IES DR LLUÍS SIMARRO
ENSENYAMENT PÚBLIC
XATIVA

Acceso a datos

UD06. Creación de servicios web: Spring Boot

Desarrollo de Aplicaciones Multiplataforma

Enric Giménez Ribes

e.gimenezribes@edu.gva.es

ÍNDICE

1. INTRODUCCIÓN

Spring es un framework de Java para el desarrollo de aplicaciones y servicios web. En nuestro caso, lo que queremos construir es una pequeña aplicación web con una Base de Datos y que podamos, si así lo queremos, proporcionar algo de lógica en el lado servidor cuando sea necesario. Para eso utilizaremos **Spring Boot** que es una parte de este framework que facilita bastante el trabajo para casos como el que a nosotros nos interesa.

- https://spring.io/

Spring Framework comprende diversos módulos que proveen un rango de servicios:

- Contenedor de inversión de control (IoC): permite la creación de los componentes de aplicación y la administración del ciclo de vida de los objetos Java, se lleva a cabo principalmente a través de la inyección de dependencias.
- Programación orientada a aspectos: habilita la implementación de rutinas transversales.
- Acceso a datos: se trabaja con SGBD, usando Java DataBase Connectivity (JDBC) y herramientas de Mapeo Objeto Relacional (ORM).
- Gestión de transacciones: unifica distintas APIs de gestión y coordina las transacciones para los objetos Java.
- Modelo-vista-controlador: Un framework basado en HTTP y servlets, que provee herramientas para la extensión y personalización de aplicaciones web y servicios web REST.
- Framework de acceso remoto: Permite la importación y exportación estilo RPC, de objetos Java a través de redes que soporten RMI, CORBA y protocolos basados en HTTP incluyendo servicios web (SOAP).

- Convención sobre Configuración: el módulo Spring Roo ofrece una solución rápida para el desarrollo de aplicaciones basadas en Spring Framework, privilegiando la simplicidad sin perder flexibilidad.
- Procesamiento por lotes: un framework para procesamiento de mucho volumen que como características incluye funciones de registro/trazado, manejo de transacciones, estadísticas de procesamiento de tareas, reinicio de tareas, y manejo de recursos.
- Autenticación y Autorización: procesos de seguridad configurables que soportan un rango de estándares, protocolos, herramientas y prácticas a través del subproyecto Spring Security.
- Mensajes: Registro configurable de objetos receptores de mensajes, para el consumo transparente a través de JMS, una mejora del envío de mensajes sobre las API JMS estándar.
- Testing: Soporte de clases para desarrollo de unidades de prueba e integración.

1.1. Spring Boot

Spring Boot es una de las tecnologías dentro del mundo de Spring de las que más se está hablando últimamente. ¿Qué es y cómo funciona Spring Boot? Para entender el concepto primero debemos reflexionar sobre cómo construimos aplicaciones con Spring Framework.

Fundamentalmente, existen tres pasos a realizar. El primero es crear un proyecto Maven/Gradle y descargar las dependencias necesarias. En segundo lugar, desarrollamos la aplicación y en tercer lugar, la desplegamos en un servidor. Si nos ponemos a pensar un poco a detalle en el tema, únicamente el paso dos es una tarea de desarrollo. Los otros pasos están más orientados a infraestructura.

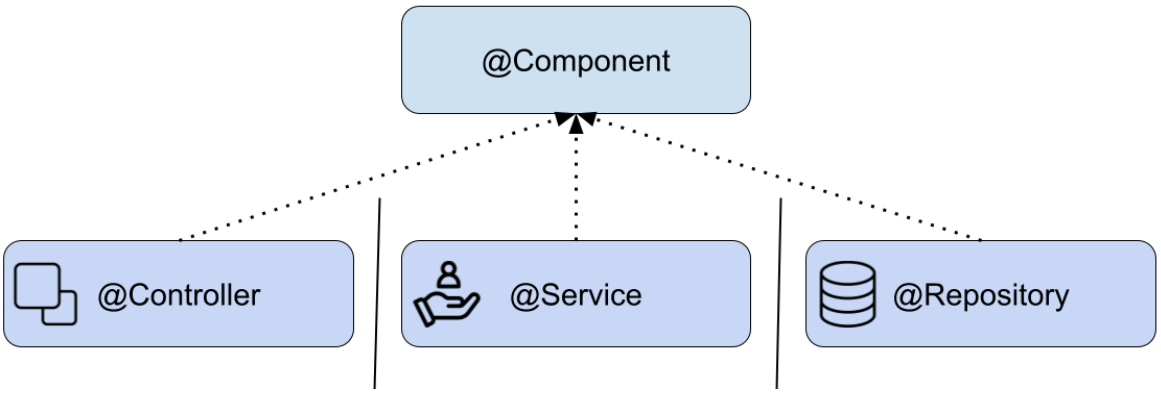
SpringBoot nace con la intención de simplificar los pasos 1 y 3 y que nos podamos centrar en el desarrollo de nuestra aplicación.

1.2. Anotaciones en Spring

Las **anotaciones en Spring** son una forma de añadir metadatos a las clases, métodos y variables, lo que permite que Spring realice una serie de acciones automáticamente en función de estos metadatos.

Entre las anotaciones, encontramos los **estereotipos en Spring**, que categorizan cada uno de los componentes asociando una responsabilidad concreta y declaran una instancia (o bean) en el contenedor de Spring (Core Container).

Existen únicamente 4 estereotipos:



- **@Component:** se utiliza para marcar una clase como un componente de Spring. Los componentes son clases que forman parte de la lógica de negocio de una aplicación y pueden ser inyectados en otros componentes mediante la inyección de dependencias.
- **@Controller:** se utiliza para marcar una clase como un controlador en una aplicación web basada en Spring MVC. Los controladores se encargan de recibir y gestionar las solicitudes HTTP de los usuarios y devolver una respuesta adecuada.
- **@Service:** se utiliza para marcar una clase como un servicio, que es una clase que proporciona funcionalidades lógicas y de negocio para otras clases de la aplicación.
- **@Repository:** se utiliza para marcar una clase como un repositorio, que es una clase que se encarga de acceder a datos almacenados en una base de datos o una fuente externa de datos.

Aparte de los estereotipos vistos, existen **otras anotaciones** en Spring:

- **@SpringBootApplication:** anotación que aparece en la clase principal (main) de todo proyecto que definamos con Spring Boot. Esta anotación hereda el comportamiento de las siguientes anotaciones:
    - **@EnableAutoConfiguration:** Esta es una anotación clásica de Spring que se encarga de forma inteligente de intentar configurar Spring de forma automática. Es la anotación encargada de buscar en el Classpath todas las clases con @Entity y registrarlas con el proveedor de persistencia que tengamos. Por lo tanto, con Spring Boot es suficiente configurar simplemente el dataSource a nivel application.properties, ya que Spring buscará todas las clases.
    - **@SpringConfiguration:** Es la anotación que define que el fichero es un fichero de Configuración de Spring. Normalmente, esto se solía hacer antiguamente con @Configuratión. La particularidad que tiene @SpringConfiguration es que solo puede haber una en la aplicación.
    - **@ComponentScan:** Se encarga de revisar los paquetes actuales y registrar de forma automática cualquier @Servie @Repository @Controller que la aplicación tenga de forma totalmente transparente para Spring Framework.
- **@RestController:** es una especialización de controller que contiene las anotaciones de @Controller y @ResponseBody. Este tipo de anotación se utiliza mucho en las aplicaciones de Spring Boot.
- **@Configuration:** se encargan de configurar los componentes de la aplicación. Se suelen iniciar al comienzo de nuestra aplicación.
- **@Bean:** nos sirve para indicar que este bean será administrado por Spring Boot (Spring Container).
- **@Autowired:** se utiliza para inyectar una dependencia en un atributo de una clase. Cuando se utiliza esta anotación, Spring busca una implementación de la interfaz o clase dependiente y la asigna al atributo correspondiente.
    - **@Qualified:** si utilizamos interfaces, nos ayuda a elegir la implementación deseada.

Estas son solo algunas de las anotaciones más comunes en Spring, pero hay muchas más que se pueden utilizar en función de las necesidades de la aplicación.

1.3. Contenedor de inversión de control

El corazón de Spring Framework es su contenedor de inversión de control (IoC). Se refiere a la habilidad de Spring para invertir el control de una aplicación a través de la inyección de dependencias. En otras palabras, en lugar de tener que crear y administrar manualmente las dependencias de una aplicación, Spring puede hacerlo por usted mediante la inyección de dependencias en las clases donde se necesiten. Los objetos creados y gestionados por el contenedor se denominan objetos gestionados o beans.

Por lo tanto, **la inyección de dependencias** en Spring es una técnica que se utiliza para inyectar objetos en una clase dependiente en lugar de crearlos manualmente dentro de la clase.

Esto puede ayudar a hacer que su código sea más limpio y mantenible, ya que reduce la cantidad de código que necesita escribir y mantiene las dependencias de su aplicación separadas de su lógica de negocio.

1.3.1. Ejemplo de funcionamiento

Supongamos que tenemos una clase que representa una base de datos, llamada **DBService**, y otra clase que representa una aplicación, llamada **MyApp**. La aplicación necesita acceder a la base de datos para recuperar y almacenar datos. En lugar de crear una instancia de DBService manualmente en MyApp, podemos utilizar la inyección de dependencias de Spring para que lo haga por nosotros.

Primero, necesitamos anotar la clase DBService con la anotación **@Component** para indicar a Spring que esta es una dependencia que puede inyectar en otras clases. Luego, en la clase MyApp, necesitamos anotar el campo que almacenará la instancia de DBService con la anotación **@Autowired** para indicar a Spring que debe inyectar una instancia de DBService en ese campo.

A continuación, se muestra un ejemplo de código que ilustra esto:

```
// Clase que representa una base de datos
@Component
public class DBService {
  ...
}
```

```
// Clase que representa una aplicación
public class MyApp {
  // Campo donde se almacenará la instancia de DBService
  // Utilizamos la anotación @Autowired para indicar que se debe inyectar una
  // instancia de DBService
  @Autowired
  private DBService dbService;
  ...
}
```

En este ejemplo, cuando Spring inicia la aplicación, detectará la anotación @Autowired en el campo dbService y buscará una instancia de DBService para inyectar en ese campo. Debido a que la clase DBService está anotada con @Component, Spring creará una instancia en su contenedor de tipo Singleton y sabrá que esta es una dependencia que puede inyectar en otras clases, por lo que inyectará una instancia de DBService en el campo dbService de MyApp.

1.4. Estructuras de paquetes en un proyecto Spring Boot

No hay una forma obligatoria que Spring nos imponga para el nombrado y la estructura de nuestros paquetes. Sin embargo, sí existen recomendaciones, usos comunes y buenas prácticas.

Una buena recomendación y buena práctica es dividir nuestro proyecto en capas.

Aprovechando esto podemos crear los paquetes para los 3 estereotipos descendientes de @Component y un paquete para las entidades de datos:

- **Paquete controller:** para todos los endpoints que tenga nuestra aplicación.
- **Paquete service:** para agregar allí las clases que respondan a la funcionalidad y lógica.
- **Paquete repository:** para las clases que establecen la comunicación con la base de datos.
- **Paquete model:** para las representaciones de nuestro modelo de datos (entidades).

Más información:

- https://gustavopeiretti.com/estructura-de-paquetes-spring-boot/

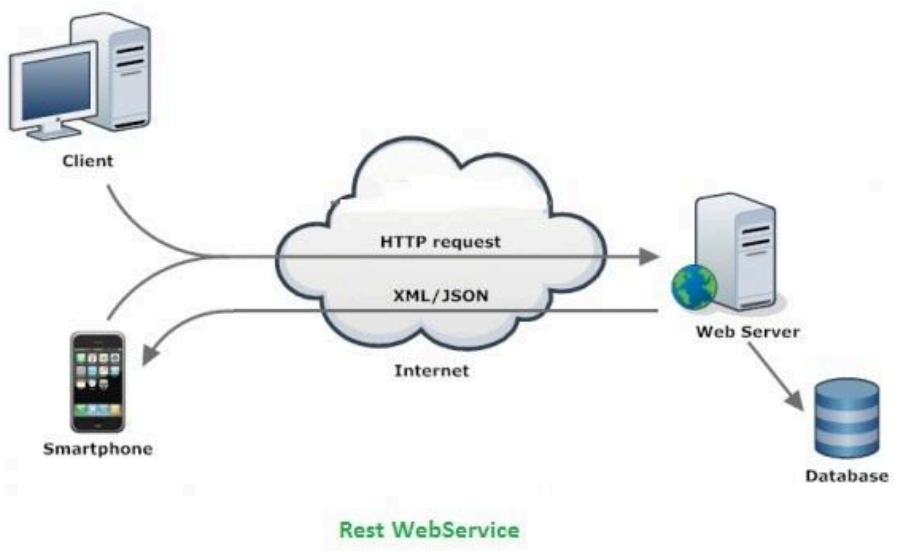
2. ¿QUÉ SON LOS SERVICIOS WEB REST?

Un **servicio web** es una aplicación que se encuentra en el lado servidor y permite que otra aplicación cliente conecte con ella a través de Internet para el intercambio de información utilizando el protocolo HTTP.

Una de las principales características de los servicios web es que no es necesario que ambas aplicaciones (servidor y cliente) estén escritas en el mismo lenguaje, lo que hace que la interoperabilidad sea máxima. Por ejemplo, podríamos crear un servicio web en Python y utilizarlo conectándonos desde una aplicación móvil con Android, desde otra aplicación programada en Java o incluso desde otro servicio web escrito con .NET.

Además, utilizan el protocolo HTTP para el intercambio de información, lo que significa que la conexión se establece por el puerto 80, que es el mismo que utilizan los navegadores y que es prácticamente seguro que se encuentre abierto en cualquier organización protegida por firewall. Esto hace que no sea necesario tener especial cuidado abriendo puertos innecesarios para poder conectarnos a ellos. Antes de la llegada de los servicios web, como los conocemos ahora, existían otros protocolos más complicados que requerían de servicios y puertos adicionales, incrementando el riesgo de ataques en las organizaciones que los ponían en marcha.

En la siguiente imagen podemos ver el funcionamiento de un servicio web.



Rest WebService

2.1. REST Web Service

Los **Servicios Web REST** son Servicios Web que cumplen una serie de requisitos según un patrón de arquitectura definida hacia el año 2000 y que se ha extendido siendo el patrón predominante a la hora de implementar este tipo de aplicaciones.

Básicamente, consiste en seguir una serie de reglas que definen dicha arquitectura. Entre ellas están el uso del protocolo HTTP por ser el más extendido a lo largo de Internet en la actualidad. Además, cada recurso del servicio web tiene que ser identificado por una dirección web (una URL) siguiendo una estructura determinada. Además, la respuesta tendrá que tener una estructura determinada en forma de texto que normalmente vendrá en alguno de los formatos abiertos más conocidos como XML o JSON.

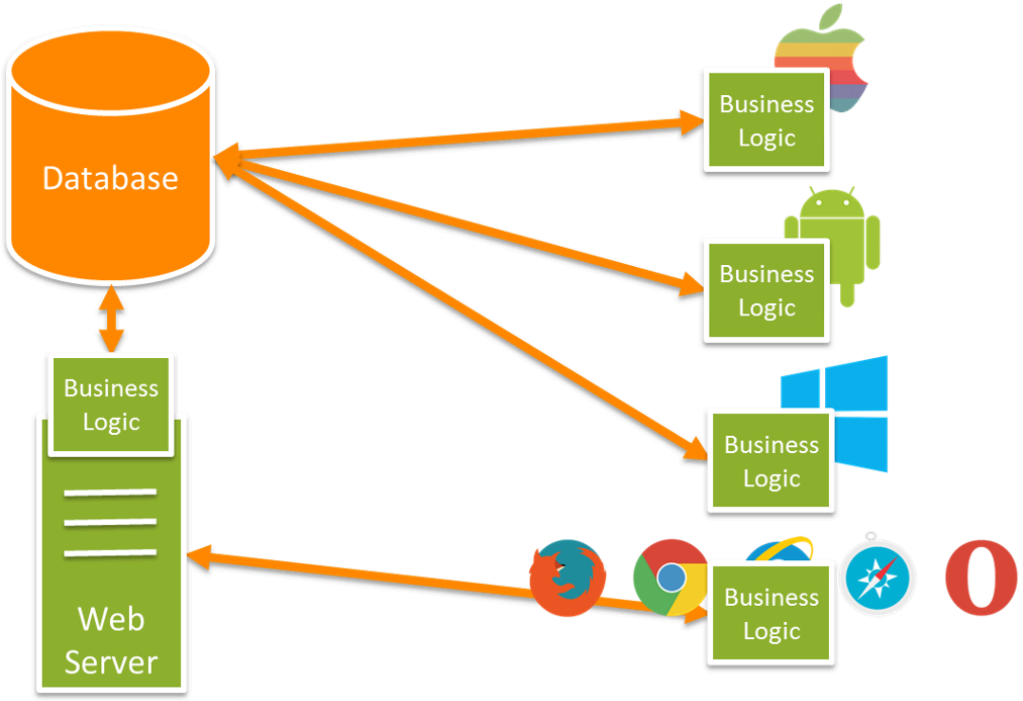
Task	Method	Path
Create a new task	POST	/tasks
Delete an existing task	DELETE	/tasks/{id}
Get a specific task	GET	/tasks/{id}
Search for tasks	GET	/tasks
Update an existing task	PUT	/tasks/{id}

Esas URLs y su estructura son lo que definen lo que se conoce como la API del Servicio Web, que son las diferentes operaciones a las que los clientes tienen acceso para comunicarse con el mismo. En este caso se trata de una API Web.

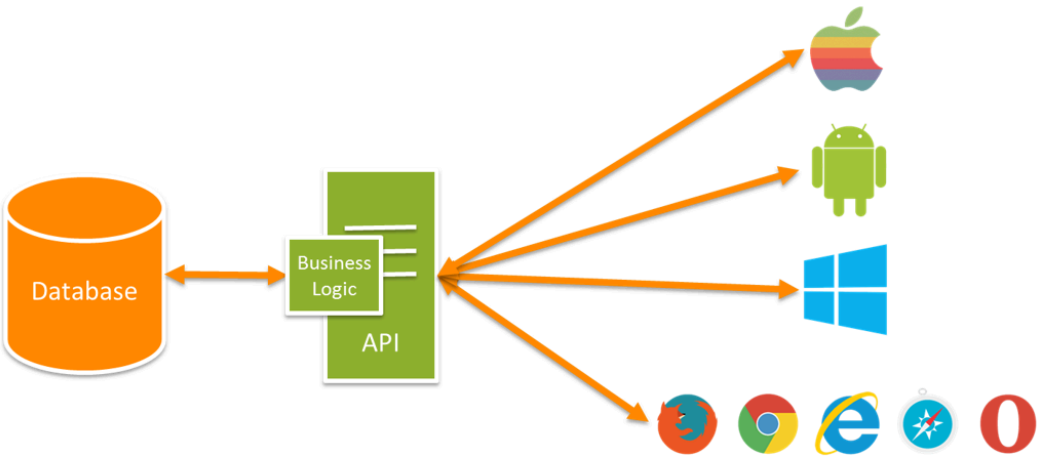
2.2. Web API

Una **Web API** es una API (Application Programming Interface) implementada para un Servicio Web de forma que este puede ser accesible mediante el protocolo HTTP, en principio por cualquier cliente web (navegador) aunque existen librerías que permiten que cualquier tipo de aplicación (escritorio, web, móvil, otros servicios web…) accedan a la misma para comunicarse con dicho servicio web.

La Web API es una de las partes de los Servicios Web que, tal y como comentamos anteriormente, mejoran sustancialmente la interoperabilidad de estos con los potenciales clientes, ya que permiten que solamente haya que implementar un único punto de entrada para comunicarse con el servicio web independientemente del tipo de aplicación que lo haga. De esa manera, el desarrollador del Servicio Web define la lógica de negocio en el lado servidor y los diferentes clientes que quieran comunicarse con el mismo lo hacen a través de la Web API, realizando solicitudes a las diferentes URLs que definen las operaciones disponibles.



Arquitectura de aplicación web con API:



Cuando se consume una web API, se utiliza el protocolo HTTP para la comunicación entre el cliente y el servidor. Como hemos visto anteriormente, el cliente realiza peticiones HTTP (como GET, POST, PUT, DELETE) al servidor y el servidor responde con un código de estado HTTP y, a veces, con datos en formato JSON o XML.

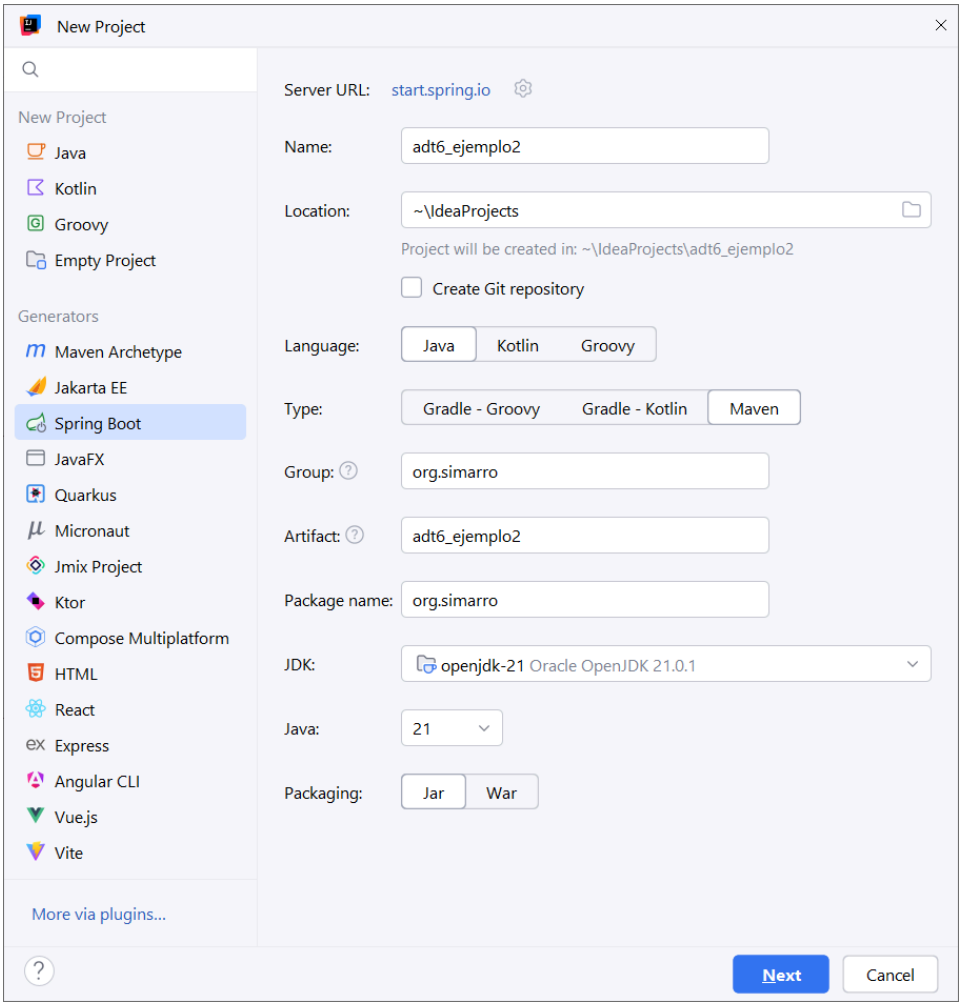
Ejemplo JSON:

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    },
    {
      "type": "mobile",
      "number": "123 456-7890"
    }
  ],
  "children": []
}
```

Para entender mejor esta introducción, revisar el **Anexo1: Diseño de APIs: Protocolo HTTP** y el video de ejemplo con Spring Boot **adt6_ejemplo1**.

3. DESARROLLO DE SERVICIOS WEB CON SPRING BOOT

Lo primero que haremos será crear un proyecto nuevo utilizando el IDE IntelliJ IDEA.



Tener en cuenta que he modificado el “Package name” por uno más corto.

También podemos utilizar la web Spring Initializr para preparar el proyecto inicial sobre el que luego diseñaremos nuestra aplicación.

La siguiente pantalla es para escoger la versión de Spring Boot (4.0.0) y sus dependencias:

- **Developer Tools:**
  - **Spring Boot DevTools:** proporciona reinicios rápidos de aplicaciones y configuraciones para mejorar la experiencia de desarrollo.
  - **Lombok:** anotaciones en Java que nos ayudan a reducir el código repetitivo.

- **Web:**
  - **Spring Web:** crear aplicaciones web, incluidas las RESTFul, utilizando Spring MVC. Utiliza Apache Tomcat como contenedor incrustado predeterminado.
- **SQL:**
  - **Spring Data JPA:** persistencia de datos con la API de persistencia usando Spring Data e Hibernate.
  - **MySQL Driver:** driver que permite que los programas se conecten con MySQL.
  - **PostgreSQL Driver:** driver que permite que los programas Java se conecten a una base de datos PostgreSQL utilizando código Java estándar e independiente. O trabajamos en MySQL o en PostgreSQL, por lo tanto, escogeremos uno de los dos.

Una vez tengamos creado el proyecto inicial, podemos empezar a trabajar en él. En este caso se trata de crear una aplicación web para una tienda de comercio electrónico, en concreto implementaremos un Controlador REST con todas sus funcionalidades. Por lo tanto, implementaremos servicios web, por lo que definiremos una clase que hará de @RestController, la comunicación se hará utilizando JSON.

3.1. Archivo de configuración

El archivo de configuración del proyecto se encuentra en la ruta /src/main/resources con el nombre de **application.properties**.

En el siguiente enlace, podemos conocer todas las propiedades que podemos modificar:

- https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html

Como ejemplo (no hace falta hacerlo), podemos modificar el puerto y banner de nuestra aplicación:

```
server.port=9595

# Dejar los ficheros logo.txt y logo.jpg en el /src/main/resources
spring.banner.location=logo.txt
spring.banner.image.location=logo.jpg
```

3.1.1. Configuración del SGBD MySQL

En el siguiente ejemplo modificamos el **application.properties** para configurar nuestra aplicación para conectar con una base de datos **MySQL**:

```
# Configuración para el acceso a la Base de Datos
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=update

# Datos de conexión con la base de datos MySQL
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost/adt6_ejemplo2
spring.datasource.username=root
spring.datasource.password=
```

Hay que tener en cuenta que la propiedad **spring.jpa.hibernate.ddl-auto** se utiliza para que la base de datos se genere automáticamente en cada arranque de la aplicación. Esto nos interesará cuando estemos en desarrollo, pero no cuando queramos desplegarla en producción. Por lo que tendremos que tener cuidado y controlar el valor de dicha propiedad.

- **none:** para indicar que no cree la base de datos.
- **create:** si queremos que la cree cada vez que inicie la aplicación, destruyendo datos previos.
- **create-drop:** si queremos que la cree y la destruya al terminar la sesión.
- **update:** actualiza la base de datos basándose en las entidades mapeadas.
- **validate:** válida el esquema, pero no realiza cambios en la base de datos.

3.1.2. Configuración del SGBD PostgreSQL

```
# Configuración para el acceso a la Base de Datos
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=update

# Datos de conexión con la base de datos PostgreSQL
spring.datasource.driver-class-name=org.postgresql.Driver
spring.datasource.url=jdbc:postgresql://localhost/adt6_ejemplo2
spring.datasource.username=postgres
spring.datasource.password=1234
```

3.2. Definir la base de datos

Hay que tener en cuenta que Spring utiliza por debajo el framework de **Hibernate** para trabajar con la Base de Datos. Eso nos va a permitir trabajar con nuestras clases Java directamente sobre la Base de Datos, ya que será Hibernate quien realizará el mapeo entre el objeto Java (y sus atributos) y la tabla de MySQL (y sus columnas) a la hora de realizar consultas, inserciones, modificaciones o borrados. E incluso a la hora de crear las tablas, puesto que bastará con definir nuestro modelo de clases con las anotaciones apropiadas para que Spring pueda crearlas basándonos en estas (y porque tenemos la opción **spring.jpa.hibernate.ddl-auto=update** en el fichero de configuración). Cuando ya no queramos que Spring genere automáticamente la base de datos en cada arranque (por ejemplo, en producción), tendremos que cambiar esa opción a valor **none**.

Aunque tenemos la configuración para dos SGBD como MySQL y PostreSQL, optamos por MySQL para su implementación, simplemente tendremos que crear la base de datos. Y ya de paso aprovecharemos para crear un usuario con el que la aplicación web se conectará (de esa manera evitamos tener que configurar el acceso usando el usuario root).

```
CREATE DATABASE adt6_ejemplo2;
CREATE USER pepe@locahost IDENTIFIED BY '123456';
GRANT ALL PRIVILEGES ON adt6_ejemplo2.* TO pepe;
```

Por lo tanto, podríamos utilizar pepe como usuario y 123456 como contraseña en el fichero de configuración (application.properties) del proyecto.

En nuestros ejemplos, durante el curso, utilizaremos el usuario root sin password. Esta práctica no es la más adecuada y deberíamos crearnos un usuario con su contraseña para cada ejercicio como se ha comentado anteriormente, pero por temas de agilizar nuestro desarrollo no se utilizarán usuarios específicos.

3.3. Implementación del modelo de datos: Model

Como **Spring Boot utiliza Hibernate como librería ORM** (Object-Relationship Mapping), para definir el modelo de datos de nuestra base de datos, nos bastará con escribir las clases Java que representarán a los datos en nuestra aplicación web. A través de las anotaciones que veremos a continuación, le daremos las instrucciones a Spring acerca de cómo crear la base de datos de forma transparente para nosotros.

Así, simplemente tenemos que crear en el paquete **org.simarro.model**, la clase con los atributos y métodos que queramos y añadir las anotaciones que orientarán a Hibernate para saber a qué tabla corresponden los objetos de la clase y a qué columnas sus atributos.

Además, la librería **lombok** para la generación automática de getters, setters y constructores.

```
import lombok.*;
import jakarta.persistence.*;
import java.time.LocalDate;

@Data
@AllArgsConstructor
@NoArgsConstructor
@Entity
public class Producto {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Column(nullable = false, length = 80)
    private String nombre;

    @Column(nullable = true, length = 150)
    private String descripcion;

    @Column(nullable = false, length = 50)
    private String categoria;

    @Column(nullable = false)
    private float precio;

    @Column(name = "fecha_creacion")
    @JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "dd/MM/yyyy")
    private LocalDate fechaCreacion;
}
```

Recordad que todas las anotaciones Java en el ejemplo anterior son clases que pertenecen al paquete 'jakarta.persistence'. Tened cuidado de no importar las mismas clases que existen en otros paquetes, aunque estén relacionados con Spring.

3.4. Implementación del acceso a la base de datos: Repository

Ahora creamos la interfaz donde se definirán los métodos que permitirán acceder a la Base de Datos en el paquete **org.simarro.repository**. En este caso nos basta con definir las cabeceras de los mismos, puesto que se trata de una interfaz. Será el framework el que se encargue de su implementación.

```
import java.util.List;

import org.simarro.model.Producto;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface IProductoRepository extends JpaRepository<Producto, Integer> {

    // Métodos específicos de esta entidad diferentes al CRUD
}
```

Cuando creamos la nueva interfaz, en este caso (**IProductRepository.java**) podemos heredar cualquiera de las dos interfaces diferentes, de la interfaz **CrudRepository** o de **JpaRepository**. Cuando heredamos le indicamos el tipo de entidad y el tipo de clave primaria **<Product, Integer>**. Por heredar de estas interfaces, podremos contar con que tenemos las operaciones que nos permiten registrar/modificar (save), leer (select) y eliminar (delete) información de la BD.

Nosotros vamos a utilizar la interfaz JpaRepository porque tiene más métodos implementados. Aunque para el ejemplo que vamos a realizar nos daría igual heredar de CrudRepository.

Por otro lado, el IDE nos indica que no hace falta estereotipar la clase con @Repository por haber heredado de la interfaz anterior. Pero para clarificar los conceptos del tema, no lo vamos a eliminar.

3.5. Implementación de la lógica de negocio: Services

Los **services** serán la capa de nuestra aplicación web donde implementaremos toda la lógica de negocio. Definiremos una interface **IProductoService.java** con todos los métodos que necesitemos dentro del paquete **org.simarro.service**:

```
import java.util.List;
import org.simarro.model.Producto;

public interface IProductoService {

    Producto registrar(Producto producto);
    Producto modificar(Producto producto);
    List<Producto> listar();
    Producto listarPorId(Integer id);
    void eliminar(Integer id);

}
```

Que implementaremos en la clase **ProductoServiceImpl**:

```
import java.util.List;
import java.util.Optional;

import org.simarro.model.Producto;
import org.simarro.repository.IProductoRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class ProductoServiceImpl implements IProductoService {

	@Autowired
	private IProductoRepository repo;

	@Override
	public Producto registrar(Producto producto) {
		return repo.save(producto);
	}

	@Override
	public Producto modificar(Producto producto) {
		return repo.save(producto);
	}
```

```
	@Override
	public List<Producto> listar() {
		return repo.findAll();
	}

	@Override
	public Producto listarPorId(Integer id) {
		Optional<Producto> op = repo.findById(id);
		return op.isPresent() ? op.get() : null;
	}

	@Override
	public void eliminar(Integer id) {
		repo.deleteById(id);
	}
}
```

Clase Optional

La clase Optional fue introducida en Java 8 con el objetivo de reducir los errores relacionados con valores nulos (null). Este concepto implementa el patrón Option, que permite manejar situaciones en las que un método puede no devolver un valor esperado. De esta manera, se obliga a gestionar explícitamente la posible ausencia de valor, ya sea proporcionando un valor alternativo o ejecutando una acción específica.

Optional actúa como un contenedor que puede o no contener un valor no nulo. Si el valor está presente, el método isPresent() devuelve true, y get() permite acceder a dicho valor. En caso de que el valor no esté presente, se puede utilizar el método orElse() para retornar un valor por defecto.

3.6. Implementación de la capa final: Controller

Antes de continuar, es muy conveniente leer el siguiente artículo sobre los diferentes métodos HTTP en servicios web REST. Explica muy claramente cómo deben ser las diferentes operaciones que se pueden llevar a cabo sobre los recursos del sistema.

A continuación, definiremos un controlador REST. En él se han definido diferentes endpoints que permite ejecutar las siguientes operaciones:

- **Obtener todos los productos:** método GET que devuelve toda la colección.
- **Obtener un producto determinado:** método GET que devuelve un objeto determinado utilizando un parámetro Path.
- **Registrar un nuevo producto:** método POST que registra un nuevo producto en la BD.
- **Modificar un producto:** método PUT que modifica un producto.
- **Eliminar un producto:** método DELETE que elimina un producto existente.

Para entender el siguiente fragmento de código conviene tener en cuenta lo siguiente:

- Cada método anotado define un **endpoint** que podrá ser invocado por otra aplicación.
- Las anotaciones **@GetMapping**, **@PostMapping**, **@PutMapping**, **@DeleteMapping** definen el método (GET, POST, PUT, DELETE) y la URL de dicho endpoint.
- Si el endpoint debe utilizar Path Params vendrán definidos en la URL y también en el método como **@PathVariable**.
- Si el endpoint debe utilizar Body Params vendrán definidos solamente en el método como **@RequestBody**.
- Si el endpoint debe utilizar Query Params vendrán definidos solamente en el método como **@RequestParam**.

```
import java.util.List;

import org.simarro.model.Producto;
import org.simarro.service.IProductoService;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
```

```java
@RestController
@RequestMapping("/productos")
public class ProductoController {

	@Autowired
	private IProductoService service;

	@GetMapping
	public List<Producto> listar(){
		return service.listar();
	}

	@GetMapping("/{id}")
	public Producto listarPorId(@PathVariable("id") Integer id){
		return service.listarPorId(id);
	}

	@PostMapping
	public Producto registrar(@RequestBody Producto producto) {
		return service.registrar(producto);
	}

	@PutMapping
	public Producto modificar(@RequestBody Producto producto) {
		return service.modificar(producto);
	}

	@DeleteMapping("/{id}")
	public void eliminar(@PathVariable("id") Integer id){
		service.eliminar(id);
	}
}
```

Podemos probar todos los métodos por una aplicación llamada **Postman** explicada en el apartado 8.

📖 **Aplicación recomendada:**

Postman es una herramienta popular para realizar pruebas y depuración de servicios web y APIs (Application Programming Interfaces). Es ampliamente utilizada por desarrolladores para enviar, recibir y analizar solicitudes HTTP, lo que facilita la interacción con APIs de forma sencilla, eficiente y visual.

3.7. Personalizando la obtención de datos mediante JPQL

Hasta ahora, hemos visto cómo realizar nuestra API Rest con las operaciones básicas. Ahora vamos a personalizar las operaciones de lectura mediante JPQL (Java Persistence Query Language) o HQL.

Para ello vamos a modificar el proyecto anterior para añadir las siguientes operaciones:

- **Obtener todos los productos de una categoría determinada:** método GET que devuelve la colección filtrada por el campo categoría.
- **Obtener todos los productos por la fecha de creación:** método GET que devuelve la colección filtrada por el campo fechaCreación.

IProductoRepository.java

```
@Repository
public interface IProductoRepository extends JpaRepository<Producto, Integer> {

	// Métodos específicos de esta entidad diferentes al CRUD mediante JPQL
	@Query("FROM Producto p WHERE LOWER(p.categoria) LIKE %:categoria%")
	List<Producto> listarPorCategoria(@Param("categoria") String categoria);

	// >= 03/02/2024 <04/02/2024
	@Query("FROM Producto p WHERE p.fechaCreacion BETWEEN :fecha1 AND :fecha2")
	List<Producto> listarPorFecha(@Param("fecha1") LocalDate fecha1,
	                              @Param("fecha2") LocalDate fecha2);
}
```

IProductoService.java

```
public interface IProductoService {
	...

	// Métodos específicos de esta entidad
	List<Producto> listarPorCategoria(String categoria);
	List<Producto> listarPorFecha(LocalDate fecha1, LocalDate fecha2);
}
```

ProductoServiceImpl.java

```
@Service
public class ProductoServiceImpl implements IProductoService {

	...
	// Métodos específicos de esta entidad

	@Override
	public List<Producto> listarPorCategoria(String categoria) {
		return repo.listarPorCategoria(categoria);
	}

	@Override
	public List<Producto> listarPorFecha(LocalDate fecha1, LocalDate fecha2){
		return repo.listarPorFecha(fecha1, fecha2);
	}

}
```

ProductoController.java

```
@RestController
@RequestMapping("/productos")
public class ProductoController {

	// Métodos específicos de esta entidad

	@GetMapping("/buscar")
	public List<Producto> listarPorCategoria(
				@RequestParam(value = "categoria") String categoria){
		return service.listarPorCategoria(categoria);
	}


	@GetMapping("/fecha")
	public List<Producto> listarPorFecha(
			@RequestParam(value = "fecha1") String fecha1,
			@RequestParam(value = "fecha2") String fecha2){
		return service.listarPorFecha(
			LocalDate.parse(fecha1, DateTimeFormatter.ofPattern("dd/MM/yyyy")),
			LocalDate.parse(fecha2, DateTimeFormatter.ofPattern("dd/MM/yyyy")));
	}

}
```

Algunas consideraciones para trabajar con fechas en Java:

- Es recomendable utilizar el tipo de dato **LocalDate** en lugar de utilizar java.util.Date.
- Para trabajar con fechas en una API REST en Spring Boot, puedes utilizar la anotación **@JsonFormat** en tus entidades para especificar el formato de fecha que deseas utilizar en los JSON de salida.
- Para pasar una fecha en una petición HTTP utilizando Postman, deberías enviarla en el formato estándar **ISO-8601**. Esto significa que la fecha debe tener el siguiente formato: "yyyy-MM-dd'T'HH:mm:ss.SSSXXX", donde XXX es el desplazamiento horario.

  Por ejemplo, si quieres enviar la fecha 2023-12-10T10:15:30.00+01:00, deberías enviarla como un parámetro en la petición con ese valor exacto.

  Dependiendo del tipo de petición y como esté configurado el endpoint, puedes enviar el parámetro en el body, en la URL o en el header. En el caso de una petición POST o PUT, es común enviar los parámetros en el body, en formato JSON.
- En nuestro ejemplo, mandaremos la fecha en formato **"dd/MM/yyyy"**. Para convertir una cadena de texto en el **formato "dd/MM/yyyy" a un objeto LocalDate** en Java, puedes utilizar el método parse() de la clase LocalDate junto con un formateador de fecha.

```
String dateString = "01/01/2024";
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/yyyy");
LocalDate localDate = LocalDate.parse(dateString, formatter);
```

  - La primera línea establece el string de fecha que deseas convertir.
  - La segunda línea establece un formateador de fecha con el patrón "dd/MM/yyyy".
  - La tercera línea utiliza el método parse() de LocalDate para convertir la cadena de texto en un objeto LocalDate utilizando el formateador especificado.

  Es importante mencionar que, si el formato de la fecha no es el esperado, el método parse() lanzará una excepción DateTimeParseException. También es recomendable validar la fecha antes de convertirla.

3.7.1. Automatización de consultas con Spring Data JPA

Spring Data JPA automatiza la creación de consultas SQL basándose en los nombres de los métodos de tus repositorios.

Algunas de las operaciones que realiza automáticamente son (entidad Persona):

1. **Búsquedas por Campo:** por ejemplo, findByNombre(String nombre) generará una consulta para buscar entidades por el campo nombre.
2. **Condiciones And/Or:** métodos como findByNombreAndApellido(String nombre, String apellido) o findByNombreOrApellido(String nombre, String apellido).
3. **Operaciones de Comparación:** findByEdadGreaterThan(int edad) para buscar entidades con una edad mayor a un valor específico.
4. **Ordenación:** métodos como findByNombreOrderByApellidoAsc(String nombre).
5. **Limitar Resultados:** findFirstByOrderByNombreAsc() o findTop3ByOrderByEdadDesc().

Spring Data JPA proporciona una amplia gama de palabras clave para la derivación automática de consultas en los métodos de los repositorios. Algunas de estas incluyen:

- **Operaciones básicas:** find...By, read...By, get...By, query...By, search...By.
- **Proyecciones:** exists...By, count...By.
- **Operaciones de borrado:** delete...By, remove...By.
- **Limitar resultados:** ...First<number>..., ...Top<number>....
- **Distinción de resultados:** ...Distinct....
- **Operadores lógicos:** And, Or.
- **Comparaciones:** GreaterThan, LessThan, Between, Like, Is, In.
- **Verificaciones nulas y de vacío:** IsNull, IsNotNull, IsEmpty, IsNotEmpty.
- **Ordenación:** OrderBy....

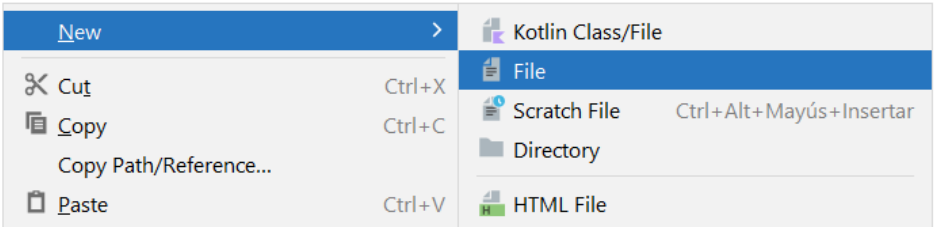
Estas palabras clave se pueden combinar para formar métodos de consulta sofisticados que cumplen con una variedad de criterios de búsqueda en tus entidades JPA. Para más información:

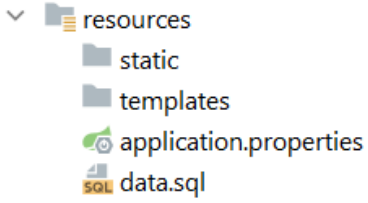
- https://docs.spring.io/spring-data/jpa/reference/repositories/query-keywords-reference.html

4. AÑADIR DATOS DE PRUEBA

Vamos a crear un archivo de texto (o lo traemos copiado desde fuera del proyecto) donde tengamos unos inserts de datos de prueba. Lo importante es que el archivo se llame **data.sql** exactamente y que esté colocado dentro de **resources**.

Sobre la carpeta resources hacemos clic en el botón derecho New → File:





Añadir en el archivo de propiedades **application.properties**:

```
spring.sql.init.mode=always
```

Ejemplo de sentencias INSERT dentro de **data.sql**, para nuestro modelo:

```
INSERT INTO producto (nombre, descripcion, categoria, precio, fecha_creacion)
  VALUES('Monitor', 'DELL S Series 27 pulgadas', 'Tecnología', 216.10, '2022/12/18');
INSERT INTO producto (nombre, descripcion, categoria, precio, fecha_creacion)
  VALUES('Teclado', 'Mecánico Gaming', 'Tecnología', 52.99, '2023/01/10');
INSERT INTO producto (nombre, descripcion, categoria, precio, fecha_creacion)
  VALUES('Televisor', 'LG OLED 55 pulgadas', 'Tecnología', 1018.99, '2023/01/20');
INSERT INTO producto (nombre, descripcion, categoria, precio, fecha_creacion)
  VALUES('Bicicleta', 'Scott spark', 'Ciclismo', 2599, '2023/01/21');
INSERT INTO producto (nombre, descripcion, categoria, precio, fecha_creacion)
  VALUES('Maillot', 'Spiuk T. M', 'Ciclismo', 63.95, '2023/01/21');
INSERT INTO producto (nombre, descripcion, categoria, precio, fecha_creacion)
  VALUES('Portátil', 'Asus TUF Gaming', 'Tecnología', 1599, '2023/01/28');
```

Para que todo funcione primero tiene que estar la base de datos creada con sus tablas. Por lo tanto, ejecutaremos nuestra aplicación de Spring Boot sin tener la línea agregada en el archivo de propiedades para que la primera vez que se ejecute nos cree las tablas necesarias sin datos.

Paramos el proyecto, y agregamos la línea en el archivo de propiedades para que cuando vuelva a ejecutar el programa nos haga los INSERTS correspondientes al fichero data.sql.

Podemos encontrar más configuraciones en los siguientes enlaces:

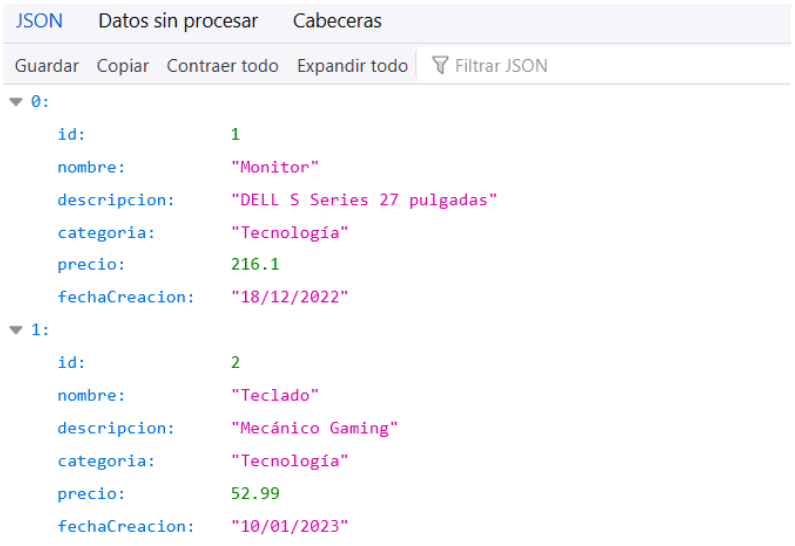
- https://www.baeldung.com/spring-boot-data-sql-and-schema-sql
- https://docs.spring.io/spring-boot/docs/2.1.x/reference/html/howto-database-initialization.html

Una vez realizado estos pasos, ya podemos probar ese servicio web de nuestra API, para ello abrimos el Chrome, Firefox o herramientas como Postman.

- En Chrome:


```
[{"id":1,"nombre":"Monitor","descripcion":"DELL S Series 27
pulgadas","categoria":"Tecnología","precio":216.1,"fechaCreacion":"18/12/2022"},
{"id":2,"nombre":"Teclado","descripcion":"Mecánico
Gaming","categoria":"Tecnología","precio":52.99,"fechaCreacion":"10/01/2023"},
```

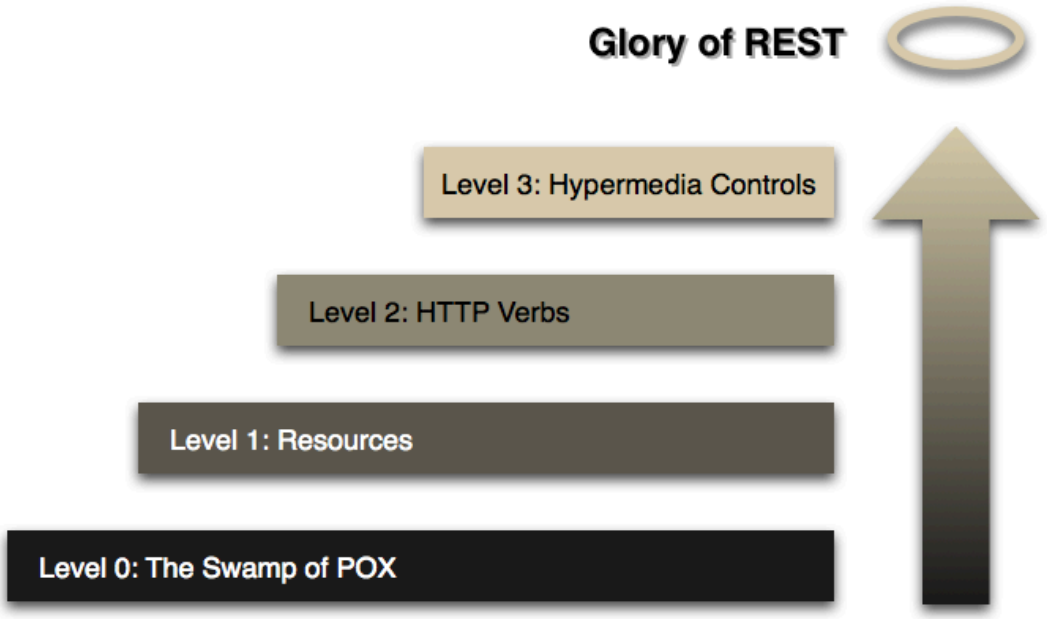

- En Firefox sale el JSON más organizado:



5. MODELO DE MADUREZ DE RICHARDSON

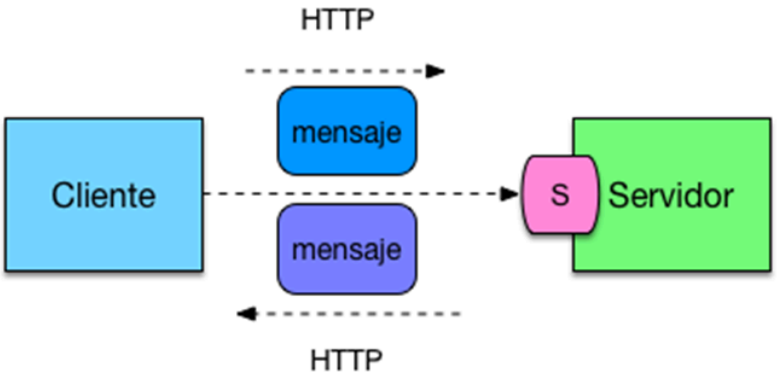
Hemos realizado una API Rest con un CRUD básico. El **modelo de madurez de Richardson** nos propone mejorar nuestro API Rest, podemos leer el siguiente post donde nos explica las mejoras que podemos introducir:

- https://martinfowler.com/articles/richardsonMaturityModel.html

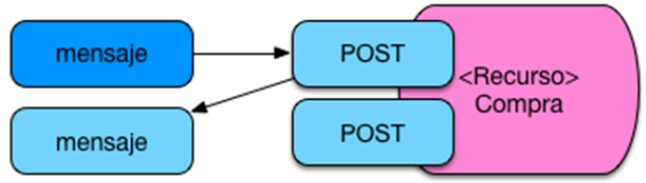


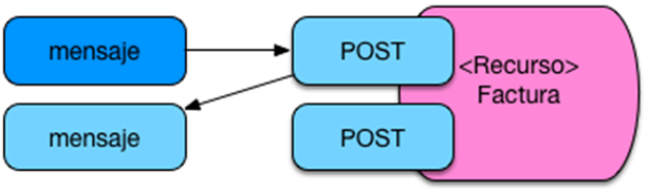
Vamos a explicar de forma rápida los 4 niveles que tenemos hasta llegar a la gloria de los servicios REST según el modelo de madurez de Richardson:

- **Nivel 0:** enviar y recibir información mediante HTTP sin importar ningún estándar.

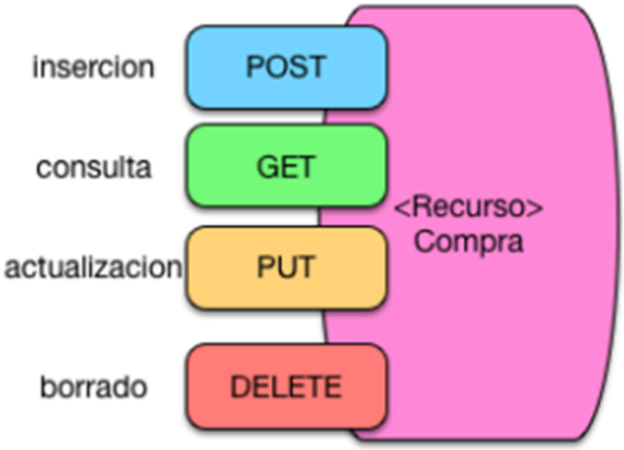


- **Nivel 1:** ídem que el anterior. Importa la nomenclatura de cómo se llaman los recursos. Normalmente los recursos se llaman en sustantivo y en plural (/pacientes). Aunque en las imágenes aparece el nombre en singular debería llamarse compras, facturas… tenerlo en cuenta.



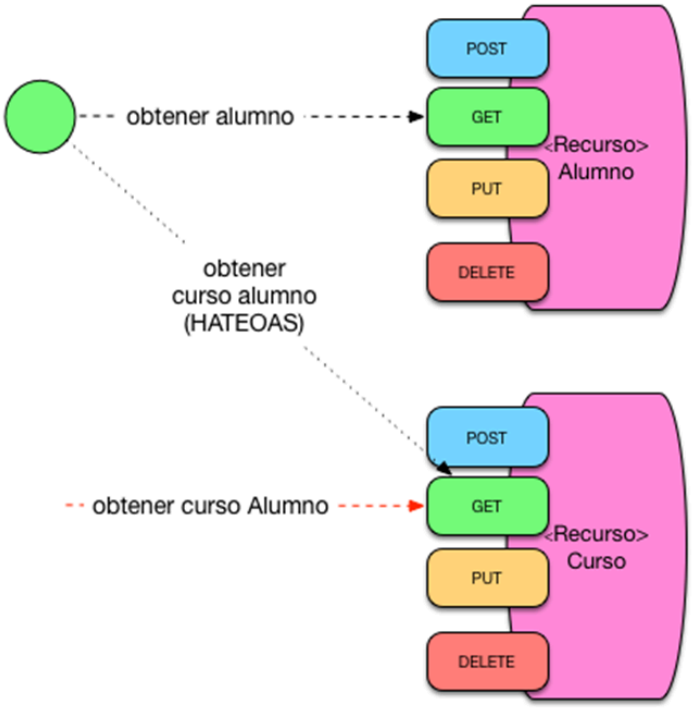


- **Nivel 2:** ídem que el anterior. Importa el uso de los verbos HTTP (Get para leer, POST para añadir, PUT para modificar, DELETE para borrar). También hace referencia a los códigos de respuesta que devuelven según la operación realizada.



- **Nivel 3:** ídem que el anterior. Sobre la respuesta del servicio, expresar información complementaria para poder consumir otros servicios. También es conocido como HATEOAS (Hypertext As The Engine Of Application State).

¿Para qué sirve este nivel? Supongamos que queremos acceder a un recurso Alumno. Si tenemos una Arquitectura a nivel 2 primero accederemos a ese recurso utilizando GET. En segundo lugar, deberemos acceder al recurso de Cursos para obtener su curso (línea roja).



Esto implica que el cliente que accede a los servicios REST asume un acoplamiento muy alto, debe conocer la url del Alumno y la del Curso. Sin embargo, si el recurso del Alumno contiene un link al recurso de Curso esto no hará falta conocerla. Podríamos tener una estructura JSON como la siguiente:

```
{
     id: 48,
     nombre: "Pedro",
     curso: "http://miservidor/alumnos/48/cursos/3"
}
```

Podremos acceder directamente al recurso de Curso utilizando las propiedades del Alumno esto es HATEOAS. De esta forma se aumenta la flexibilidad y se reduce el acoplamiento. Construir arquitecturas sobre estilo REST no es sencillo y hay que ir paso a paso.

- https://www.adictosaltrabajo.com/2013/12/02/spring-hateoas/

5.1. Mejorando nuestro Controller

En este apartado mejoraremos nuestro CRUD básico del ejemplo anterior y crearemos un nuevo proyecto con el nombre **adt6_ejemplo3** con las mejoras que nos dice el nivel 2 (haremos una copia exacta del código adt6_ejemplo2). No llegaremos al nivel 3 de HATEOAS porque normalmente no se trabaja con ese nivel de detalle a nivel empresarial.

Si recordamos lo que nos decía:

- **Nivel 2:** Importa el uso de los verbos HTTP (Get para leer, POST para añadir, PUT para modificar, DELETE para borrar). También hace referencia a los códigos de respuesta que devuelven según la operación realizada.

Por lo tanto, la respuesta será siempre un objeto **ResponseEntity** que contendrá la información de respuesta y un código de estado HTTP asociado (más información en el Anexo I).

```
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.util.List;

import org.simarro.model.Producto;
import org.simarro.service.IProductoService;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/productos")
public class ProductoController {

	@Autowired
	private IProductoService service;

	@GetMapping
	public ResponseEntity<List<Producto>> listar(){
		List<Producto> lista = service.listar();

		// Código 200 OK para select
		return new ResponseEntity<>(lista, HttpStatus.OK);
	}
```

```java
@GetMapping("/{id}")
public ResponseEntity<Producto> listarPorId(@PathVariable("id") Integer id){
	Producto obj = service.listarPorId(id);

	// Código 200 OK para select
	return new ResponseEntity<>(obj, HttpStatus.OK);
}

@PostMapping
public ResponseEntity<Producto> registrar(@RequestBody Producto producto) {
	Producto obj = service.registrar(producto);

	// Código 201 CREATED para insert
	return new ResponseEntity<>(obj, HttpStatus.CREATED);
}

@PutMapping
public ResponseEntity<Producto> modificar(@RequestBody Producto producto) {
	Producto obj = service.modificar(producto);

	// Código 200 OK para update
	return new ResponseEntity<>(obj, HttpStatus.OK);
}

@DeleteMapping("/{id}")
public ResponseEntity<Void> eliminar(@PathVariable("id") Integer id){
	service.eliminar(id);

	// Código 204 NOT CONTENT para delete
	return new ResponseEntity<>(HttpStatus.NO_CONTENT);
}



////////////////////////////////////////
// Métodos específicos de esta entidad //
////////////////////////////////////////

@GetMapping("/buscar")
public ResponseEntity<List<Producto>> listarPorCategoria(
		@RequestParam(value = "categoria") String categoria){

	List<Producto> lista = service.listarPorCategoria(categoria);

	// Código 200 OK para select
	return new ResponseEntity<>(lista, HttpStatus.OK);
}
```

```
    @GetMapping("/fecha")
    public ResponseEntity<List<Producto>> listarPorFecha(
                @RequestParam(value = "fecha1") String fecha1,
                @RequestParam(value = "fecha2") String fecha2){

            List<Producto> lista = service.listarPorFecha(
                LocalDate.parse(fecha1, DateTimeFormatter.ofPattern("dd/MM/yyyy")),
                LocalDate.parse(fecha2, DateTimeFormatter.ofPattern("dd/MM/yyyy")));

            // Código 200 OK para select
            return new ResponseEntity<>(lista, HttpStatus.OK);
    }

}
```

Mejorando los códigos de respuestas

Podemos mejorar los códigos de respuestas para una petición. Como ejemplo, modificaremos solamente la petición listarPorId().

```
@GetMapping("/{id}")
public ResponseEntity<Producto> listarPorId(@PathVariable("id") Integer id){

    Producto obj = service.listarPorId(id);

    if (obj == null) {
       // Código 404 No encontrado
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }

    // Código 200 OK para select
    return new ResponseEntity<>(obj, HttpStatus.OK);
}
```

Por lo tanto, si se encuentra el producto, se devuelve una respuesta HTTP con un estado 200 OK y el cuerpo de la respuesta contiene el producto. Si no se encuentra el producto, se devuelve una respuesta HTTP con un estado 404 Not Found.

6. VALIDACIONES

Para que las validaciones funcionen en una aplicación de Spring Boot, necesitas hacer lo siguiente:

1. Incluir la dependencia de validación en el archivo **pom.xml** de tu proyecto. Esto se hace añadiendo la siguiente línea al bloque <dependencies> a partir de Spring Boot 2.3:

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

2. Anotar el argumento del método controlador con **@Valid** para activar la validación. Lo anotamos en el controlador en las operaciones de registrar y modificar porque vamos a recibir datos de entrada. Por ejemplo:

```
@PostMapping
public void registrar(@Valid @RequestBody Producto producto) {
  // ...
}
```

3. Añadir **reglas de validación** a la clase Producto utilizando anotaciones de validación de Java Bean Validation, como @NotNull, @Size, etc. Por ejemplo:

```
public class Producto{
    @NotNull
    private String nombre;

    @Size(min = 3, max = 100, message = "Descripción debe tener entre 3 y 100.")
    private String descripcion;

}
```

   Para saber todas las reglas de validación que puedes utilizar consulta la siguiente página:

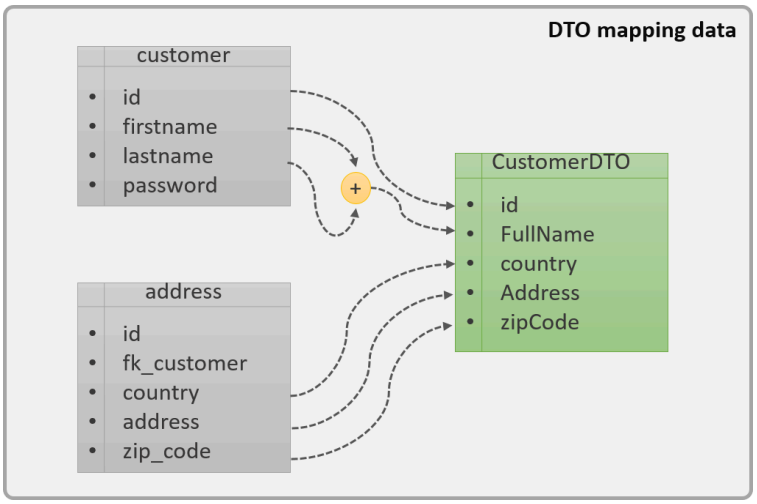
   - https://reflectoring.io/bean-validation-with-spring-boot/

Si has seguido estos pasos y las validaciones aún no funcionan, asegúrate de que no estés recibiendo una excepción durante la ejecución.

7. PATRÓN DTO

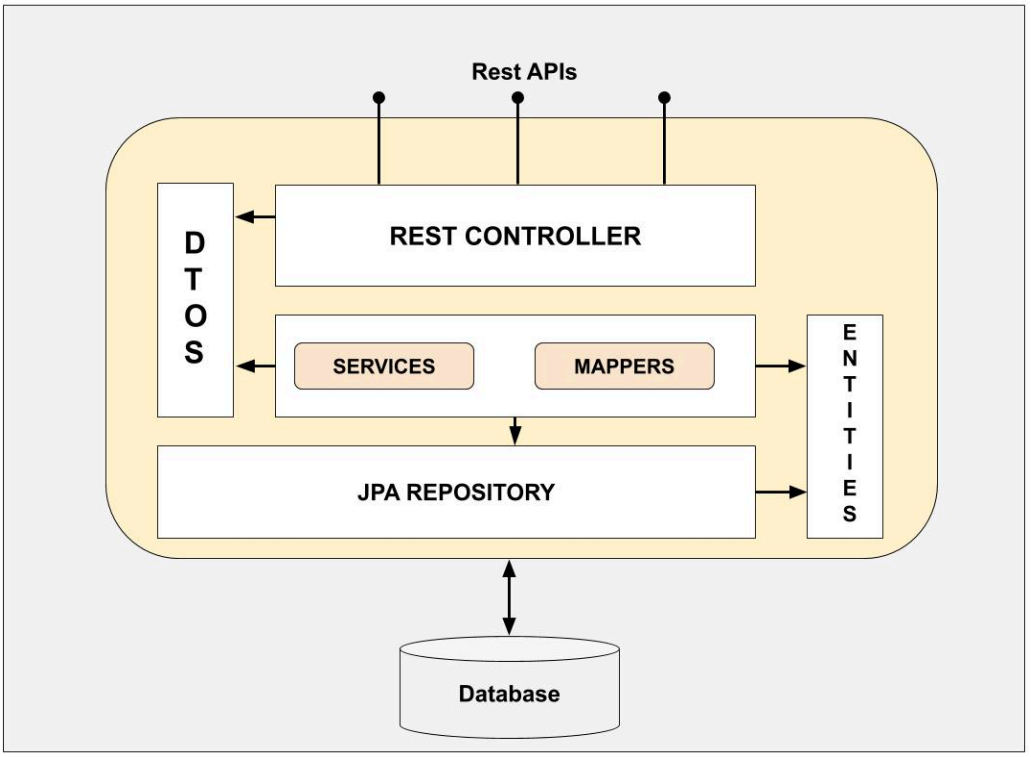
El uso del **patrón de DTO (Data Transfer Objects)** en Spring es una herramienta muy utilizada porque nos permite desacoplar la estructura de los parámetros de entrada del API de las Entidades. Una de las problemáticas más comunes cuando desarrollamos aplicaciones, es diseñar la forma en que la información debe viajar entre las diferentes capas o paquetes, ya que muchas veces por desconocimiento o pereza, utilizamos las clases de entidades para retornar los datos, lo que ocasiona que retornemos más datos de los necesarios.

El patrón DTO tiene como finalidad de crear un objeto plano (POJO) con una serie de atributos que puedan ser enviados o recuperados del servidor en una sola invocación, de tal forma que un DTO puede contener información de múltiples fuentes o tablas y empaquetarlas en una única clase.



En la imagen anterior podemos apreciar gráficamente cómo es que un DTO se conforma de una serie de atributos que puede o no, estar conformados por más de una fuente de datos. Para esto, el servidor obtiene la información de las tablas customer y address (izquierda) y realiza un "mapping" con el DTO. Adicional, la información puede ser pasada de un lado intacta como es el caso del id, country, address y zipCode o ser una derivada de más de un campo, como es el caso del fullName, el cual es la unión del firstname y lastname.

Otra de las ventajas no tan claras en la imagen, es que nos permite omitir información que el usuario no requiere, como es el caso de password. No es solo que no lo requiere, sino que además podría ser un gran fallo de seguridad, es por ello que en el DTO lo omitimos.



Por lo tanto, usaremos los **DTOs**, como objetos de transporte y con ellos facilitar la transición entre peticiones (*request*) y entidades, y entre entidades y respuestas (*responses*). Así podemos ensamblar y desensamblar los objetos de modelo y de transporte según nuestra necesidades.

Los **mapeadores** nos ayudarán en la misión de mapear los objetos de modelo a objetos de transporte y viceversa. Podemos usar una librería específica para ello como **ModelMapper**, **MapStruct** o podemos usar un mapeador propio.

Propósito de los DTO

1. **Seguridad y Encapsulación:** evitan la exposición directa de las entidades de la base de datos al cliente. Esto es crucial para la seguridad y la integridad de los datos.
2. **Optimización:** permiten enviar solo los datos necesarios en una solicitud o respuesta, lo que reduce la carga de datos y mejora el rendimiento.
3. **Abstracción y Desacoplamiento:** facilitan la separación entre la lógica de la base de datos y la lógica de la interfaz de usuario, lo que resulta en un diseño de software más limpio y mantenible.

7.1. Configuración del ModelMapper

Primero, debes agregar la dependencia de **ModelMapper** en tu archivo pom.xml:

```
<dependency>
    <groupId>org.modelmapper</groupId>
    <artifactId>modelmapper</artifactId>
    <version>3.0.0</version>
</dependency>
```

Luego, debes configurar un bean de ModelMapper en tu clase de configuración de Spring Boot, para ello creamos un nuevo paquete org.simarro.utils:

```
import org.modelmapper.ModelMapper;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class ModelMapperConfig {

    @Bean
    public ModelMapper modelMapper() {
        return new ModelMapper();
    }
}
```

7.2. Creación y utilización de DTO

Imaginemos una aplicación Spring Boot con una entidad Usuario y queremos exponer algunos de sus datos a través de una API REST.

```
@Entity
public class Usuario {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column
    private String nombre;
    @Column
    private String email;
    // Otros atributos, getters y setters
}
```

DTO Usuario

```
public class UsuarioDTO {
    private String nombre;
    private String email;
    // Constructor, Getters y setters
}
```

Creamos un paquete org.simarro.model.dto y creamos la clase UsuarioDTO, sólo incluimos los campos nombre y email, que son los que queremos exponer.

Controlador

Tenemos que tener en cuenta que nuestro controlador trabaja con la entidad UsuarioDTO.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/usuarios")
public class UsuarioController {

    @Autowired
    private IUsuarioService service;

    ...

    @GetMapping("/{id}")
    public ResponseEntity<UsuarioDTO> listarPorId(@PathVariable Integer id) {

        UsuarioDTO usuarioDTO = service.listarPorId(id);

        if(usuarioDTO == null) {
             // Código 204 NoData para select
             return new ResponseEntity<>(HttpStatus.NO_CONTENT);
        }else{
             // Código 200 OK para select
             return new ResponseEntity<>(usuarioDTO, HttpStatus.OK);
        }
    }
}
```

Uso de ModelMapper en el Service

Ahora, en tu service, puedes inyectar ModelMapper y usarlo para mapear entre la entidad Usuario y UsuarioDTO.

```
import org.modelmapper.ModelMapper;

public class UsuarioServiceImpl {

    @Autowired
    private IUsuarioRepository repo;

    @Autowired
    private ModelMapper modelMapper;

    @Override
    public UsuarioDTO listarPorId(Integer id) {
        Optional<Usuario> op = repo.findById(id);

        if(op.isPresent()){
           UsuarioDTO usuarioDTO = modelMapper.map(op.get(), UsuarioDTO.class);
           return usuarioDTO;
        }else{
           return null;
        }
    }
}
```

En este ejemplo, **modelMapper.map(usuario, UsuarioDTO.class)** realiza el mapeo de la entidad Usuario a UsuarioDTO. ModelMapper se encarga de mapear automáticamente los campos que coinciden por nombre y tipo entre las dos clases.

📖 **Práctica Recomendada:**

Estas transformaciones se hacen en la capa **Service** donde, en el método correspondiente, tendremos que implementar la lógica necesaria para mapear la información necesaria del modelo de clases a este DTO que será lo que devolvamos en la respuesta de la operación correspondiente. Ese mapeo tendrá que implementarse un tanto a mano en algunas ocasiones, pero también podremos sacar partido de librerías como ModelMapper para evitar en lo posible el código repetitivo.

Blog Óscar Blancarte:

- https://www.oscarblancarteblog.com/2018/11/30/data-transfer-object-dto-patron-diseno/

Librería ModelMapper:

- https://modelmapper.org/

Librería ModelMapper con Jey Code:

- https://www.youtube.com/watch?v=RapJOnLGvjY

Baeldung:

- https://www.baeldung.com/java-dto-pattern
- https://www.baeldung.com/entity-to-and-from-dto-for-a-java-spring-application

8. PROBAR LOS SERVICIOS WEB: POSTMAN

Si antes de integrar una aplicación con un determinado servicio web, queremos probar éste para comprobar cómo funciona, tenemos que usar aplicaciones destinadas para ese propósito, como Postman, que es una aplicación destinada exclusivamente a testear APIs.

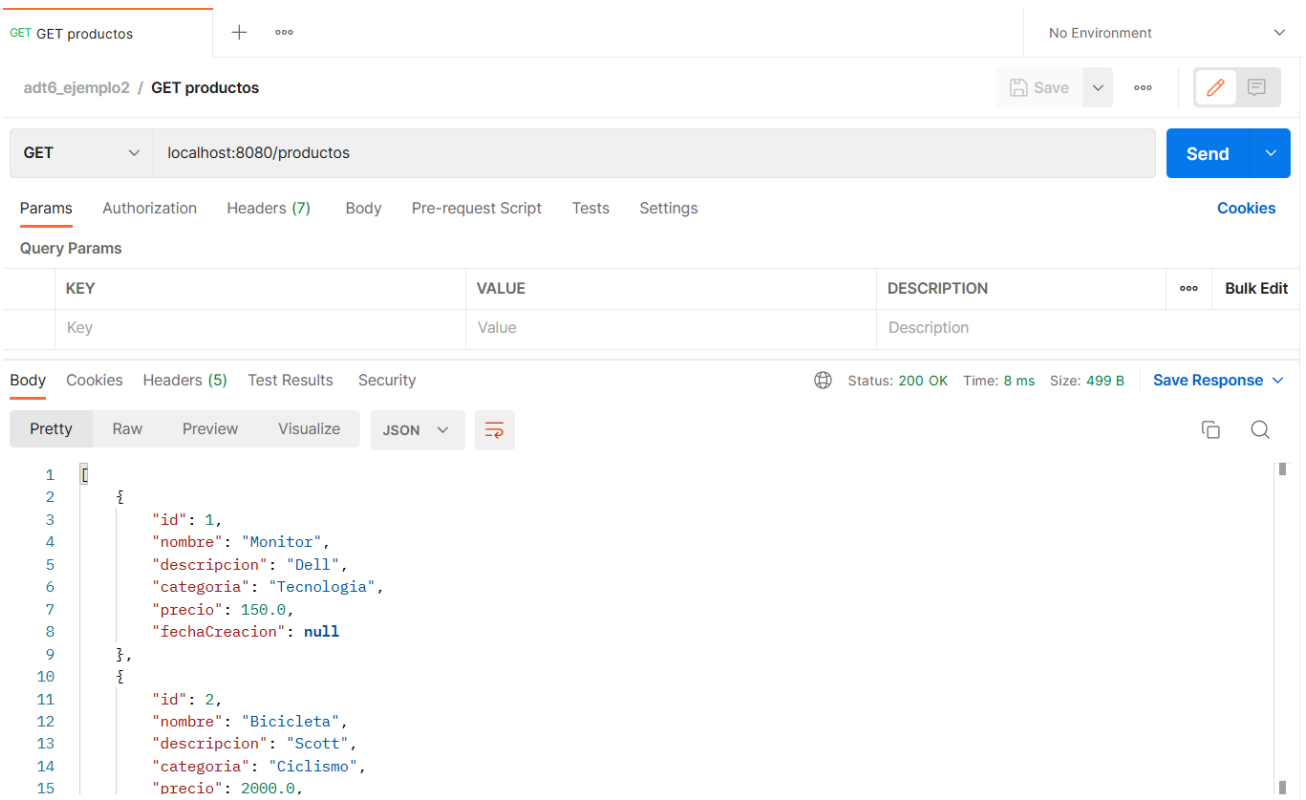
Para el servicio web desarrollado a lo largo de este tema, vamos a ver cómo se definirían una serie de pruebas para todos sus endpoints utilizando Postman.

Crearemos una colección y diferentes peticiones que nos permitan probar todos los endpoints desarrollados en este proyecto.

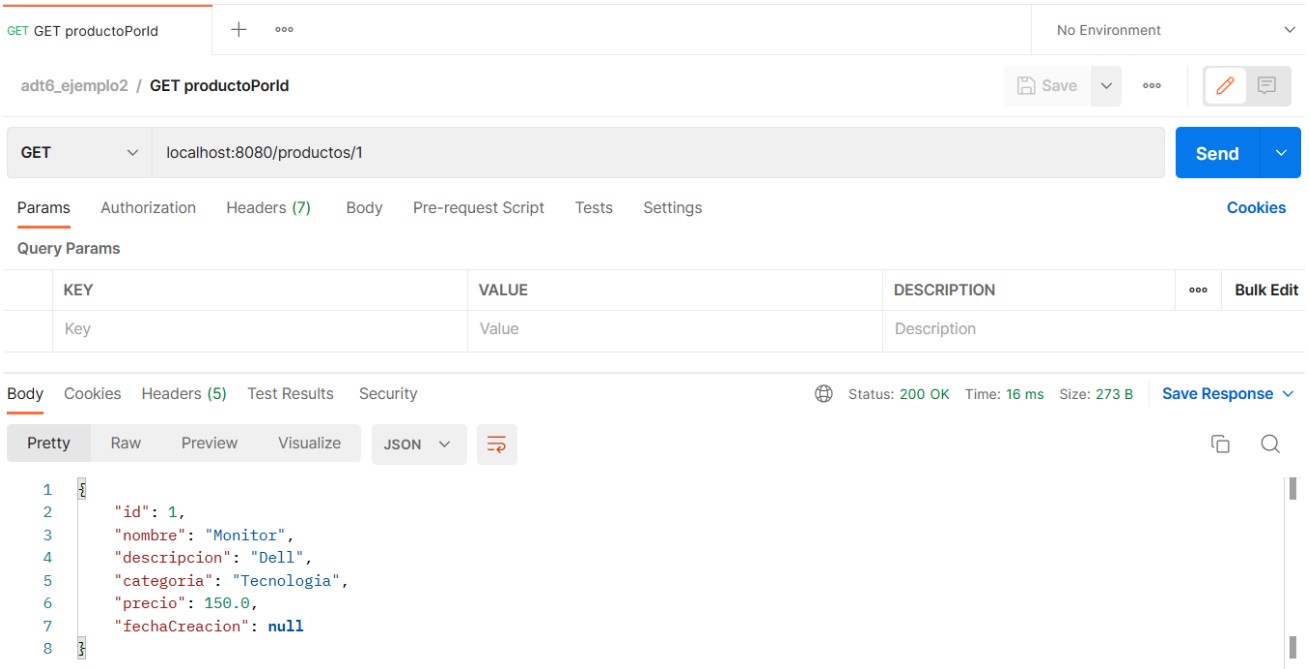
Existen muchos tutoriales de Postman, pero te dejo un vídeo de ejemplo de cómo funciona:

- Enlace

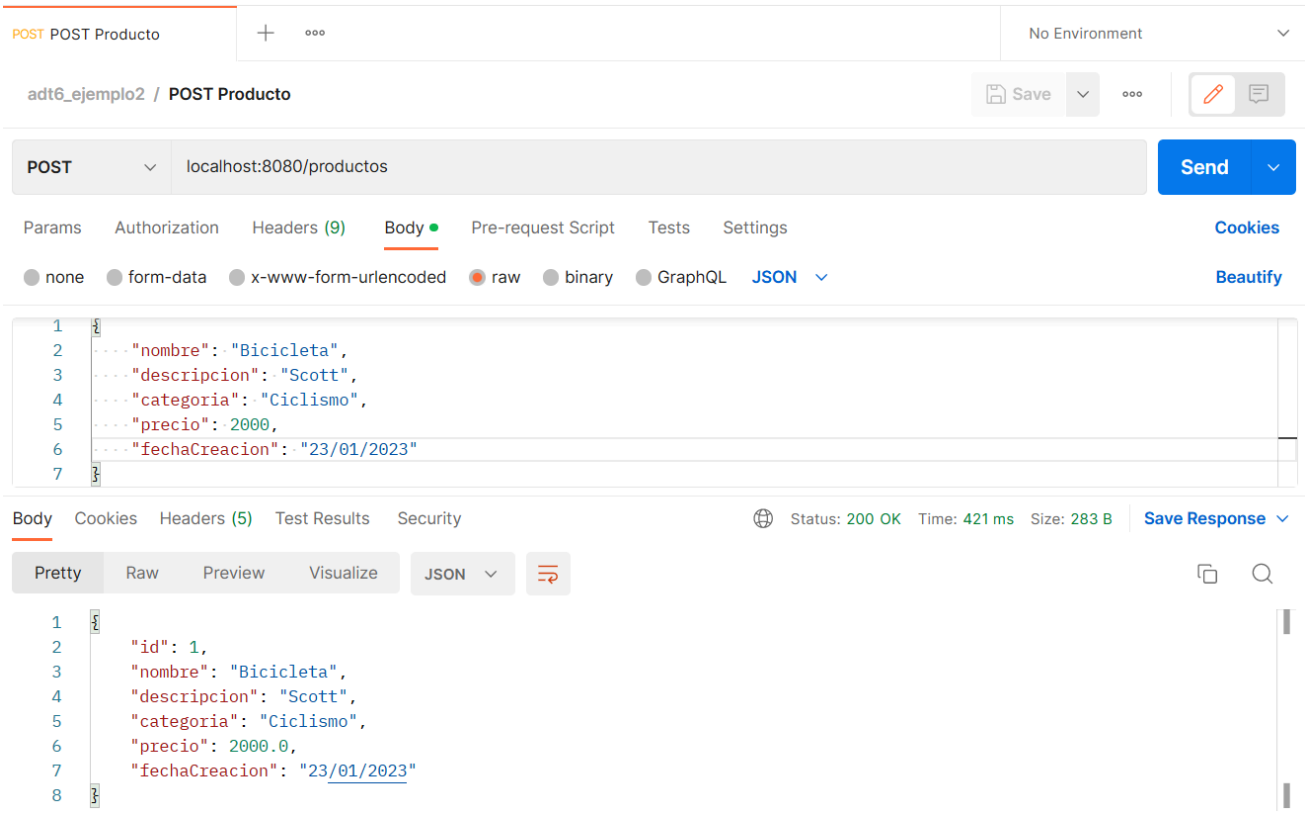
8.1. Obtener todos los productos



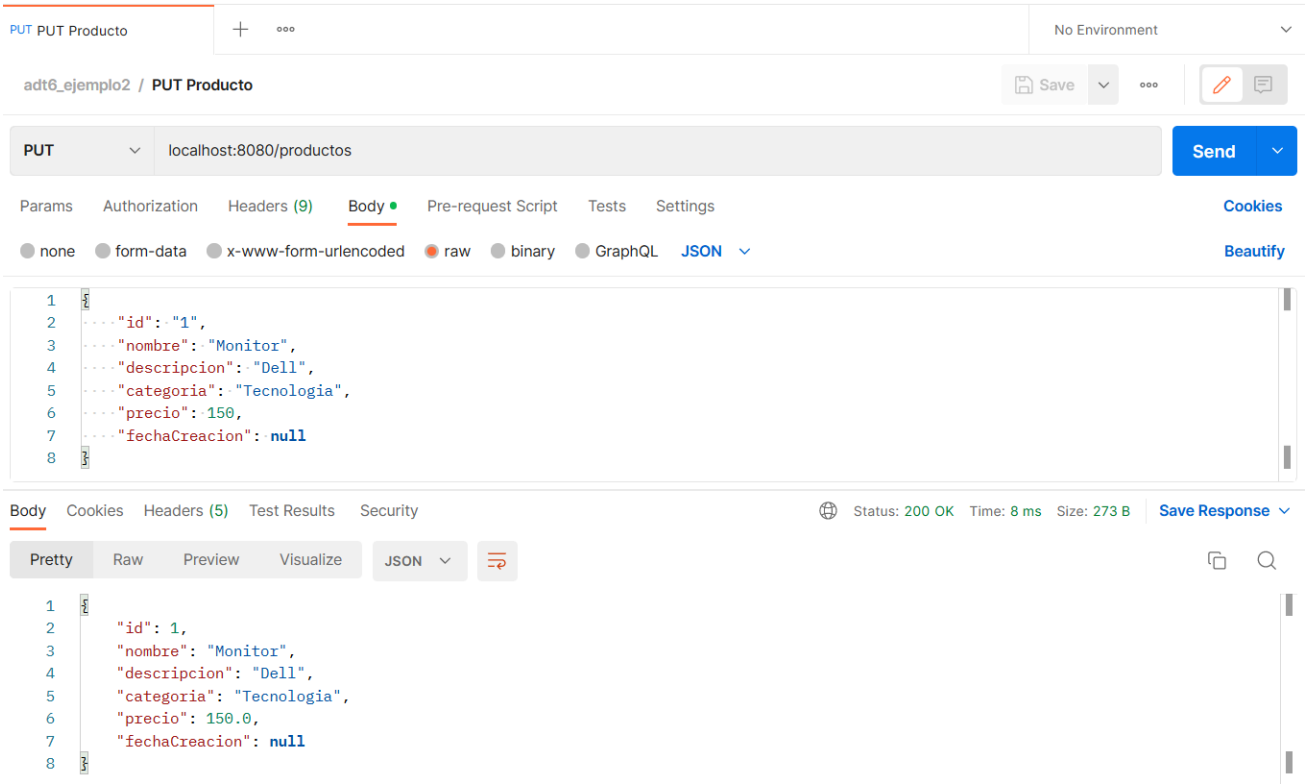
8.2. Obtiene un producto a partir de su Id



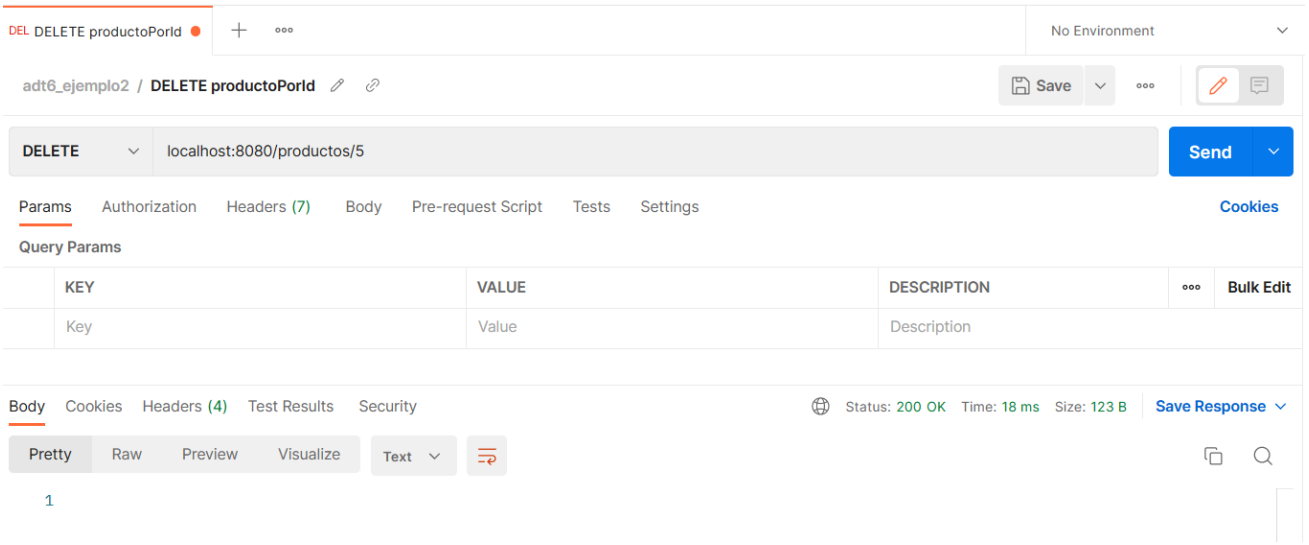
8.3. Registra un nuevo producto



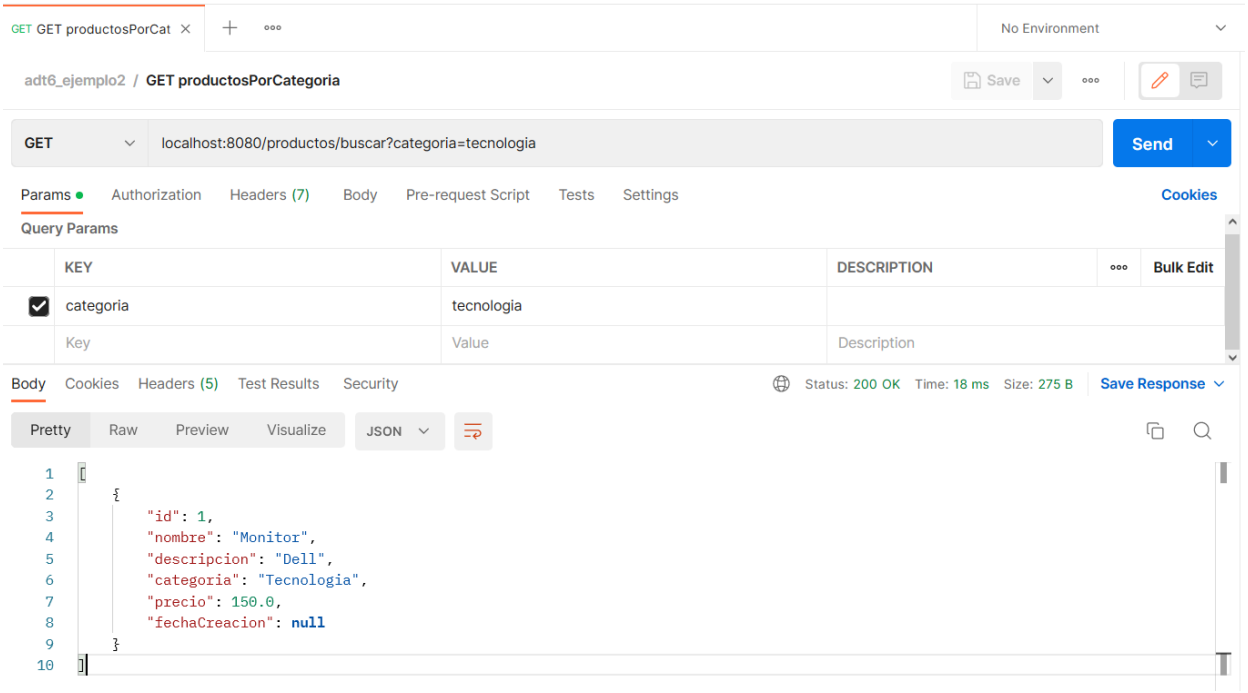
8.4. Modifica un producto existente



8.5. Elimina un producto



8.6. Obtiene todos los productos de una misma categoría



8.7. Obtiene todos los productos entre un rango de fechas