE UN SERVICIO API REST SENCILLO

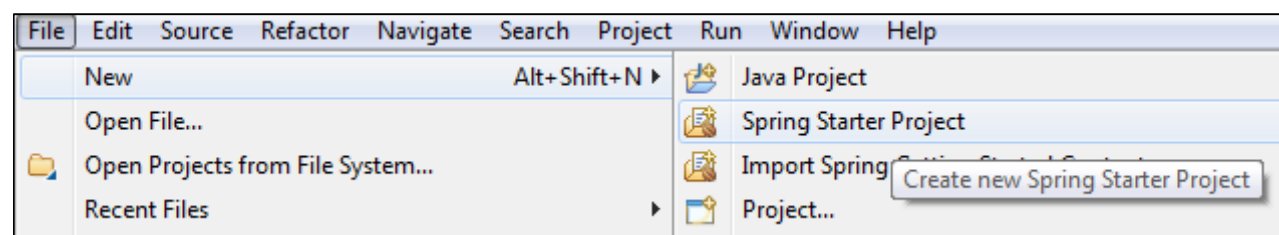
Eduard Lara

INDICE

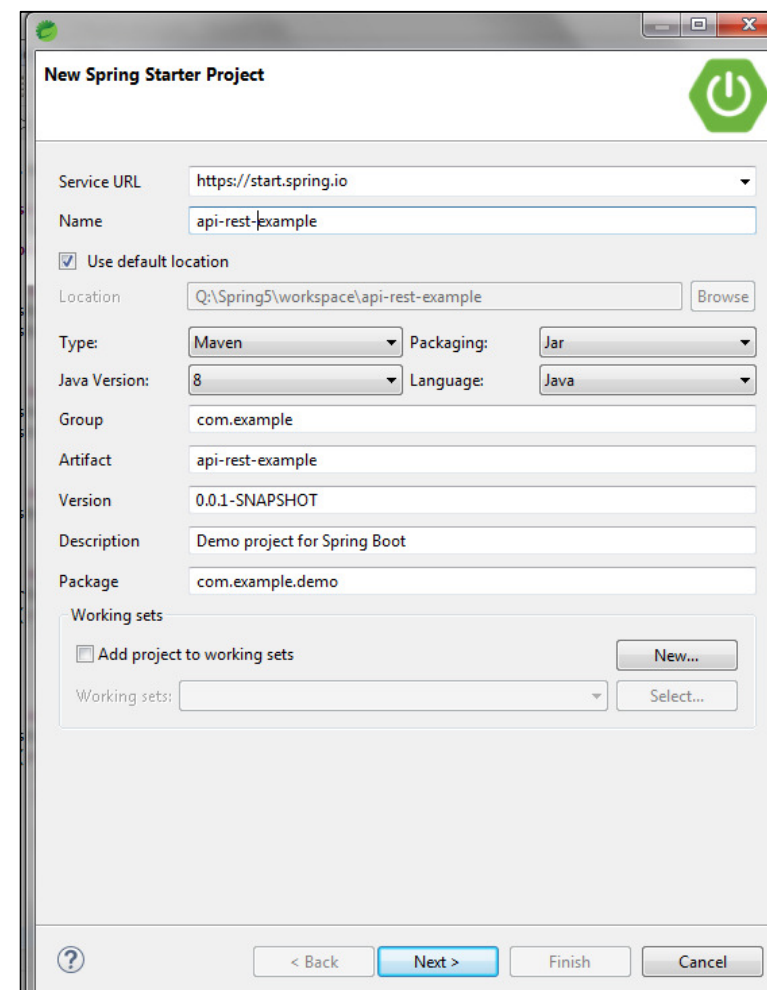
1. Inyección de dependencia
2. Modelo vista controlador

1. CREACION PROYECTO

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



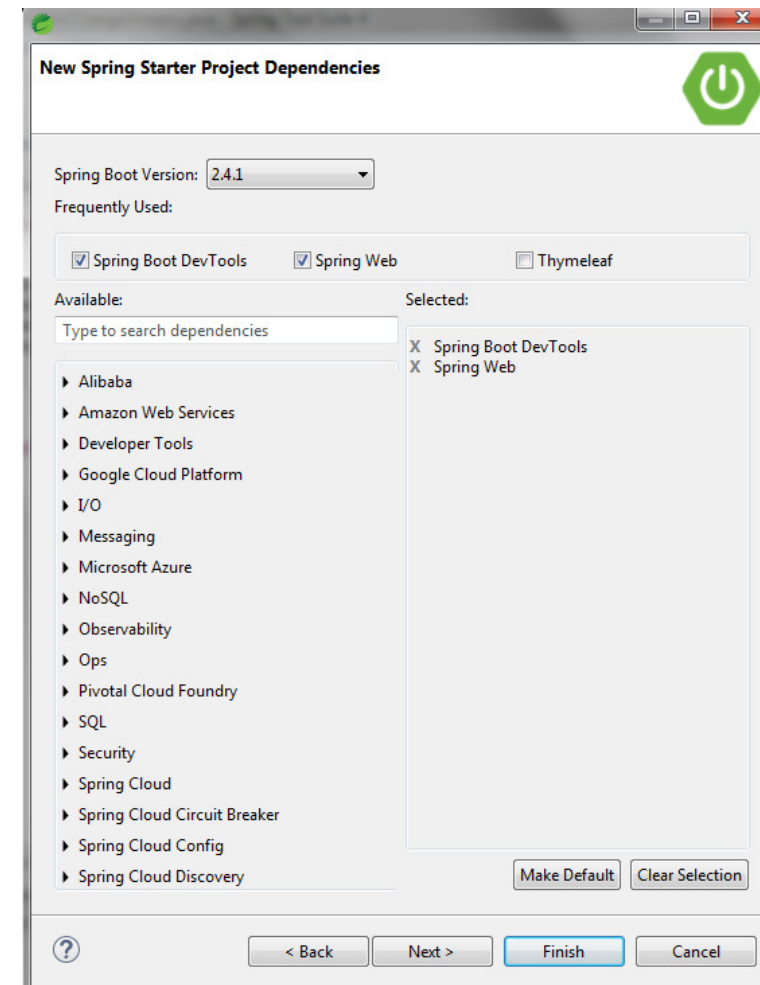
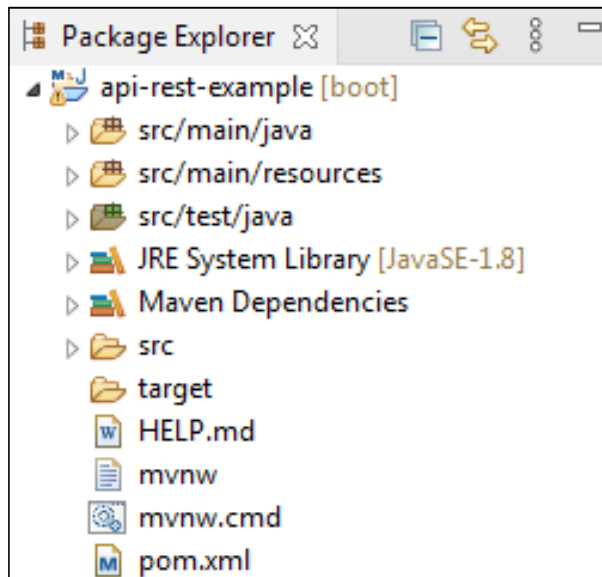
Podemos dejar por defecto los valores que nos presenta el wizard. Si se desea se puede cambiar el nombre de proyecto, el package raíz, el tipo de proyecto (Maven o Gradle) y/o la versión de Java.



1. CREACION PROYECTO

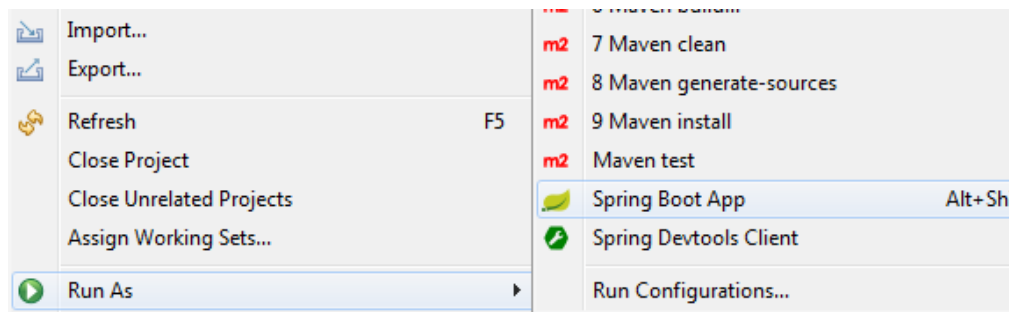
Paso 2) Agregamos las librerías:

- 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

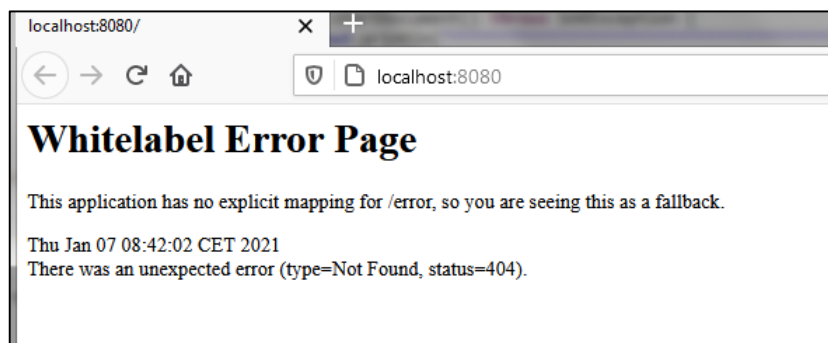
Paso 3) Probamos de ejecutar el proyecto, para ello levantamos el servidor Tomcat haciendo Run As/Spring Boot App. Una vez vemos que ha arrancado correctamente el servidor, vamos a un navegador y ponemos **localhost:8080**. Nos da error porque no tenemos ninguna página de inicio. Pero también significa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



```

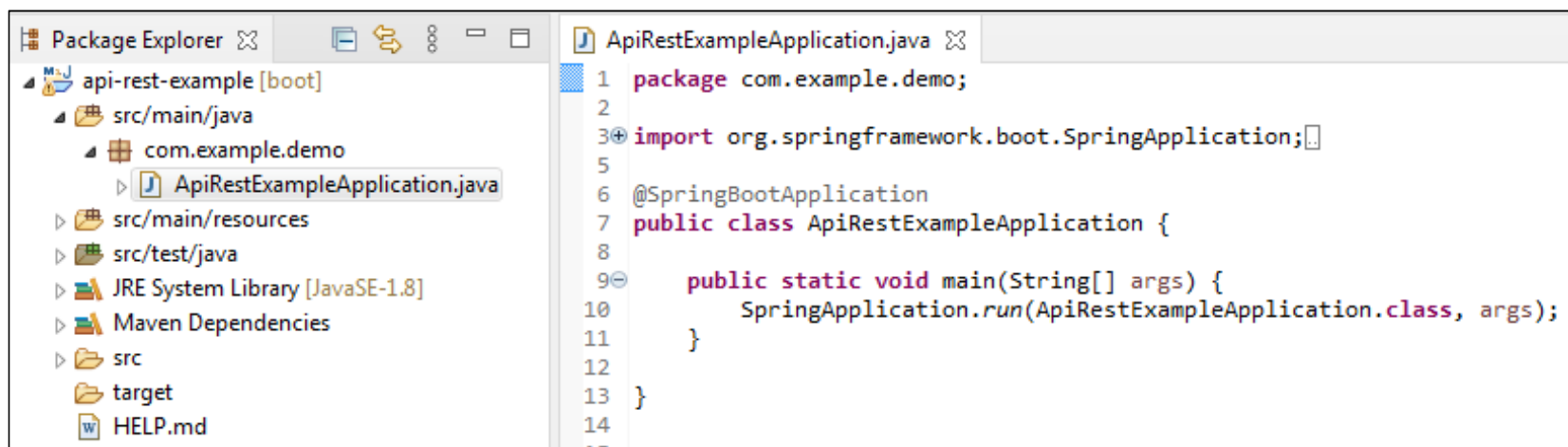
2021-01-07 08:41:46.209 INFO 16080 --- [main] c.e.demo.ApiRestExampleApplication : Starting ApiRestExampleApplication using Java 15.0
2021-01-07 08:41:46.214 INFO 16080 --- [main] c.e.demo.ApiRestExampleApplication : No active profile set, falling back to default profile
2021-01-07 08:41:47.260 INFO 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] org.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] o.a.c.c.C.[Tomcat].[localhost].[/] : Started ApiRestExampleApplication in 1.994 seconds
2021-01-07 08:42:02.577 INFO 16080 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : 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

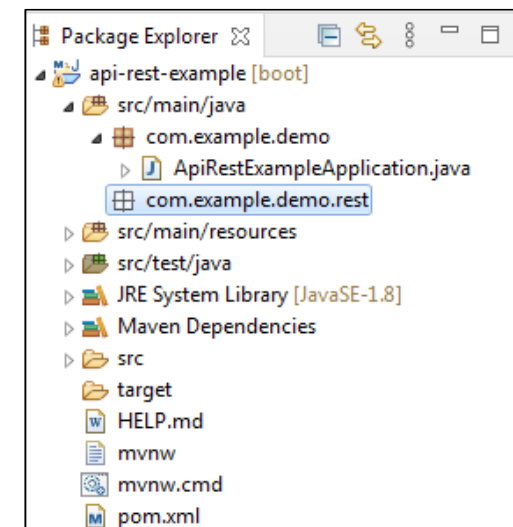
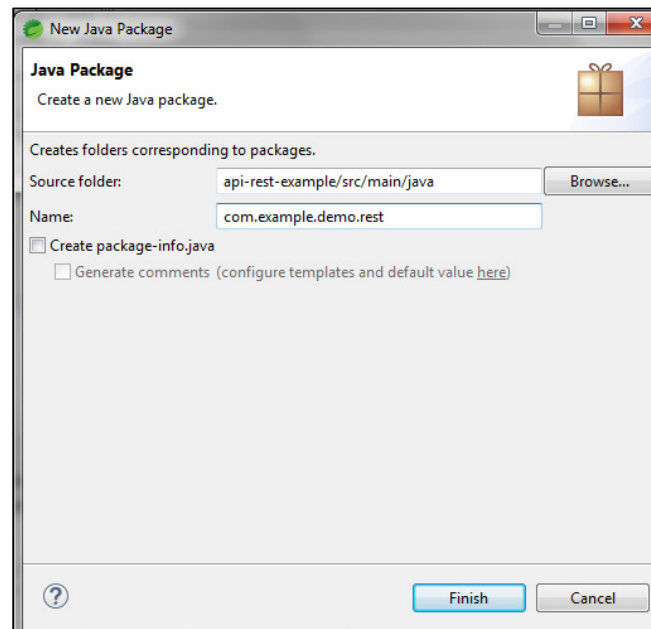
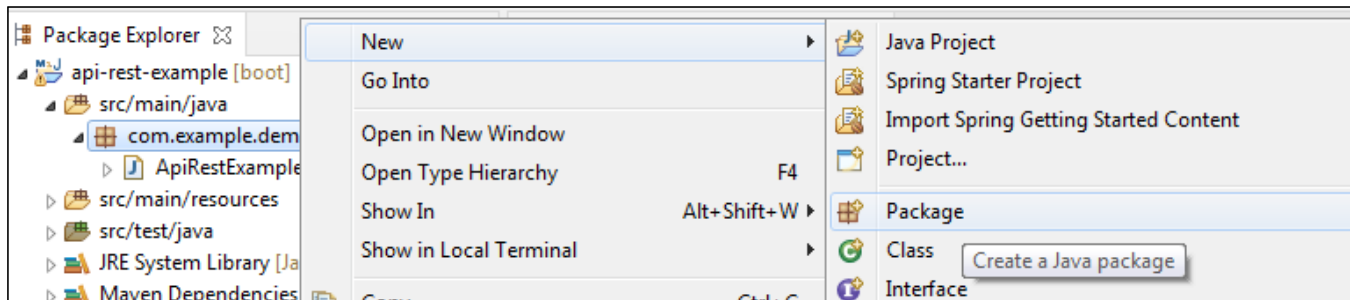
Paso 4) Podemos observar en el package raíz indicado al principio en la creación del proyecto, la clase generada automáticamente que inicia nuestro servidor y la aplicación:



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

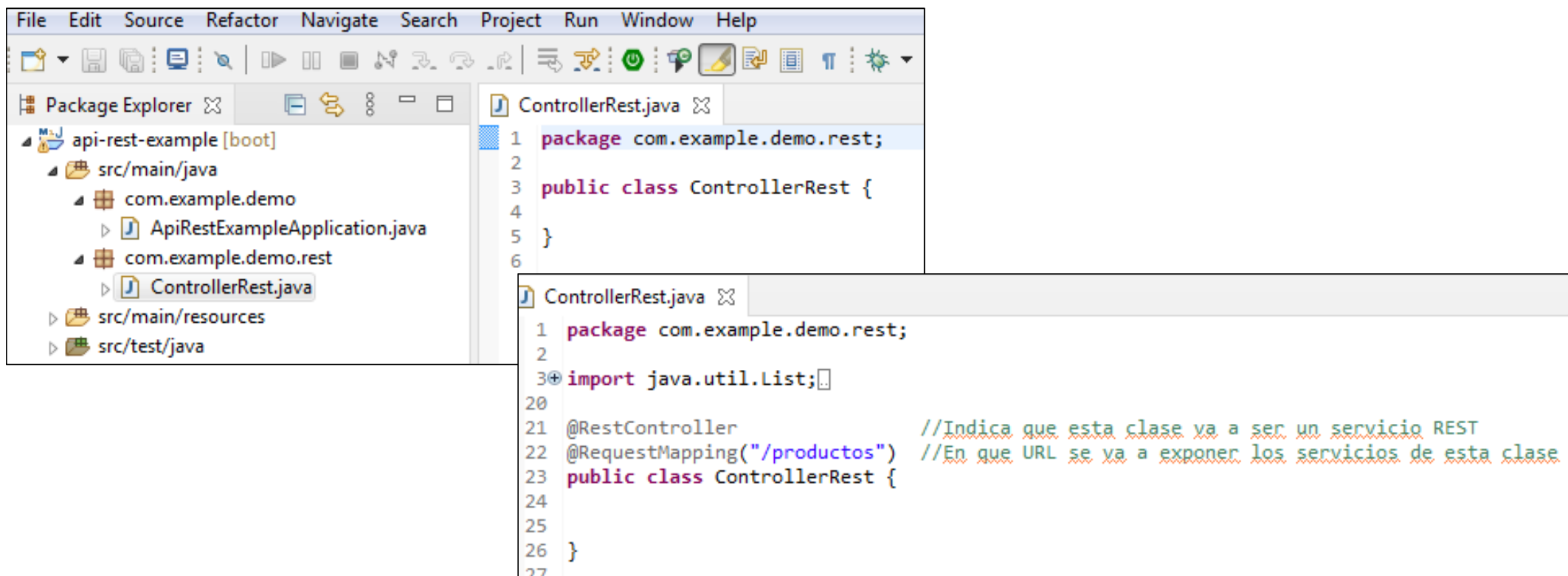
2. CONTROLADOR REST

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:



The screenshot shows an IDE with the Package Explorer on the left and the ControllerRest.java file open in the editor. The Package Explorer shows the project structure: api-rest-example [boot] with sub-packages src/main/java, src/main/resources, and src/test/java. Under src/main/java, there are two packages: com.example.demo and com.example.demo.rest. The ControllerRest.java file is located in com.example.demo.rest. The editor shows the following code:

```
1 package com.example.demo.rest;
2
3 public class ControllerRest {
4
5 }
6
```

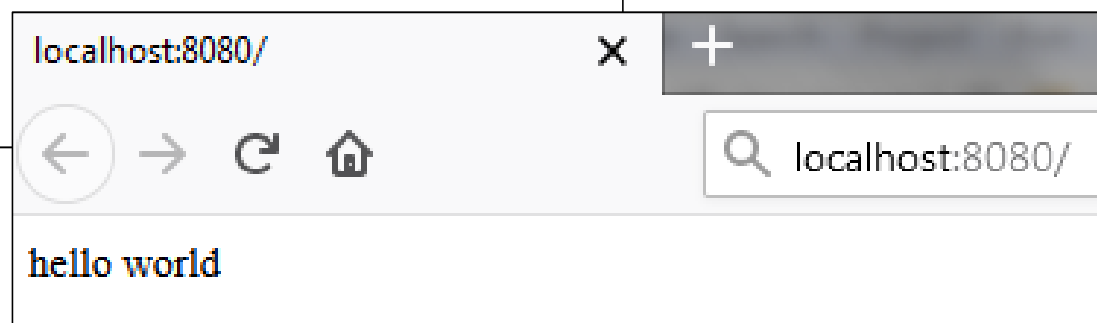
Below the editor, there is a detailed view of the ControllerRest.java file with annotations:

```
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 }
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/

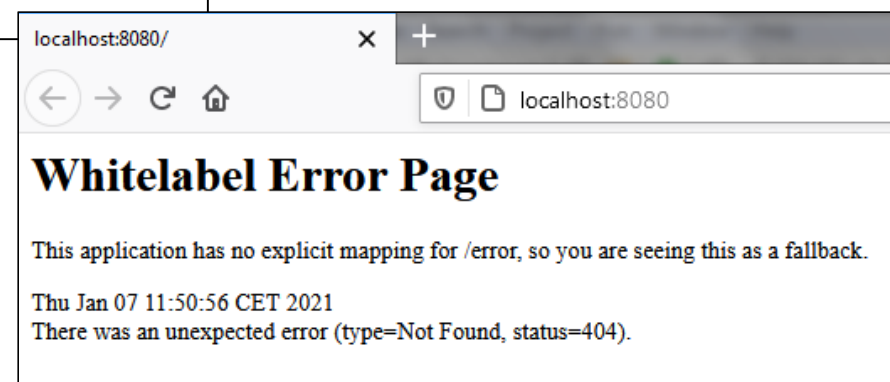
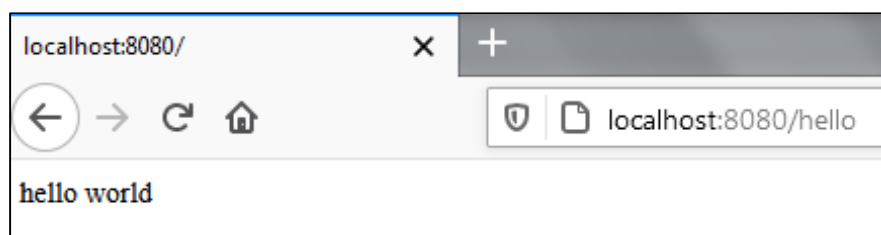
```
ControllerRest.java ✕
1 package com.example.demo.rest;
2
3 import org.springframework.web.bind.annotation.GetMapping;
4
5
6
7
8 @RestController           //Indica que esta clase va a ser un servicio REST
9 @RequestMapping("/")     //En que URL se va a exponer los servicios de esta clase
10 public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

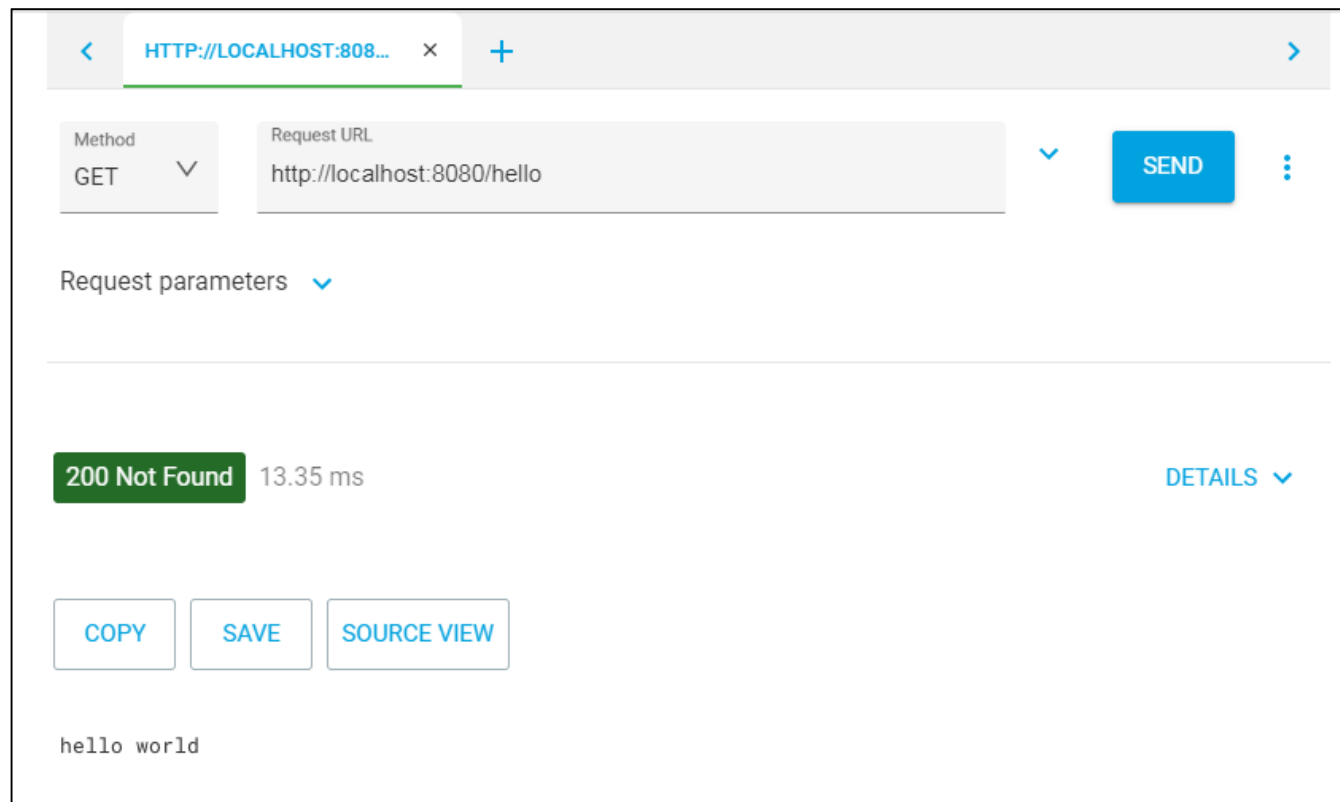
Paso 4) Si añadimos en el GetMapping el path “hello”, entonces la función daría servicio en la url localhost:8080/hello:

```
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 }
```



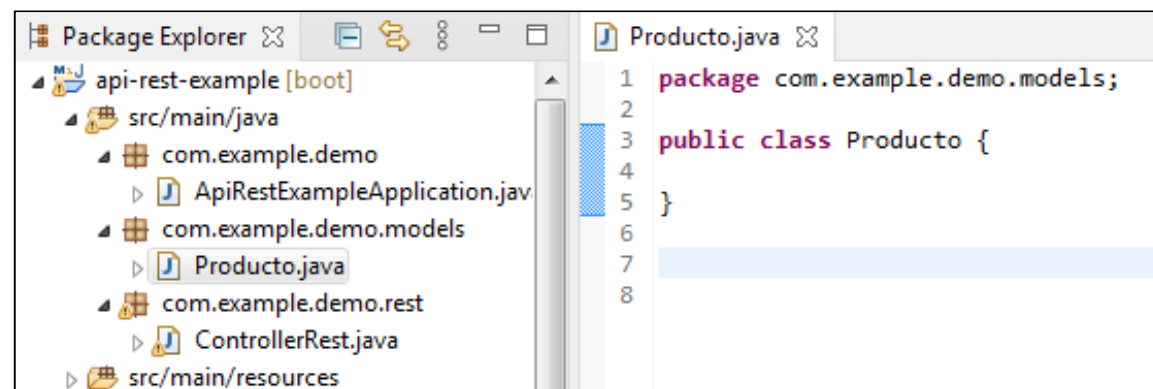
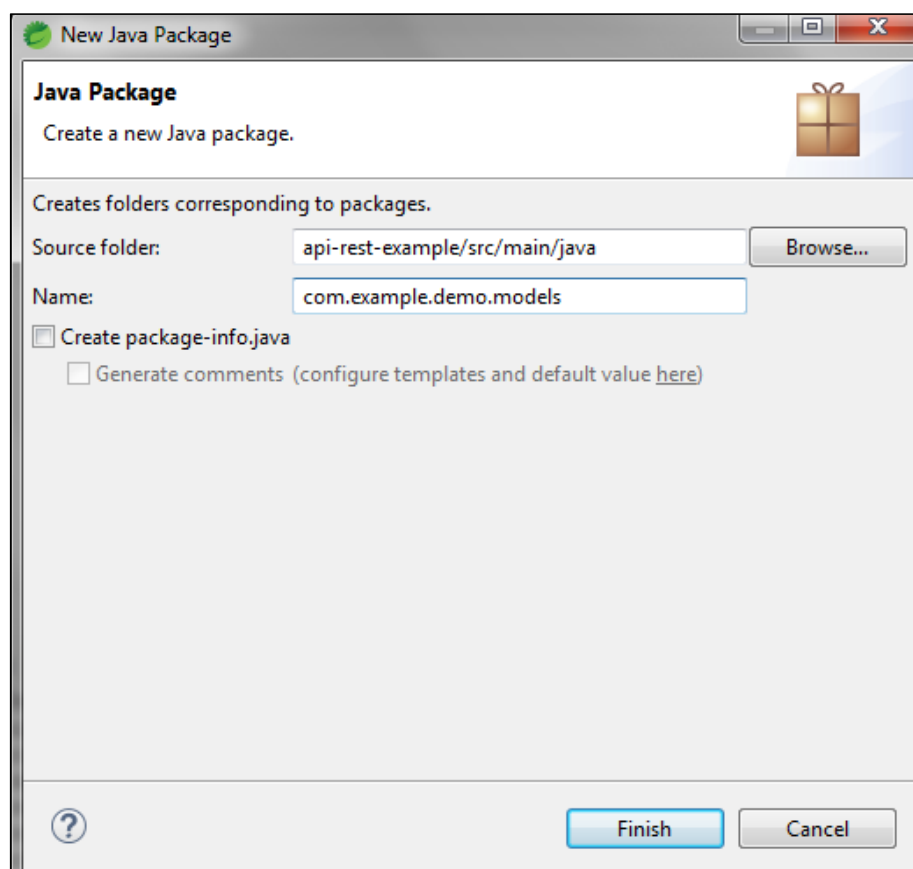
2. CONTROLADOR REST

Paso 5) Podemos probar el servicio con 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

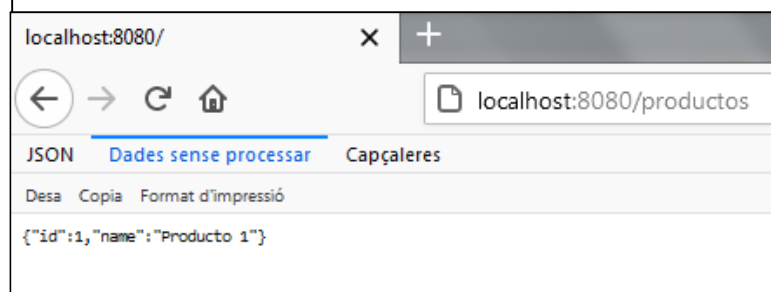
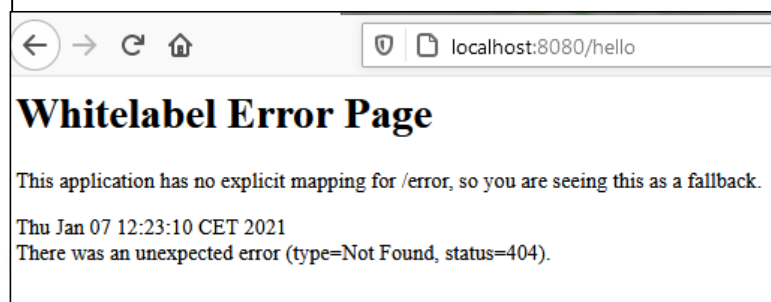
Paso 2) Creamos dos atributos simples en la clase POJO Producto, la cual representará el modelo de los datos de la base de datos:

```
Producto.java
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 }
23
24
25
26
27
28
29
```



3. MODELO DE DATOS

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

Method
GET ▼

Request URL
http://localhost:8080/productos

Request parameters ▼

200 Bad Request 19.77 ms

COPY

SAVE

SOURCE VIEW

DATA TABLE

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

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
20
21 <dependencies>
22   <dependency>
23     <groupId>org.springframework.boot</groupId>
24     <artifactId>spring-boot-starter-web</artifactId>
25   </dependency>
26
27   <dependency>
28     <groupId>org.springframework.boot</groupId>
29     <artifactId>spring-boot-starter-test</artifactId>
30     <scope>test</scope>
31   </dependency>
32
33   <dependency>
34     <groupId>mysql</groupId>
35     <artifactId>mysql-connector-java</artifactId>
36     <version>5.1.6</version>
37   </dependency>
38
39   <dependency>
40     <groupId>org.springframework.boot</groupId>
41     <artifactId>spring-boot-starter-data-jpa</artifactId>
42   </dependency>
43
44 </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.

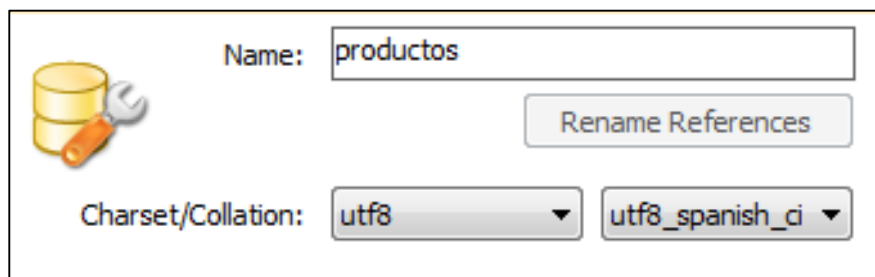
@Column mapeamos el atributo con la columna de la tabla indicada en @Table.

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

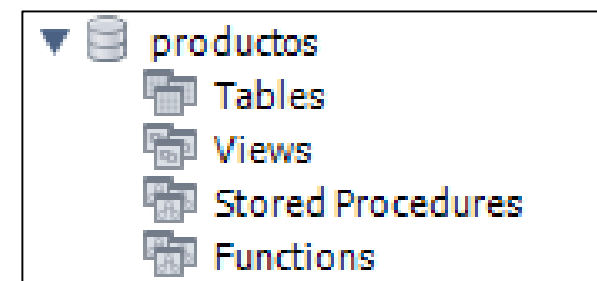
```
Producto.java
4 import javax.persistence.Entity;
5 import javax.persistence.GeneratedValue;
6 import javax.persistence.GenerationType;
7 import javax.persistence.Id;
8 import javax.persistence.Table;
9
10 @Entity
11 @Table(name="productos")
12 public class Producto {
13
14     @Id
15     @Column(name="id")
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:



MySQL Database Creation Wizard interface. The 'Name' field contains 'productos'. The 'Charset/Collation' section shows 'utf8' for the character set and 'utf8_spanish_ci' for the collation. A 'Rename References' button is visible.



```
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;
4
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("Producto");
22         return null;
23         //return ResponseEntity.ok(producto);
24     }
25 }
```

```
*****
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:

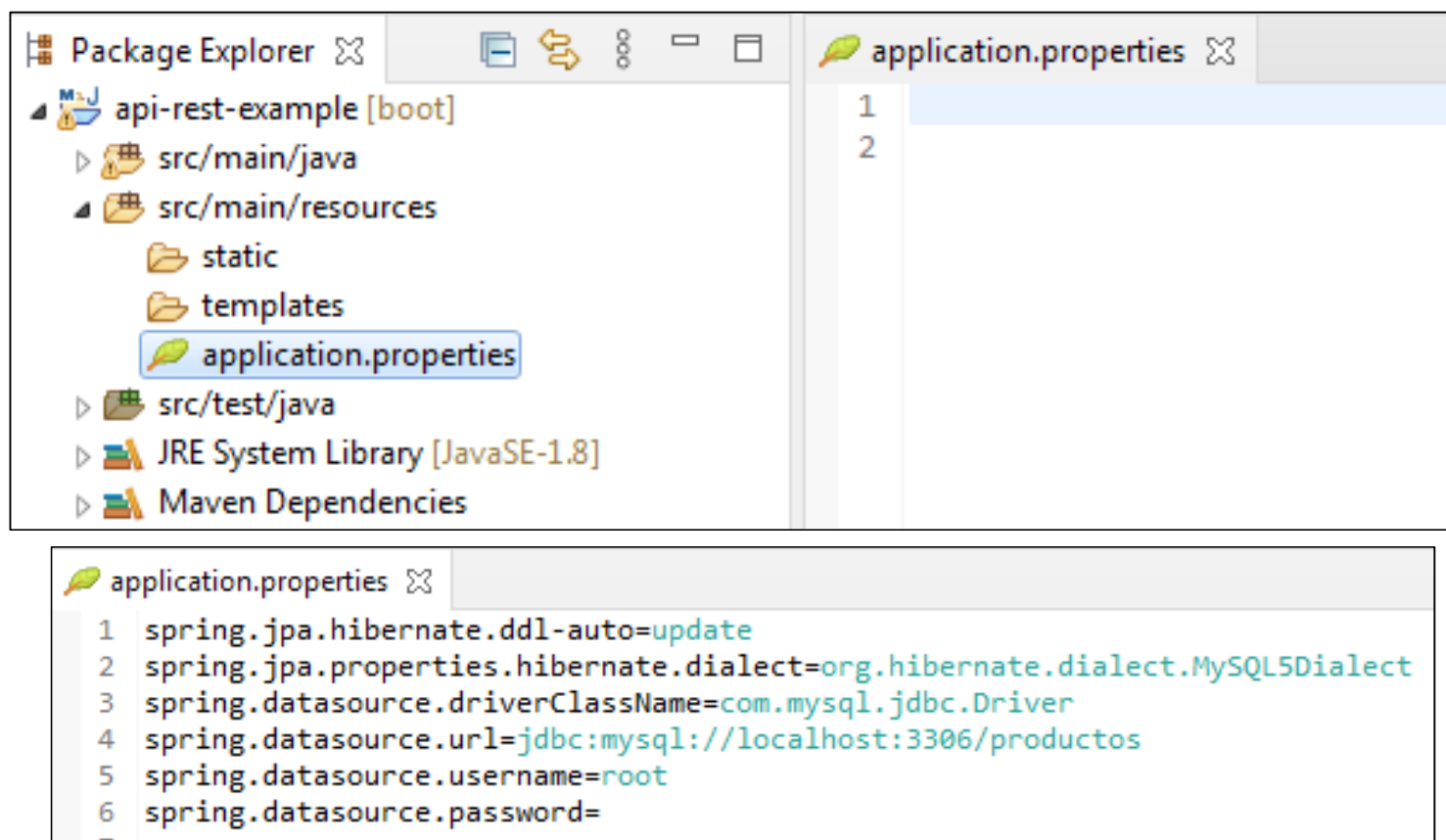
Consider the following:

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:



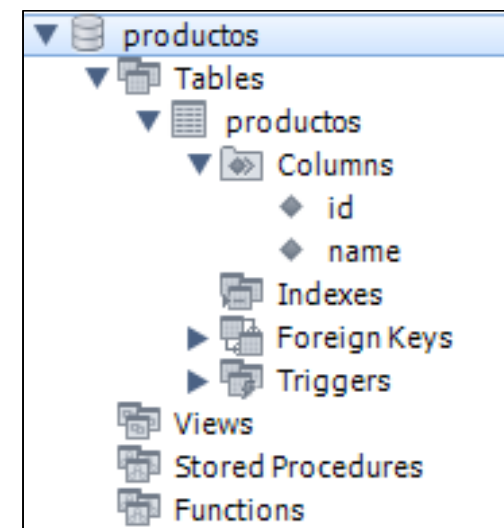
4. JDBC Y JPA

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

```
api-rest-example - ApiRestExampleApplication [Spring Boot App]

:: Spring Boot :: (v2.4.1)

2021-01-07 15:51:44.734 INFO 9148 --- [main] c.e.demo.ApiRestExampleApplication : Starting ApiRestExampleApplication using Java 15.0.1 on HP-papa with PID
2021-01-07 15:51:44.738 INFO 9148 --- [main] c.e.demo.ApiRestExampleApplication : No active profile set, falling back to default profiles: default
2021-01-07 15:51:45.628 INFO 9148 --- [main] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JPA repositories in DEFAULT mode.
2021-01-07 15:51:45.643 INFO 9148 --- [main] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 5 ms. Found 0 JPA repository
2021-01-07 15:51:46.314 INFO 9148 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2021-01-07 15:51:46.328 INFO 9148 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2021-01-07 15:51:46.329 INFO 9148 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.41]
2021-01-07 15:51:46.477 INFO 9148 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2021-01-07 15:51:46.477 INFO 9148 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 1678 ms
2021-01-07 15:51:46.752 INFO 9148 --- [main] o.hibernate.jpa.internal.util.LogHelper : HHH000204: Processing PersistenceUnitInfo [name: default]
2021-01-07 15:51:46.830 INFO 9148 --- [main] org.hibernate.Version : HHH000412: Hibernate ORM core version 5.4.25.Final
2021-01-07 15:51:47.033 INFO 9148 --- [main] o.hibernate.annotations.common.Version : HCANN000001: Hibernate Commons Annotations {5.1.2.Final}
2021-01-07 15:51:47.205 INFO 9148 --- [main] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
2021-01-07 15:51:47.468 INFO 9148 --- [main] com.zaxxer.hikari.pool.PoolBase : HikariPool-1 - Driver does not support get/set network timeout for connections
2021-01-07 15:51:47.472 INFO 9148 --- [main] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.
2021-01-07 15:51:47.494 INFO 9148 --- [main] org.hibernate.dialect.Dialect : HHH000400: Using dialect: org.hibernate.dialect.MySQL5Dialect
2021-01-07 15:51:48.207 INFO 9148 --- [main] o.h.e.t.j.p.i.JtaPlatformInitiator : HHH000490: Using JtaPlatform implementation: [org.hibernate.engine.transaction.jta.platform.internal.NoJtaPlatform]
2021-01-07 15:51:48.217 INFO 9148 --- [main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2021-01-07 15:51:48.287 WARN 9148 --- [main] jpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may
2021-01-07 15:51:48.448 INFO 9148 --- [main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2021-01-07 15:51:48.740 INFO 9148 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2021-01-07 15:51:48.750 INFO 9148 --- [main] c.e.demo.ApiRestExampleApplication : Started ApiRestExampleApplication in 4.431 seconds (JVM running for 5.6s)
```



5. CLASE DAO

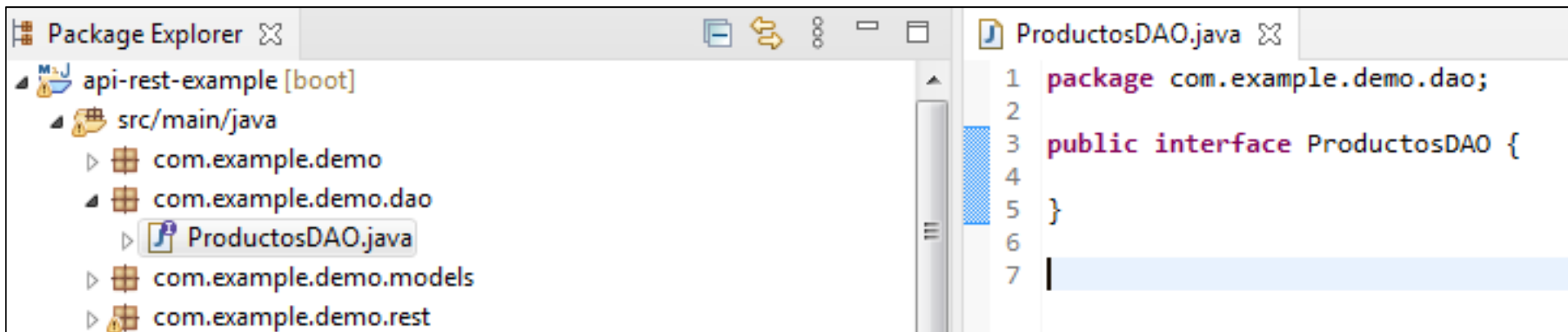
Paso 1) La clase DAO (Data Access Object) es un componente de software que suministra una interfaz común entre la aplicación y uno o más dispositivos de almacenamiento de datos, tales como una Base de datos o un archivo.

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.

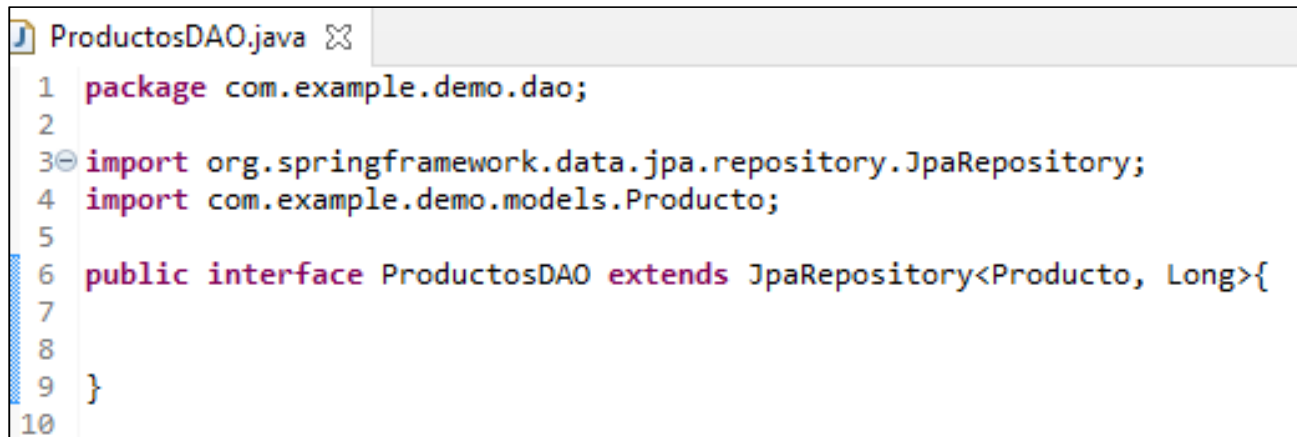


The screenshot shows the Package Explorer on the left with the following structure:

- api-rest-example [boot]
 - src/main/java
 - com.example.demo
 - com.example.demo.dao
 - ProductosDAO.java
 - com.example.demo.models
 - com.example.demo.rest

The main editor shows the initial code for ProductosDAO.java:

```
1 package com.example.demo.dao;
2
3 public interface ProductosDAO {
4
5 }
6
7
```



The screenshot shows the updated code for ProductosDAO.java:

```
1 package com.example.demo.dao;
2
3 import org.springframework.data.jpa.repository.JpaRepository;
4 import com.example.demo.models.Producto;
5
6 public interface ProductosDAO extends JpaRepository<Producto, Long>{
7
8 }
9
10
```

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.

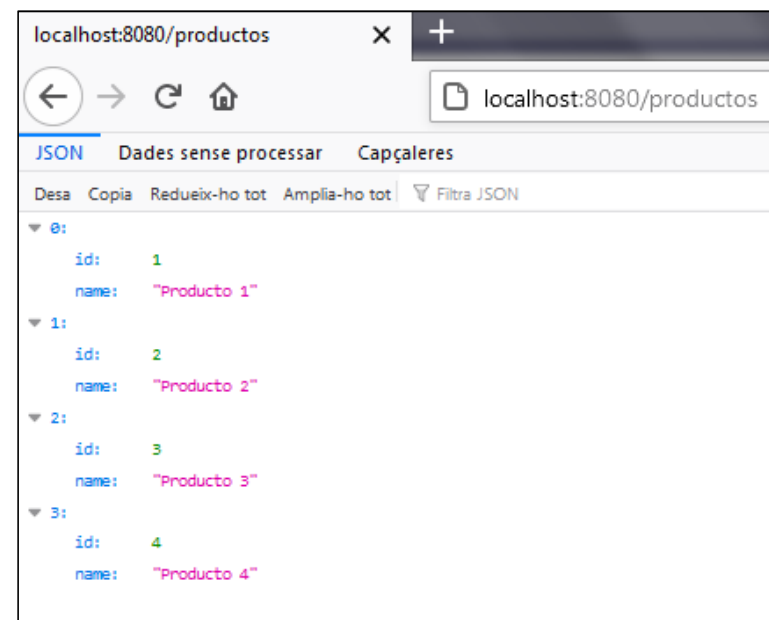
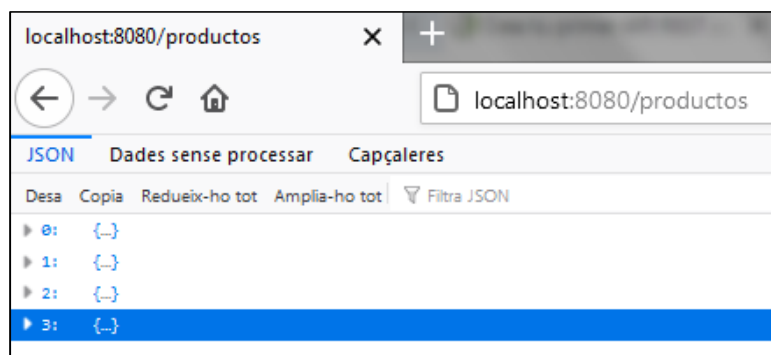
Esto recibe el nombre de inyección de dependencias: dejo que el sistema llame a una clase que implemente dicha interfaz y de esta manera ya podemos utilizar las funciones de dicha interfaz que se corresponde con las funciones de JpaRepository

```
ControllerRest.java
1 package com.example.demo.rest;
2
3 import java.util.List;
4
14 @RestController //Indica que esta clase va a ser un servicio REST
15 @RequestMapping("/productos") //En que URL se va a exponer los servicios de esta clase
16 public class ControllerRest {
17
18     @Autowired //Inyeccion de dependencias
19     private ProductosDAO productosDAO;
20
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

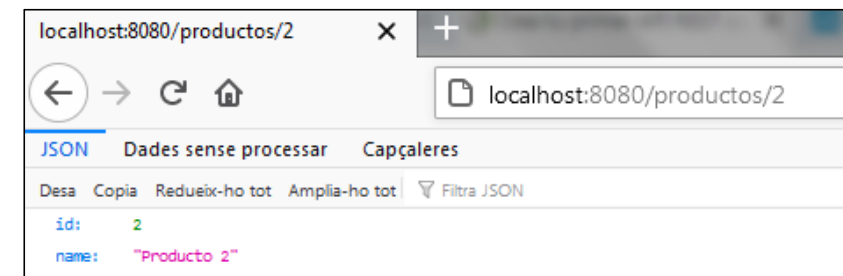
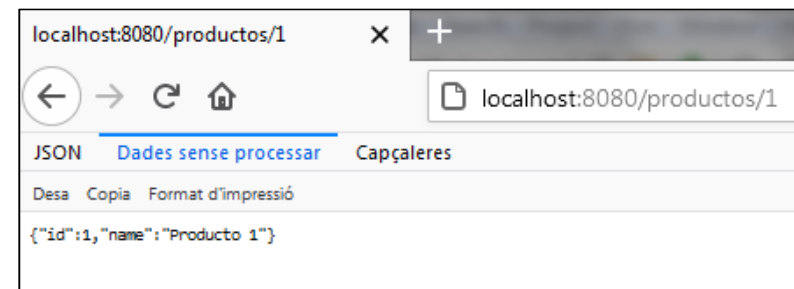
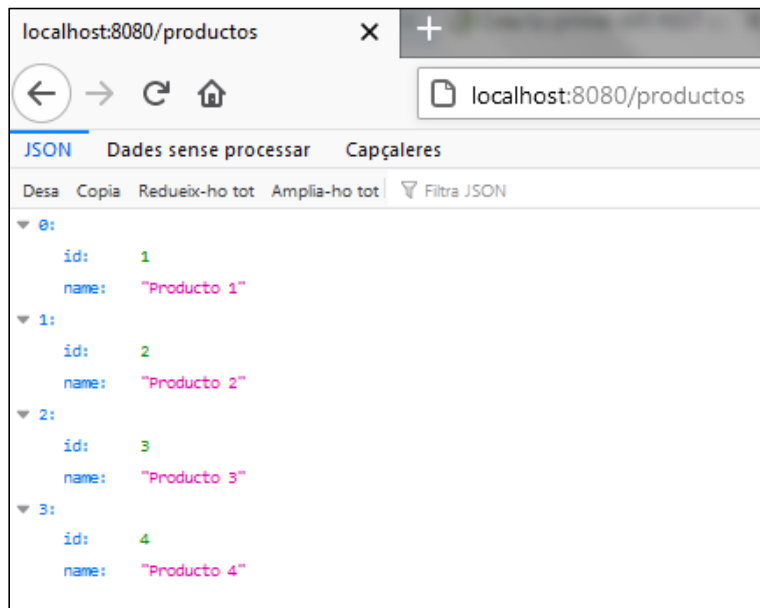
Paso 2) Creamos el servicio que nos permita leer un producto en concreto mediante GET usando la anotación `@PathVariable`.

Se deberá usar la url `localhost:8080/productos/{productId}` de manera que el se información sobre el producto que se desea lo extraera de rviceo 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.



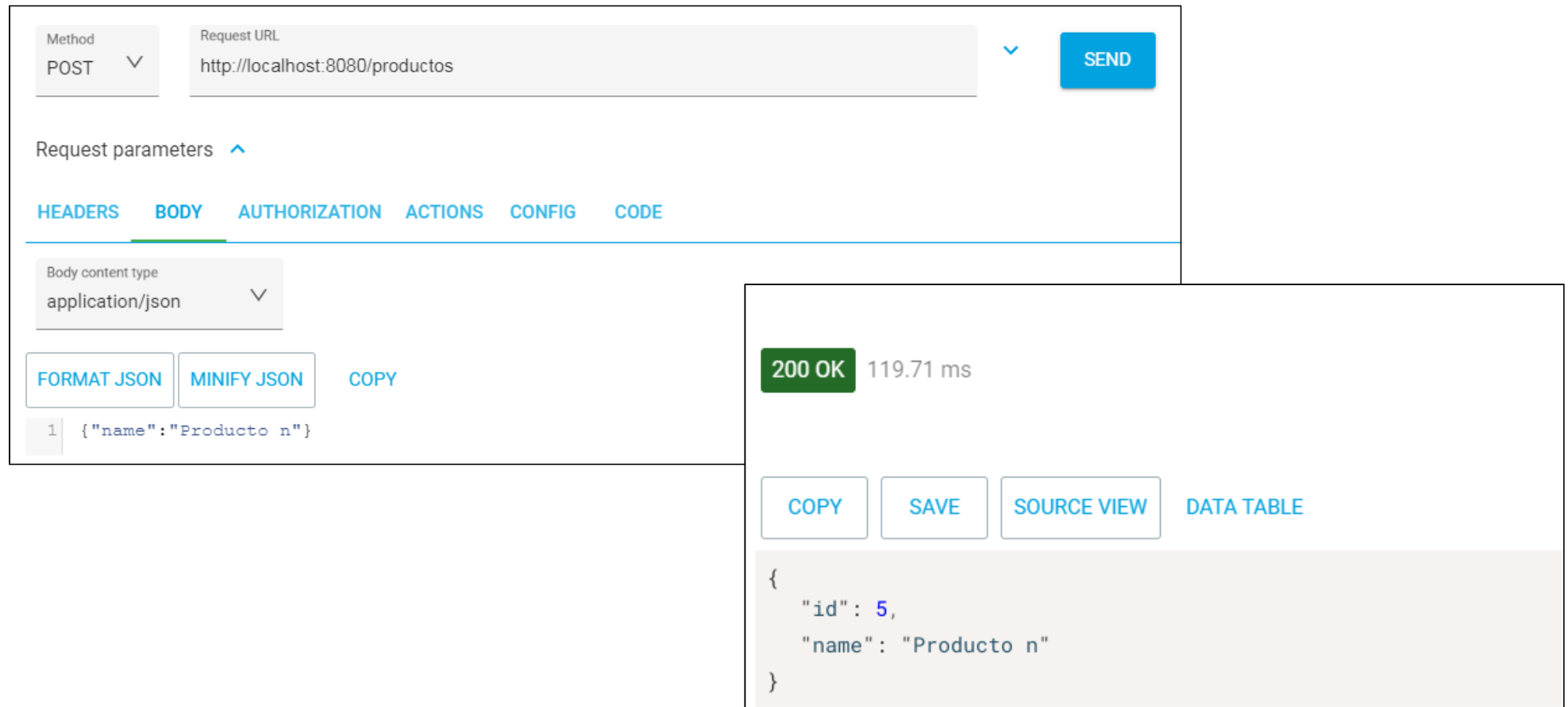
6. API REST POST

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

```
ControllerRest.java
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 }
```

6. API REST POST

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



The screenshot displays the Advanced REST Client interface. The top section shows the request configuration: Method is POST, Request URL is http://localhost:8080/productos, and a blue SEND button is on 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 are buttons for FORMAT JSON, MINIFY JSON, and COPY. The body content is {"name": "Producto n"}. The bottom right panel shows the response: a green status bar indicates 200 OK with a response time of 119.71 ms. Below this are buttons for COPY, SAVE, SOURCE VIEW, and DATA TABLE. The response body is displayed in a light gray box with the following JSON:

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

6. API REST DELETE

Paso 25) Haremos el borrado de un producto

```
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 productId) {
41         productosDAO.deleteById(productId);
42         return ResponseEntity.ok(null);
43     }
44 }
```

6. API REST DELTET

Paso 25) Haremos el borrado de un producto

Method
DELETE

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

SEND

Request parameters

HEADERS

BODY

AUTHORIZATION

ACTIONS

CONFIG

CODE

COPY

SOURCE VIEW

☐

Header name
Content-Type

Header value
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"}]
```

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

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"}]
```