- 1. Introducción
- 2. Spring Boot
- 3. Primer servicio REST
- 4. Proyecto completo
    - 4.1 Modelo
    - 4.2 Repositorio
    - 4.3 Servicio
    - 4.2 Controlador
- 5. Mejorando el controlador
    - 5.1 GET
    - 5.2 POST
    - 5.3 PUT
    - 5.4 DELETE
    - 5.5 Uso de clases DTO

1. Introducción

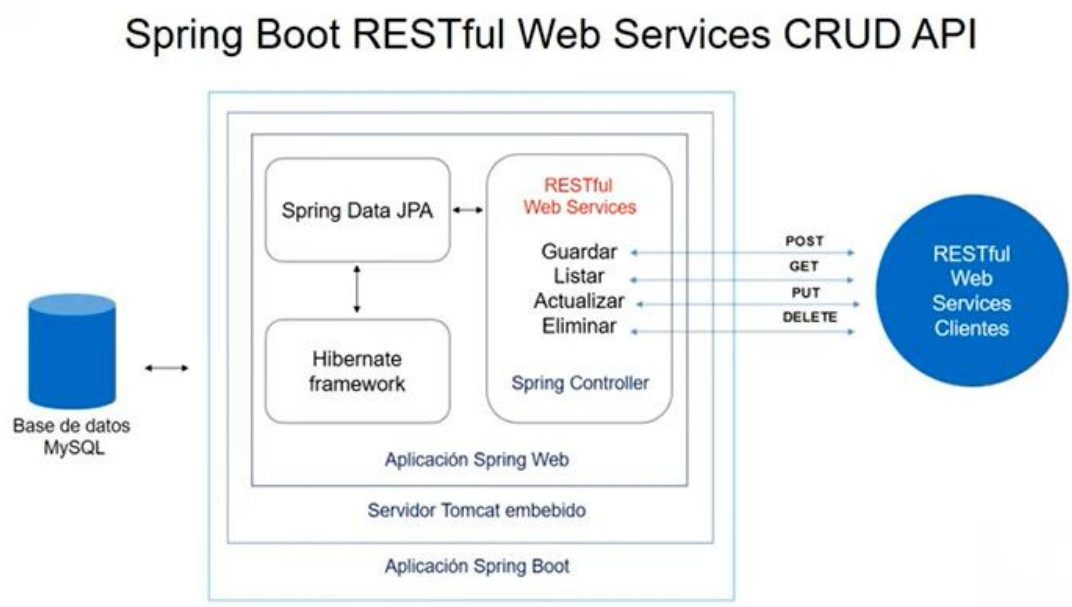
¿Qué es REST?

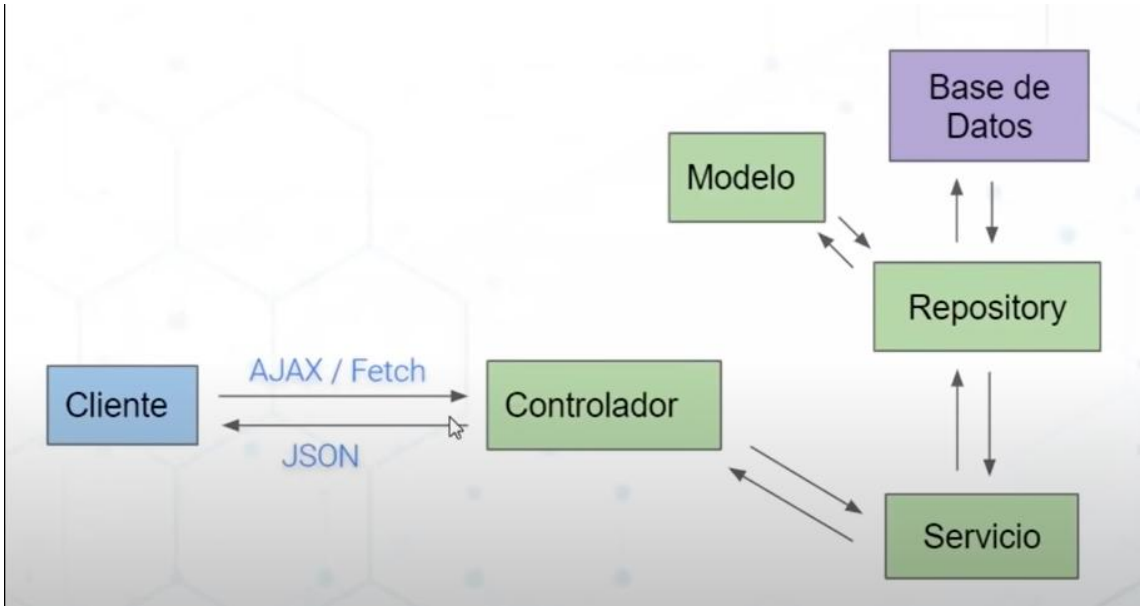
- REST es una interfaz para conectar varios sistemas basados en el protocolo **HTTP**
- Nos sirve para obtener y generar datos y operaciones
- Devuelve los datos en formatos específicos, como **XML** y **JSON**.

- El formato más usado en la actualidad es el formato **JSON**
    - Más ligero y legible en comparación a XML.
- REST se apoya en **HTTP**
- Algunos métodos básicos son:
    - *POST*: para crear recursos nuevos.
    - *GET*: para obtener un recurso en concreto.
    - *PUT*: para modificar un recurso.
    - *DELETE*: para borrar un recurso (un dato por ejemplo de nuestra base de datos)
- En nuestro servicio vamos a almacenar la **lógica de negocio**
- Vamos servir los **datos** con una serie de **recursos URL**
- La aplicación será el **BACKEND**
- La mayor **ventaja** es que nos permite **separar el frontend del backend**.

- Esto quiere decir que nuestra API REST (**backend**) se puede desarrollar con Spring Boot y JPA con acceso al almacenamiento de datos
- Y nuestro **frontend** con:
    - React o Angular por ejemplo (si fuese web)
    - App en Android o aplicación de escritorio con C#, Python o Java

- **Empresas** como como Twitter, Facebook, Google, Netflix, LinkedIn y miles de startups y empresas usan REST
- Estas API pueden ser públicas y lo pueden consumir otros usuarios, con lo cuál tenemos una forma de dar **visibilidad a nuestra API**
- Se pueden consultar miles de APIs en [ProgrammableWeb]





2. Spring Boot

- **Spring** es desde hace años el framework más popular para Java empresarial.
- Pero... ¡es difícil de configurar!

- **Spring Boot facilita la creación de aplicaciones** basadas en Spring con mínimo esfuerzo

Características

- Fácil gestión de librerías a través del fichero pom.xml (Maven)
- Servidor embebido dentro de la aplicación (dentro del jar)
- Dependencias iniciales que facilitan la configuración
- Sin tener que generar código o configuraciones

Spring initializr

- Spring nos ofrece un servicio para la generación rápida del esqueleto de la aplicación
- Se puede hacer desde https://start.spring.io/ o desde IntelliJ IDEA (integrada dentro)

3. Primer servicio REST

- El primer proyecto será un ejemplo proporcionado por Spring: https://spring.io/guides/gs/rest-service/
- Descargamos y abrimos el proyecto
- El proyecto trae dos versiones (completa e inicial). Abre la completa.
- El proyecto implementa un servicio API Rest.
- El servicio permite peticiones GET en el puerto 8080 (concretamente en la ruta http://127.0.0.1:8080/greeting)
- Devolverá un JSON

```
{
  "id": 1,
  "content": "Hello, World!"
}
```

- Si se proporciona el parámetro opcional name (http://127.0.0.1:8080/greeting?name=Luis) devolverá:

```
{
```

```
  "id": 2,
  "content": "Hello, Luis!"
}
```

- Examinamos la clase modelo **Greeting**

```
public record Greeting(long id, String content) { }
```

- Recuerda que un **record** es una clase que, con esa declaración, ya incluye dos constructores (con parámetros y sin parámetros, todos los getter y setter, toString, equals y hashCode.
- Por otro lado, tenemos un controlador **GreetingController**

```
@RestController
public class GreetingController {

        private static final String template = "Hello, %s!";
        private final AtomicLong counter = new AtomicLong();

        @GetMapping("/greeting")
        public Greeting greeting(@RequestParam(value = "name", defaultValue =
"World") String name) {
                return new Greeting(counter.incrementAndGet(),
String.format(template, name));
        }
}
```

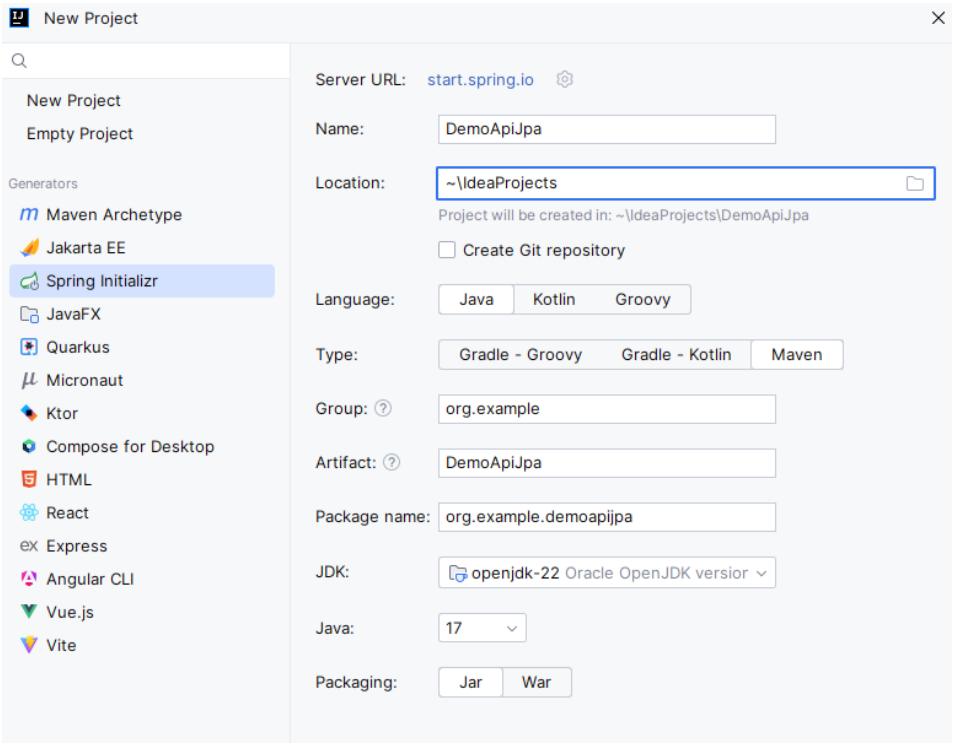
- Es básico utilizar la anotación **@RestController** para que se considere que los métodos de la clase sirven para responder a peticiones de los clientes de la API.
- La anotación **@GetMapping("/greeting")** hace se invoque al método **greeting** al hacer una solicitud http GET a la dirección http://127.0.0.1/greeting
- El parámetro **name** que recibe el método se anota con **@RequestParam**. Recogerá el **name** de la URL de la solicitud que haga el cliente y lo inyectará al parámetro **name** del método

- Ejecutaremos la aplicación

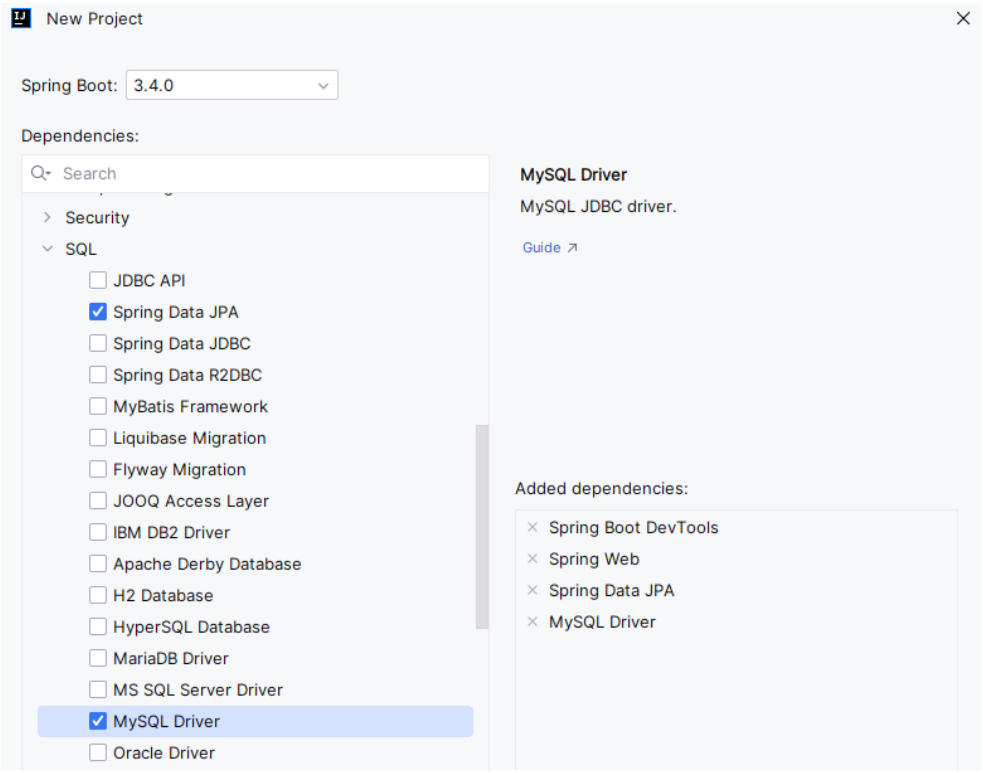
- Después, consumiremos la aplicación desde **Postman**

4. Proyecto completo

- Vamos a realizar aquí un proyecto algo más completo que el anterior
- Crearemos un proyecto en IntelliJ IDEA desde **File -> New -> Project -> Spring Initializr**
- Le pondremos un nombre, una descripción y demás datos:



- Añadimos las **dependencias**:
  - MySQL Driver
  - Spring Web
  - Spring Data JPA
  - Spring Boot DevTools
- Y **finalizar**



- En el fichero **application.properties** (en src/main/resources) estableceremos los valores de nuestra base de datos:

```
spring.datasource.url=jdbc:mysql://localhost:3306/BDcoches
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect
spring.jpa.hibernate.ddl-auto=none
spring.jpa.show-sql=true
```

4.1 Modelo

Crearemos un paquete llamado **model** y en él la clase Entity **Coche**

```
@Entity
@Table(name="coches")
public class Coche
{
        @Id
        private Integer id;

        private String marca;

        private String modelo;

        private int precio;
//constructores
//getters y setters
}
```

4.2 Repositorio

En un paquete **repository** añadiremos un repositorio.

- Será una interface llamada **CocheRepository** que herede de **JpaRepository**

```
// CocheRepository trabajará con la clase Coche cuyo Id es de tipo Integer
public interface CocheRepository extends JpaRepository<Coche, Integer>
{
//Aquí meteremos declaración de algún método que no esté en JpaRepository
}
```

4.3 Servicio

Aunque en este proyecto y para esta API muy básica no le veamos utilidad, debemos tener una capa de servicio entre la capa controlador y la de repositorio. En el servicio se implementa la lógica de negocio.

- Creamos un paquete servicio y, dentro del paquete, un interface **CocheService** que incluye un interface **CocheService**
- En el interface declararemos todos los métodos de la lógica de negocio. De momento:

```
public interface CocheService {
    List<Coche> findAll();
    Coche findById(Integer id);
}
```

En el mismo paquete creamos una clase **CocheServiceImpl** que implemente el interface:

```
@Service
public class CocheServiceImpl implements CocheService{
    //Inyección de un objeto repositorio
    @Autowired
    private CocheRepository repository;
    @Override
    public List<Coche> findAll() {
        return repository.findAll();
    }
    @Override
    public Coche findById(Integer id) {
  // el método findById de repositorio devuelve Opctional<Coche>
        return repository.findById(id).orElse(null);
    }
}
```

4.4 Controlador

- Crearemos el paquete controlller y en él la clase **CocheController**
- Será un controlador REST
- Le añadiremos el servicio que inyectará directamente

```
@RestController
@RequestMapping("/api")
public class CocheController
{
@Autowired
private CocheService service;
```

- Crearemos ahora un método **GetMapping para obtener todos los coches**

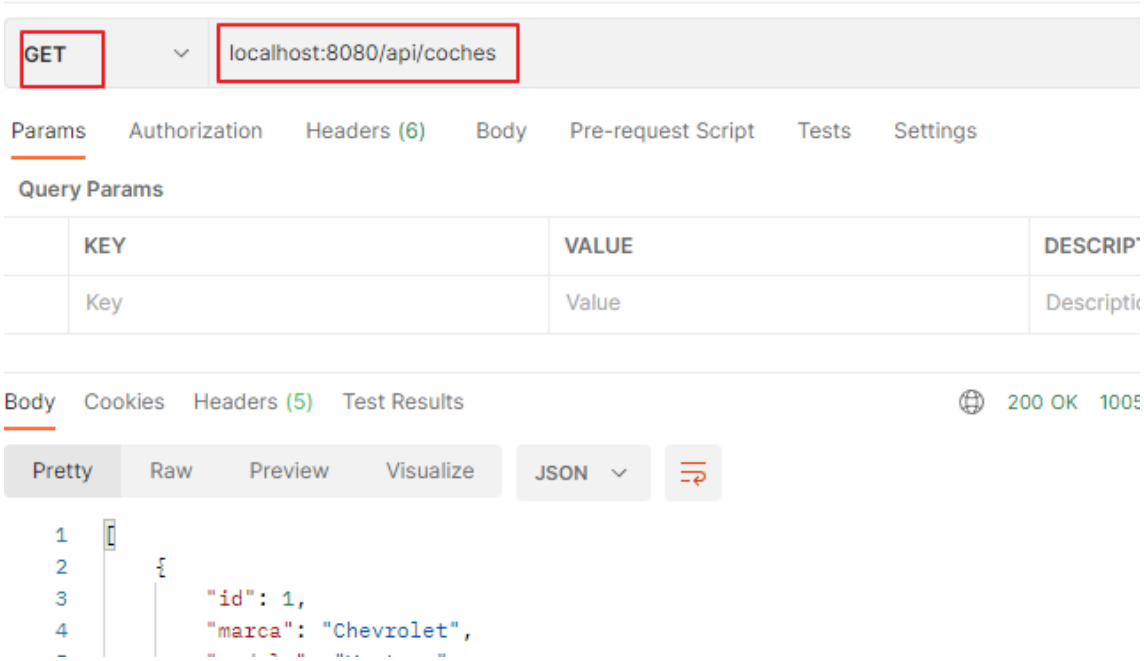
```
@RequestMapping("/api")
public class CocheController
{
@Autowired
private CocheService service;

@GetMapping("/coches")
public List<Coche> obtenerTodos()
    {
        return service.findAll();
    }
}
```

Antes de continuar vamos a crear nuestra base de datos y probar el servicio.

Para ello creamos la base de datos BDCoches y importaando el script coches.sql

Después ejecutamos la aplicación:

- Y probamos con **Postman**



Crearemos ahora un **método para devolver un sólo coche**

- De momento, si el servicio devuelve nulo en el coche, eso es lo que va a devolver al controller al cliente.

```
    @GetMapping("/coches/{id}")
    public Coche obtenerUno(@PathVariable Integer id)
    {
```

```
                return service.findById(id);
        }
```

El siguiente endpoint de la API será **para insertar un nuevo coche**

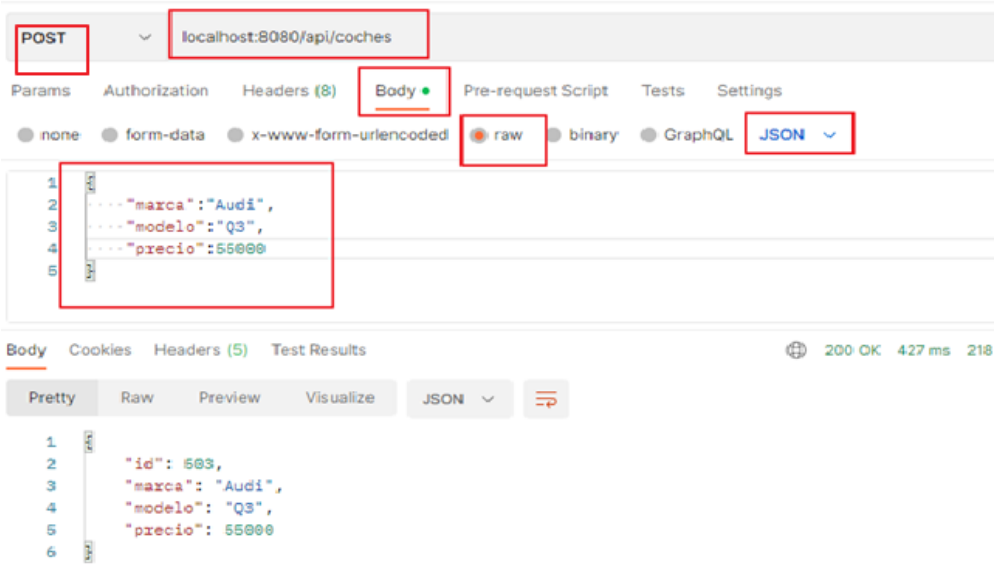
- La petición no será de tipo GET, sino de tipo **POST** ya que estamos enviando datos al servidor
- El método recibirá como parámetro un objeto Coche anotado con **@RequestBody**
- Devolverá el coche creado (con su id autoincremental asignado)

```
        @PostMapping("/coches")
        public Coche nuevoCoche(@RequestBody Coche nuevo)
        {
                return service.guardar(nuevo);  //lo mas lógico sería save
        }
```

Necesitamos añadir a Service y a ServiceImpl el método guardar(Coche)

```
@Override
public Coche guardar(Coche coche) {
    // el método save crea o modifica (si ya existe) coche
    //Devuelve el coche tras crearlo o modificarlo
    //save es un método de JpaRepository
    return repository.save(coche);
}
```

- Lo probaremos con Postman el enpoint creado. Haremos una petición POST a 127.0.0.1:8080/api/coches
- En el **Body** elegimos **raw** (de tipo **JSON**) y establecemos los valores de un coche



- Método para **editar** un coche. Será una petición de tipo **PUT**
- De momento comprobaremos que el coche exista y si existe lo guardamos.
- Si no existe devolvemos null

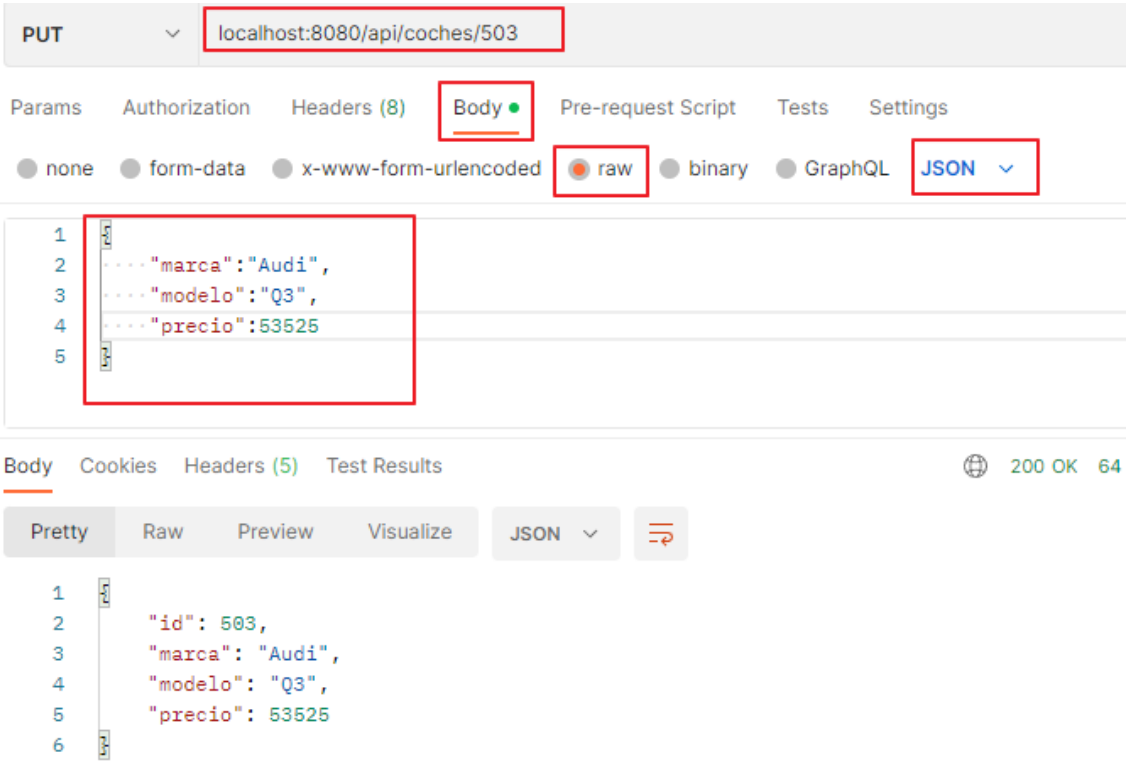
Creamos en el servicio un método para modificar un coche con los datos del nuevo coche recibido:

```
public Coche update(Coche nuevo, Integer id)
	{
		if(repository.existsById(id))
		{
			nuevo.setId(id);
			return repository.save(nuevo);
		}
		else
		{
			return null;
		}
	}
```

Y en el controller:

```
	@PutMapping("/coches/{id}")
	public Coche editarCoche(@RequestBody Coche editar, @PathVariable
Integer id)
	{

	  return service.update(editar);

	}
```

- Lo probaremos con Postman. Haremos una petición PUT a 127.0.0.1:8080/api/coches/503
- En el **Body** elegimos **raw** (de tipo **JSON**) y establecemos los valores que queramos modificar del coche



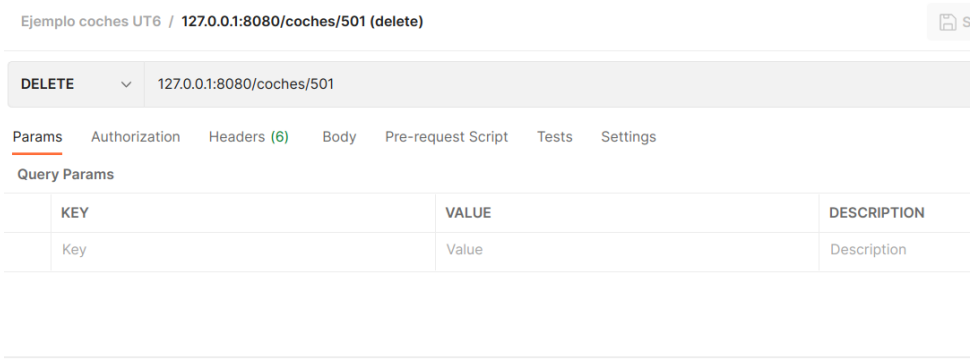
- Y por último, el método para borrar un coche.
- En el servicio

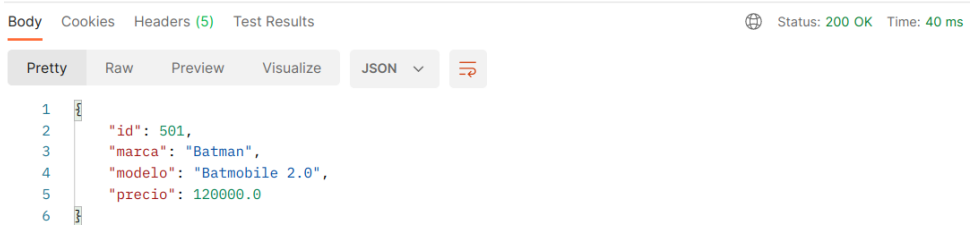
```
public Coche delete(Integer id)
	{
		if(repositiry.existsById(id))
		{
			Coche coche = repository.findById(id).get();
			repository.delete(coche);
			return coche;
		}
		else
		{
			return null;
		}
	}
```

Y en el controlador:

```
@DeleteMapping("/coches/{id}")
	public Coche borrarCoche(@PathVariable Integer id)
	{

		return cocheRepositorio.delete(coche);
	}
```

- Lo probaremos con Postman. Haremos una petición DELETE a 127.0.0.1:8080/api/coches/503





5. Mejorando el controlador

- Vamos a hacer algunas modificaciones en nuestro servicio
- Usaremos la clase **ResponseEntity** para mejorar la respuesta que enviamos al cliente.
- Nos permitirá indicar un código de respuesta además de enviar el cuerpo.

- En los métodos **GET** devolveremos un código **200 (OK)** si se localiza el coche. En caso contrario devolveremos un código **404 (Not Found)**
- En el método **POST** devolveremos el código **201 (Created)**
- En el metodo **PUT** devolveremos el código **200 (OK)** si la operación es correcta. En caso contrario el **404**
- En el método **DELETE** devolveremos un código **204 (No Content)**

5.1 GET

- Método que devuelve todos los coches

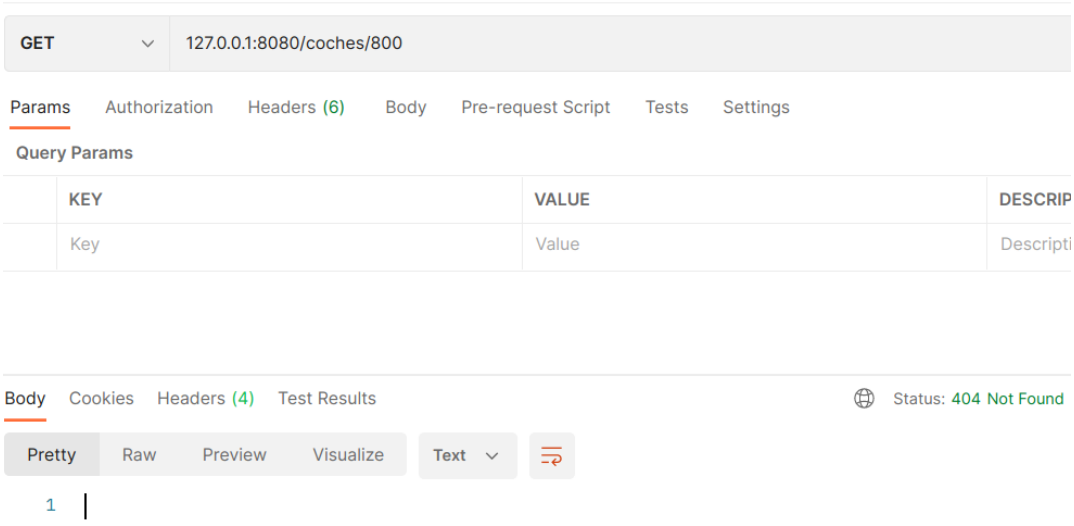
```
@GetMapping("/coches")
public ResponseEntity<?> obtenerTodos()
{
        List<Coche> coches = service.findAll();
        if(coches.isEmpty())
                return ResponseEntity.notFound().build();
        else
                return ResponseEntity.ok(coches);
}
```

- Método que devuelve un único coche pasado su id

```
@GetMapping("/coches/{id}")
public ResponseEntity<?> obtenerUno(@PathVariable Long id)
{
```

```
		Coche coche = service.findById(id);
		if(coche==null)
			return ResponseEntity.notFound().build();

		return ResponseEntity.ok(coche);
	}
```

- Podemos probar a obtener un coche con un id inexistente. Ahora nos debería dar el código 404

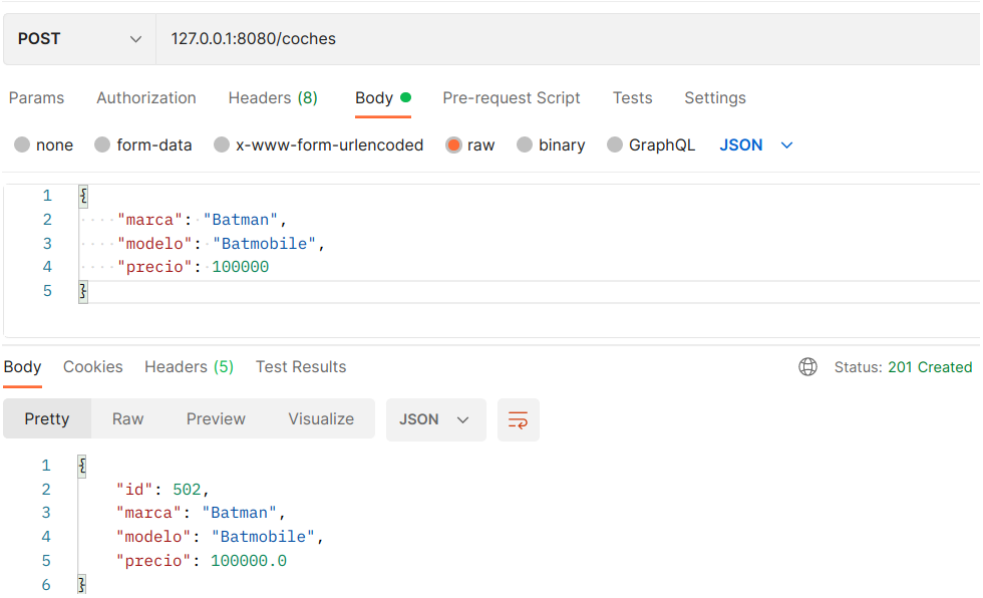


5.2 POST

- El método POST de creación de un coche devuelve el código 201 (created)

```
	@PostMapping("/coches")
	public ResponseEntity<Coche> nuevoCoche(@RequestBody Coche nuevo)
	{
		// se ha modificado el nombre guardar en service por save
		Coche guardado = service.save(nuevo);
	return ResponseEntity.status(HttpStatus.CREATED).body(guardado);
	}
```

- Probamos a guardar un nuevo coche



5.3 PUT

El método PUT de actualización de un coche lo modificamos así:

```
	@PutMapping("/coches/{id}")
	public ResponseEntity<?> editarCoche(@RequestBody Integer editar,
@PathVariable Long id)
	{
		Coche coche = cocheRepositorio.findById(id).orElse(null);
		if(coche==null)
			return ResponseEntity.notFound().build();

		coche.setMarca(editar.getMarca());
		coche.setModelo(editar.getModelo());
		coche.setPrecio(editar.getPrecio());

		return ResponseEntity.ok(cocheRepositorio.save(coche));
	}
```

Y habremos modificado el método update en Servicio asï:

```
@Override
public Coche update(Coche editar, Integer id) {
   Coche coche=repository.findById(id).orElse(null);
   if(coche!=null){
      coche.setMarca(editar.getMarca());
      coche.setModelo(editar.getModelo());
      coche.setPrecio(editar.getPrecio());
      repository.save(coche);
   }
```

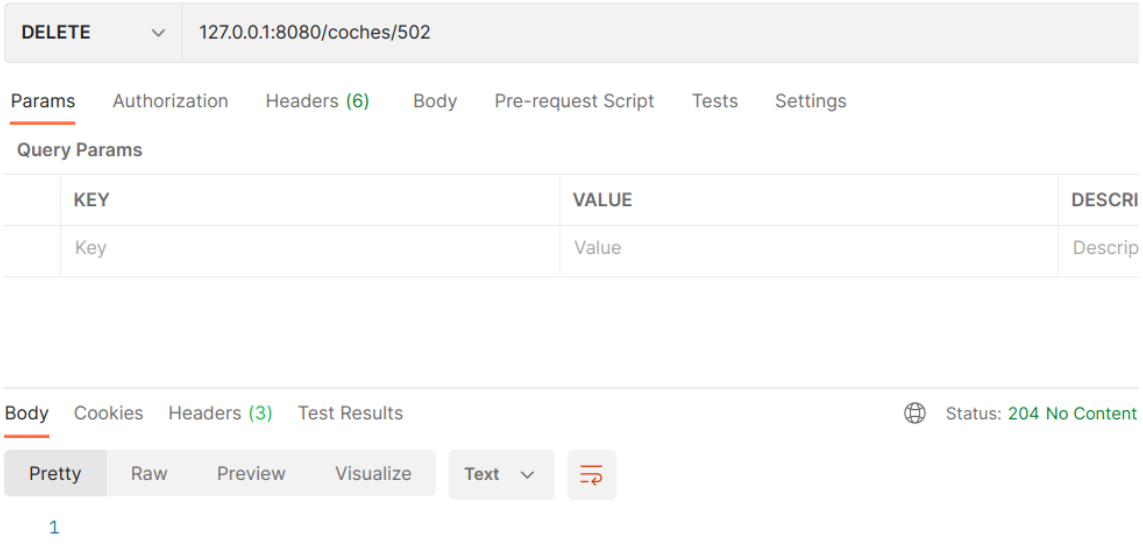
```
    return coche;
}
```

5.4 DELETE

- Por último, el método DELETE para que devuelva el código 204 (No Content)

```
@DeleteMapping("/coches/{id}")
public ResponseEntity<?> borrarCoche(@PathVariable Integer id)
{
        cocheRepositorio.deleteById(id);
        return ResponseEntity.noContent().build();
}
```

- Probamos a borrar el coche creado antes



5.5 Uso de clases DTO

En el contexto del desarrollo de una API Rest Spring, una clase DTO facilita que el controlador pueda intercambiar información de las entidades de forma que:

- No se envíen todos los datos de una entidad. Por ejemplo, datos personales, datos de una lista de objetos, etc.
- Se envíen datos adicionales a los que contiene una entidad. Por ejemplo, en una entidad Propietario no se tiene una lista de los Coches de los que cada persona es propietaria y se quiere que se envíe esa lista.

Para ver como funcionan los DTO vamos a ver un ejemplo pero antes:

En la tabla **coches,** usando Workbench, añadimos una columna clave ajena propietario que contendrá el id del propietario de cada coche. La columna admite nulos ya que inicialmente tendremos todos los coches sin propietario. Cargamos algunos propietarios en los coches.

En nuestra API, añadimos una Entity Class **Propietario:**

```
@Entity
@Table(name="propietarios")
public class Propietario {

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

    @Column(length=30,nullable=false)
    private String nombre;

    @Enumerated(EnumType.STRING) //Guarda valores del enum como texto
    private SituacionEnum situacion;
```

Y en la Entity Class **Coche** añadimos una asociación **ManyToOne** para representar al propietario del coche:

```
@ManyToOne
@JoinColumn(name="propietario")
private Propietario propietario;
```

Si lanzamos la API y probamos los endpoint GET, veremos que ahora se envían en los datos de los coches los datos de sus propietarios con un objeto JSON interno en cada coche.

Supongamos que no queremos que se envíen todos los datos del propietario sino solamente el id del propietario junto con todos los datos de cada coche. Tenemos que hacer una clase DTO que contenga los atributos que se van a enviar. La clase DTO la podemos llamar como queramos, pero, si solo hay un DTO para Coche, lo mejor es llamarla CocheDTO. Esta clase puede incluirse dentro del package model aunque se puede usar otro package para los DTO.

```
public class CocheDTO {

    private Integer id;
    private String marca;
    private String modelo;
    private int precio;
    private Integer idPropietario;
// Constructores, getter, setter
}
```

Aunque no es necesario, se debe crear una clase que contenga métodos mapper para convertir un Coche en un CocheDTO y un CocheDTO en un Coche. Las clases

mapper es adecuado tenerlas en un package mapper. Añadimos la clase CocheMapper:

```
public class CocheMapper {

    public CocheDTO toDTO(Coche coche) {
        CocheDTO dto = new CocheDTO();
        dto.setId(coche.getId());
        dto.setMarca(coche.getMarca());
        dto.setModelo(coche.getModelo());
        dto.setPrecio(coche.getPrecio());
        dto.setIdPropietario(coche.getPropietario() != null ?
coche.getPropietario().getId() : null);
        return dto;
    }

 }
```

El método **toDTO** convierte un **Coche** en un **CocheDTO**. Debería incluirse también un método de conversión inverso **toEntity**, pero no lo vamos a hacer (exigiría usar el repositorio).

Ahora rehacemos los endpoints **GET** del controlador para que trabajen con **CocheDTO**:

```
@GetMapping("/coches/{id}")
private ResponseEntity<?> obtenerCochePorId(@PathVariable int id){
    CocheDTO coche = service.findDTOById(id);
    if(coche==null)
        return ResponseEntity.notFound().build();
    else
        return ResponseEntity.ok(coche);
}
```

Como en **ServiceCoche** no tenemos el método **findDTOById**, lo tenemos que declarar e implementarlo en **ServiceCocheImpl**. Además, en **ServiceCocheImpl** tenemos que construir un objeto **mapper** para usarlo en **findDTOById**:

```
private CocheMapper mapper=new CocheMapper();

@Override
public CocheDTO findDTOById(int id) {
    Coche coche=findById(id);
    CocheDTO cocheDTO=mapper.toDTO(coche);
    return cocheDTO;
}
```