TRABAJO FINAL
GRAN DAO

MARÍA LÓPEZ PATÓN Y UNAI NIETO JIMÉNEZ

ACCESO A DATOS

03/03/2025

ÍNDICE

1. BASES DE DATOS RELACIONALES SQL (MARIADB) 3

1.1 AUTORES 3

1.1.1 GET ALL AUTORES 3

1.1.2 GET AUTOR BY ID 3

1.1.3 POST AUTOR 4

1.1.4 POST AUTOR MEDIANTE PARÁMETROS 5

1.1.5 UPDATE AUTOR 6

1.1.6 DELETE AUTOR 7

1.2 LIBROS 8

1.2.1 GET ALL LIBROS 8

1.2.2 GET LIBRO BY ID 8

1.2.3 POST LIBRO 9

1.2.4 POST LIBRO MEDIANTE PARÁMETROS 10

1.2.5 UPDATE LIBRO 11

1.2.6 DELETE LIBRO 12

2. BASES DE DATOS NO RELACIONALES NOSQL (MONGODB) 14

2.1 CIUDADES 14

2.1.1 GET ALL CIUDADES 14

2.1.2 GET CIUDAD BY ID 14

2.1.3 POST CIUDAD 15

2.1.4 UPDATE CIUDAD 16

2.1.5 DELETE CIUDAD 16

3. FICHEROS XML 18

3.1 USUARIOS 18

3.1.1 GET USUARIOS 18

3.1.2 GET USUARIO BY NOMBRE 19

3.1.3 POST USUARIO 20

3.1.4 UPDATE USUARIO 22

3.1.5 DELETE USUARIO 23

4. FICHEROS Y DIRECTORIOS 24

4.1 COCHES 24

4.1.1 GET ALL COCHES 24

4.1.2 GET COCHE BY MATRÍCULA 25

4.1.3 POST COCHE 26

1. BASES DE DATOS RELACIONALES SQL (MARIADB)

En este primer caso, utilizaremos una BDD relacional con dos tablas, Autores y libros.

1.1 AUTORES

1.1.1 GET ALL AUTORES

Código del Service:

Mediante el repositorio *repositorioAutores*, obtiene todos los autores de la BDD y devuelve la lista obtenida.

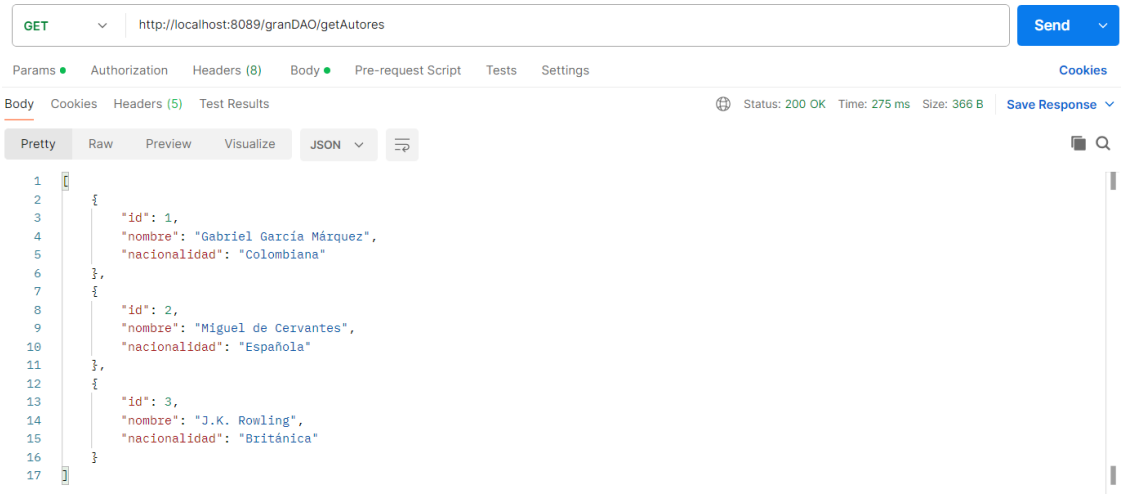
```
//Obtiene todos los autores desde la base de datos
public List<Autores> getAutores() {  1 usage   mariaasir
    return repositorioAutores.findAll();
}
```

Código del controlador:

Devuelve una lista de todos los autores almacenados en la BDD mediante el Service.

```
//GET ALL AUTORES --> SELECT *
@GetMapping("/getAutores")   mariaasir
public ResponseEntity<List<Autores>> getAutores() {
    //Llama al Service para obtener todos los autores desde la base de datos
    return ResponseEntity.ok(service.getAutores()); //Recibe una respuesta con todos los autores
}
```

Comprobación en PostMan:



1.1.2 GET AUTOR BY ID

Código del Service:

Busca un autor por su ID desde la BDD utilizando repositorioAutores. El uso de .get() devolverá una excepción si el ID no existe.

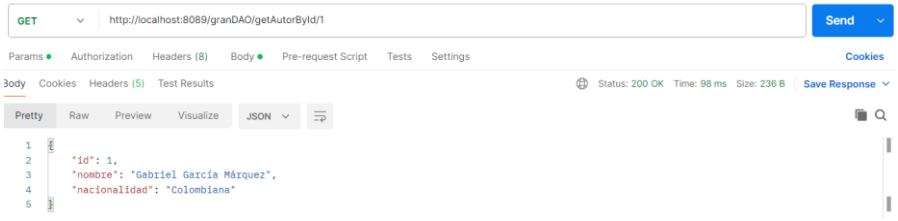
```
//Obtiene un autor por su ID desde la base de datos
public Autores getAutorById(int id) {  3 usages   mariaasir
    return repositorioAutores.findById(id).get();
}
```

Código del controlador:

Llama al Service para buscar el autor. Recibe el ID a través de @PathVariable.

```
//GET AUTOR BY ID
@GetMapping("/getAutorById/{id}")  mariaasir
@Cacheable
public ResponseEntity<Autores> getAutorById(@PathVariable int id) {
    //Llama al Service para obtener el autor por ID
    return ResponseEntity.ok(service.getAutorById(id)); // Si el autor existe, devuelve el Objeto de tipo Autor
}
```

Comprobación en PostMan:



1.1.3 POST AUTOR

Código del Service:

Guarda un nuevo autor en la BDD utilizando *repositorioAutores*.

```
//Agrega un nuevo autor a la base de datos
public Autores addAutor(@Valid Autores autor) {  1 usage  mariaasir
    return repositorioAutores.save(autor);
}
```

Código del controlador:

Usa @RequestBody para recibir el autor y @Valid para verificar que cumple las validaciones. Llama al Service y devuelve el autor creado.

```
//POST de un Objeto autor
@PostMapping("/postAutor")  mariaasir
public ResponseEntity<Autores> addAutor(@Valid @RequestBody Autores autor) {
    //Llama al Service para agregar un nuevo autor a la BDD
    return ResponseEntity.ok().body(service.addAutor(autor));  //Devuelve el Objeto del autor creado
}
```

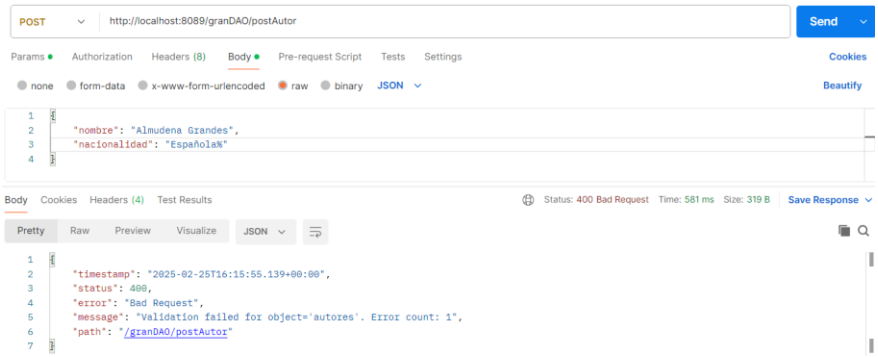
Validaciones:

Para añadir un autor tendremos que tener en cuenta una serie de validaciones. El nombre de autor no puede estar vacío y la nacionalidad solo podrá contener letras y espacios.

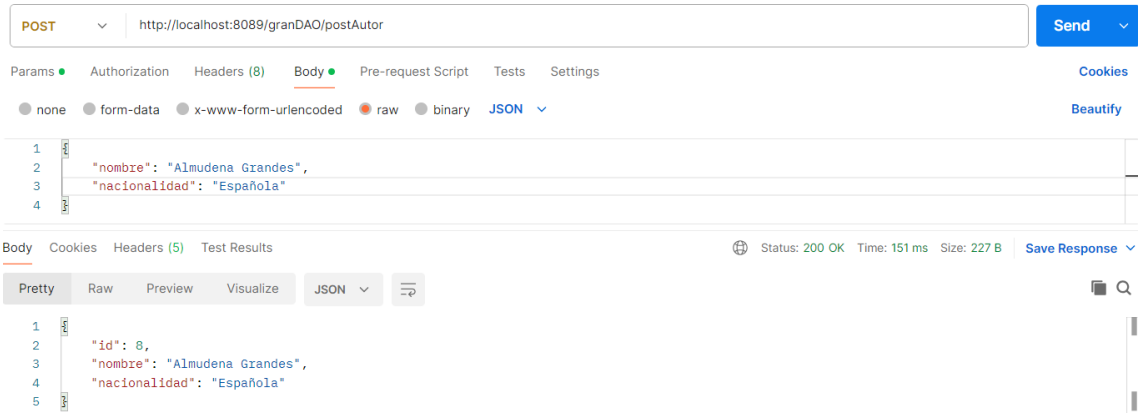
```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Column(name = "id", nullable = false)
private Integer id;

@Size(max = 100)  4 usages
@NotBlank(message = "El nombre no puede estar vacío")
@Column(name = "nombre", nullable = false, length = 100)
private String nombre;

@Size(max = 50)  4 usages
@Pattern(regexp = "^[a-zA-ZÀ-ÿ\\s]*$", message = "La nacionalidad solo puede contener letras y espacios")
@Column(name = "nacionalidad", length = 50)
private String nacionalidad;
```

Comprobación en PostMan:

Primero realizaremos un post en el que la nacionalidad contendrá “%” y veremos que nos muestra un error, comprobando así que la validación al hacer insertar un Autor funciona correctamente.



Ahora realizaremos un post con todos los parámetros válidos.



Comprobación de la inserción en la BDD:

	id	nombre	nacionalidad
Editar Copiar Borrar	1	Gabriel García Márquez	Colombiana
Editar Copiar Borrar	2	Miguel de Cervantes	Española
Editar Copiar Borrar	3	J.K. Rowling	Británica
Editar Copiar Borrar	8	Almudena Grandes	Española

1.1.4 POST AUTOR MEDIANTE PARÁMETROS

Código del Service:

Recibe los parámetros nombre y nacionalidad y crea un nuevo Autor. Llama al repositorio para guardarlo.

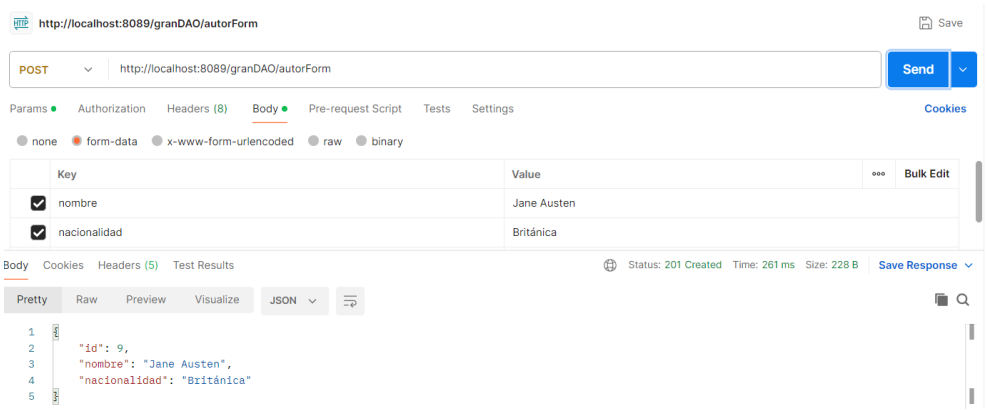
```
//Agrega un autor usando parámetros
public Autores addAutorParameters(@Valid  String nombre, @Valid String nacionalidad) {  1 usage   mariaasir *
    Autores autor = new Autores();
    autor.setNombre(nombre);
    autor.setNacionalidad(nacionalidad);
    return repositorioAutores.save(autor);
}
```

Código del controlador:

Recibe los parámetros con @RequestParam y devuelve el objeto del autor creado llamando al Service.

```
//POST con Form Normal
@PostMapping(value = "/autorForm", consumes = MediaType.MULTIPART_FORM_DATA_VALUE)   mariaasir
public ResponseEntity<Autores> addAutorForm(
        @RequestParam String nombre,  //Recibe el nombre del autor como parámetro del formulario
        @RequestParam String nacionalidad //Recibe la nacionalidad del autor como parámetro del formulario
) {
    //Llama al Service para agregar un nuevo autor con los parámetros recibidos del formulario
    return ResponseEntity.created( location: null).body(service.addAutorParameters(nombre, nacionalidad)); //Devuelve el Objeto del autor creado
}
```

Comprobación en PostMan:



Comprobación de la inserción en la BDD:

		id	nombre	nacionalidad
☐	Editar Copiar Borrar	1	Gabriel García Márquez	Colombiana
☐	Editar Copiar Borrar	2	Miguel de Cervantes	Española
☐	Editar Copiar Borrar	3	J.K. Rowling	Británica
☐	Editar Copiar Borrar	8	Almudena Grandes	Española
☐	Editar Copiar Borrar	9	Jane Austen	Británica

1.1.5 UPDATE AUTOR

Código del Service:

Recibe el autor y verifica que cumpla con las validaciones con @Valid. Actualiza los datos del autor guardándolo en el repositorio.

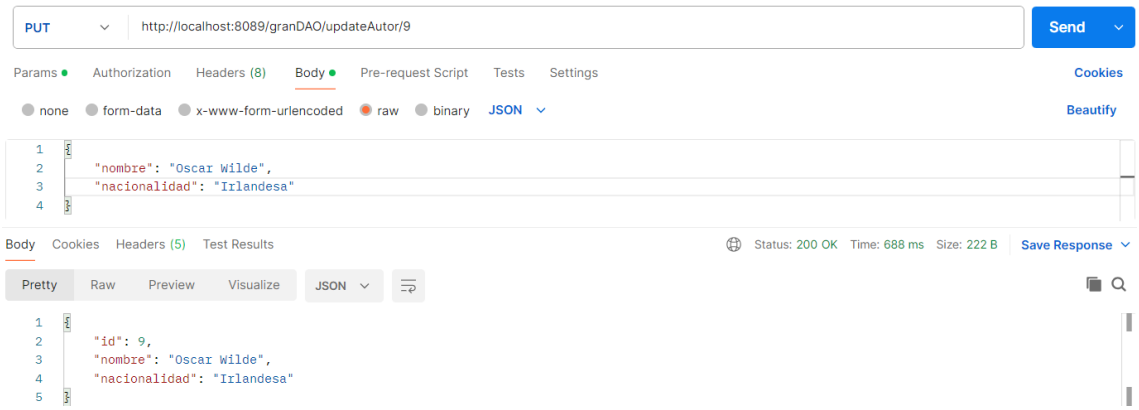
```
//Actualiza los datos de un autor en la base de datos
public Autores updateAutor(@Valid Autores autor) {  1 usage   mariaasir *
    return repositorioAutores.save(autor);
}
```

Código del controlador:

Busca el autor por su ID y si existe lo guarda en un nuevo autor llamado “autorExistente”. Si el autor no existe, devuelve un error. Si existe modifica su nacionalidad y su nombre. Llama al Service y actualiza el autor. Devuelve el objeto autor con ResponseEntity.

```
//PUT AUTOR --> UPDATE OBJETO AUTOR
@PutMapping("/updateAutor/{id}")   mariaasir
public ResponseEntity<Autores> updateAutor(@PathVariable int id, @RequestBody Autores autor) {
    Autores autorExistente = service.getAutorById(id);          //Busca al autor por su ID
    if (autorExistente == null) {
        return ResponseEntity.notFound().build(); // Si el autor no existe, devuelve un error NOT FOUND
    }
    //Actualiza los campos del autor existente con los datos proporcionados
    autorExistente.setNacionalidad(autor.getNacionalidad());
    autorExistente.setNombre(autor.getNombre());

    Autores autorActualizado = service.updateAutor(autorExistente);          //Llama al Service para actualizar el autor
    return ResponseEntity.ok().body(autorActualizado);          //Devuelve el Objeto Autor con los parámetros actualizados

}
```

Comprobación en PostMan:



Comprobación de la modificación en la BDD:

	id	nombre	nacionalidad
Editar Copiar Borrar	1	Gabriel García Márquez	Colombiana
Editar Copiar Borrar	2	Miguel de Cervantes	Española
Editar Copiar Borrar	3	J.K. Rowling	Británica
Editar Copiar Borrar	8	Almudena Grandes	Española
Editar Copiar Borrar	9	Oscar Wilde	Irlandesa

1.1.6 DELETE AUTOR

Código del Service:

Recibe el ID del autor a eliminar, llama al repositorio y lo elimina.

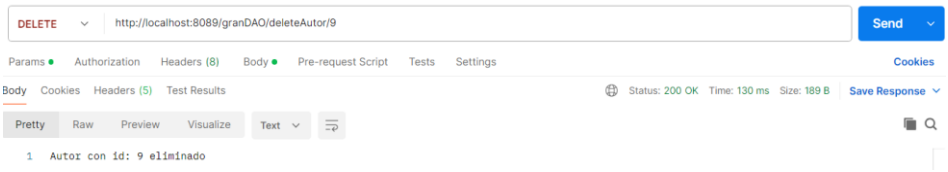
```
//Elimina un autor de la base de datos por su ID
public void deleteAutor(int id) {  1 usage   mariaasir
    repositorioAutores.deleteById(id);
}
```

Código del controlador:

Llama al Service y elimina el autor, muestra un mensaje de que se ha eliminado correctamente.

```
//DELETE --> DELETE DE UN OBJETO AUTOR
@DeleteMapping("/deleteAutor/{id}")   mariaasir
public ResponseEntity<String> deleteAutor(@PathVariable int id) {
    //Llama al Service para eliminar el autor de la BDD
    service.deleteAutor(id);
    String mensaje = "Autor con id: " + id + " eliminado";  //Muestra un mensaje de que el autor ha sido eliminado con exito
    return ResponseEntity.ok().body(mensaje);
}
```

Comprobación en PostMan:



Comprobación de la eliminación del autor en la BDD:

	id	nombre	nacionalidad
Editar Copiar Borrar	1	Gabriel García Márquez	Colombiana
Editar Copiar Borrar	2	Miguel de Cervantes	Española
Editar Copiar Borrar	3	J.K. Rowling	Británica
Editar Copiar Borrar	8	Almudena Grandes	Española

1.2 LIBROS

1.2.1 GET ALL LIBROS

Código del Service:

Mediante el repositorio *repositorioLibros*, obtiene todos los libros de la BDD y devuelve la lista obtenida.

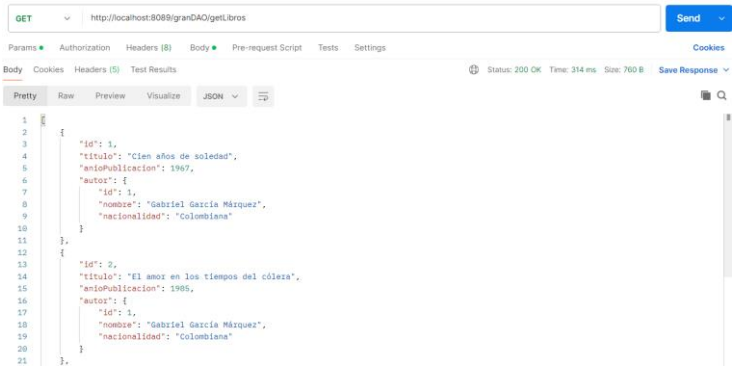
```
//Obtiene todos los libros desde la base de datos
public List<Libros> getLibros() {  1 usage   mariaasir
    return repositorioLibros.findAll();
}
```

Código del controlador:

Llama al Service para obtener la lista de todos los libros de la BDD.

```
//GET ALL LIBROS --> SELECT *
@GetMapping("/getLibros")   mariaasir
public ResponseEntity<List<Libros>> getLibros() {
    //Llama al Service para obtener todos los libros desde la base de datos
    return ResponseEntity.ok(service.getLibros()); //Recibe una respuesta con todos los libros
}
```

Comprobación en PostMan:



1.2.2 GET LIBRO BY ID

Código del Service:

Recibe por parámetro el ID del libro y lo busca en la BDD mediante el repositorio.

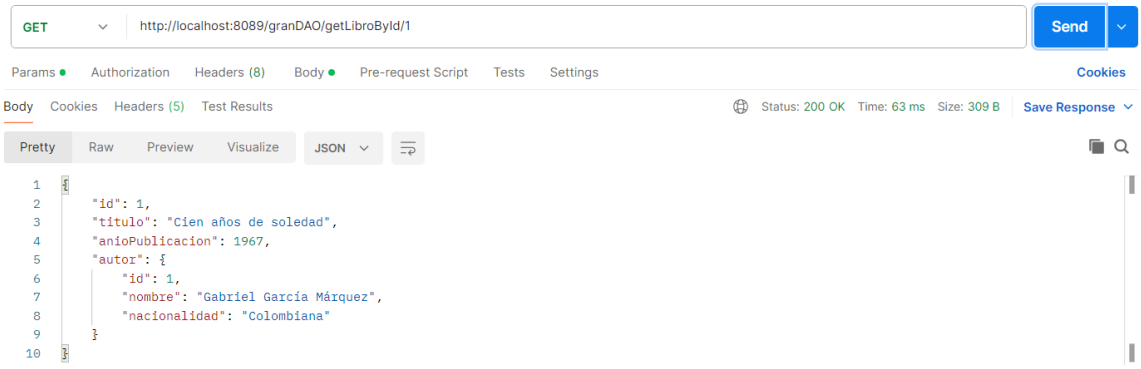
```
//Obtiene un libro por su ID desde la base de datos
public Libros getLibroById(int id) {  2 usages   mariaasir
    return repositorioLibros.findById(id).get();
}
```

Código del controlador:

Recibe por parámetro el ID del libro a buscar y llama al Service para obtener el objeto del libro deseado.

```
//GET LIBRO BY ID
@GetMapping("/getLibroById/{id}")   mariaasir
@Cacheable
public ResponseEntity<Libros> getLibroById(@PathVariable int id) {
    //Llama al Service para obtener el libro por ID
    return ResponseEntity.ok(service.getLibroById(id)); // Si el libro existe, devuelve el Objeto de tipo Libro
}
```

Comprobación en PostMan:



1.2.3 POST LIBRO

Código del Service:

Recibe por parámetro el Objeto del libro a añadir y con @Valid comprueba que cumple con las validaciones establecidas. Si cumple con ellas, lo guarda en la BDD mediante el repositorio de libros.

```
//Agrega un nuevo libro a la base de datos
public Libros addLibro(@Valid Libros libro) {  1 usage   mariaasir
    return repositorioLibros.save(libro);
}
```

Código del controlador:

Recibe el Objeto libro y lo valida con @Valida. Añade el objeto libro mediante el método *addLibro* del Service y lo muestra con ResponseEntity.

```
//POST DE UN OBJETO LIBRO
@PostMapping("/postLibro")   mariaasir
public ResponseEntity<Libros> addLibro(@Valid @RequestBody Libros libro) {
    //Llama al Service para agregar un nuevo libro a la BDD
    return ResponseEntity.ok().body(service.addLibro(libro)); //Devuelve el Objeto del libro creado
}
```

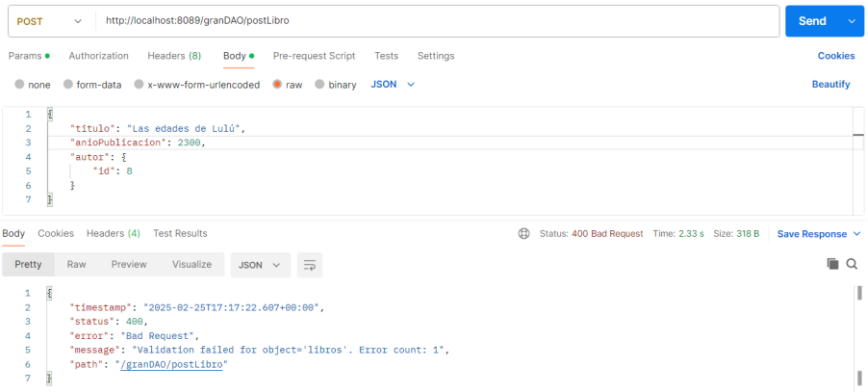
Validaciones:

Para añadir un libro tendremos que tener en cuenta una serie de validaciones. El título del libro no puede estar vacío y debe contener caracteres alfanuméricos y signos de puntuación. El año de publicación debe ser mayor al año 1000 y tiene que ser inferior o igual al año actual.

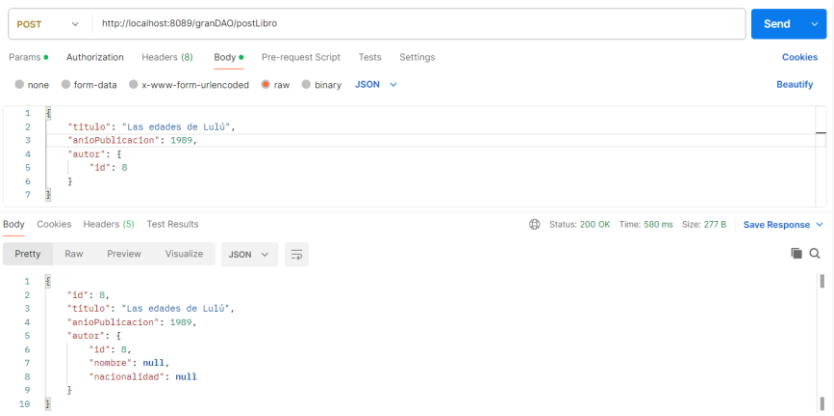
```
@Size(max = 150)  4 usages
@NotBlank(message = "El titulo no puede estar vacio")
@Pattern(regexp = "^[a-zA-Z0-9À-ÿ\\s.,'-]+$", message = "El titulo contiene caracteres no permitidos")
@Column(name = "titulo", nullable = false, length = 150)
private String titulo;

@Min(value = 1000, message = "El año de publicación debe ser válido")  4 usages
@Max(value = Year.MAX_VALUE, message = "El año de publicación no puede ser superior al actual")
@Column(name = "anio_publicacion")
private Integer anioPublicacion;
```

Comprobación en PostMan:

Primero realizaremos un post en el que el año de publicación será mayor al año actual y veremos que nos muestra un error, comprobando así que la validación al hacer insertar un Libro funciona correctamente.



Ahora realizaremos un post con todos los datos correctos.



Comprobación de la inserción en la BDD:

☐	Editar Copiar Borrar	1	Cien años de soledad	1967	1
☐	Editar Copiar Borrar	2	El amor en los tiempos del cólera	1985	1
☐	Editar Copiar Borrar	3	Don Quijote de la Mancha	1605	2
☐	Editar Copiar Borrar	4	Harry Potter y la piedra filosofal	1997	3
☐	Editar Copiar Borrar	8	Las edades de Lulú	1989	8

1.2.4 POST LIBRO MEDIANTE PARÁMETROS

Código del Service:

Recibe los parámetros del libro y valida que cumpla con las validaciones establecidas. Si cumple con ellas, se crea un nuevo libro y se guarda en la BDD mediante el repositorio de libros.

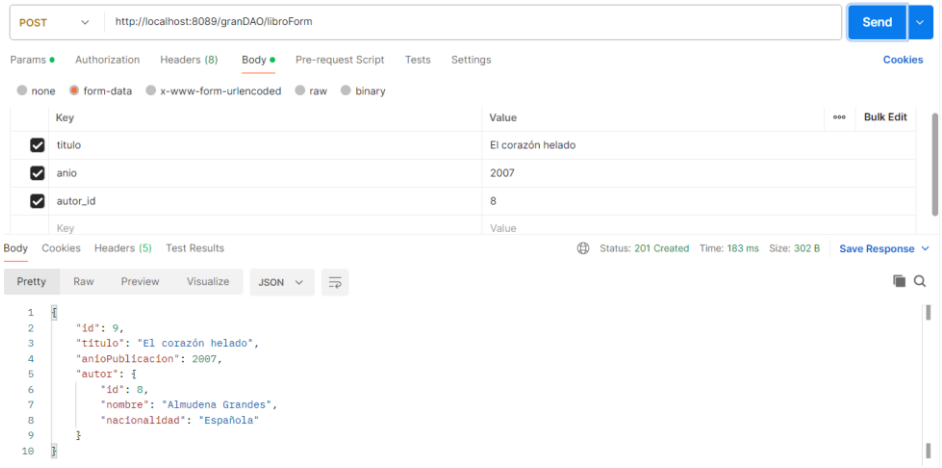
```
//Agrega un libro utilizando parámetros
public Libros addLibroParameters(@Valid String titulo, @Valid Integer anio,@Valid Integer autor_id) {  1 usage   mariaasir *
    Libros libro = new Libros();
    libro.setTitulo(titulo);
    libro.setAnioPublicacion(anio);
    libro.setAutor(getAutorById(autor_id)); // Obtener el libro por ID
    return repositorioLibros.save(libro);
}
```

Código del controlador:

Recibe los parámetros y los valida. Guarda el objeto libro llamando al Service y devuelve el objeto libro con ResponseEntity.

```java
//POST DE UN OBJETO LIBRO POR PARÁMETROS
@PostMapping(value = "/libroForm", consumes = MediaType.MULTIPART_FORM_DATA_VALUE)  mariaasir *
public ResponseEntity<Libros> addLibroForm(
        @Valid @RequestParam String titulo,    //Recibe el título del libro como parámetro del formulario
        @Valid @RequestParam Integer anio,     //Recibe el año de publicacion del libro como parámetro del formulario
        @Valid @RequestParam Integer autor_id  //Recibe el ID del autor del libro como parámetro del formulario
) {
    //Llama al Service para agregar un nuevo libro con los parámetros recibidos del formulario
    return ResponseEntity.created( location: null).body(service.addLibroParameters(titulo, anio, autor_id));   //Devuelve el Objeto del libro creado
}
```

Comprobación en PostMan:



Comprobación de la inserción en la BDD:

☐	Editar Copiar Borrar	1	Cien años de soledad	1967	1
☐	Editar Copiar Borrar	2	El amor en los tiempos del cólera	1985	1
☐	Editar Copiar Borrar	3	Don Quijote de la Mancha	1605	2
☐	Editar Copiar Borrar	4	Harry Potter y la piedra filosofal	1997	3
☐	Editar Copiar Borrar	8	Las edades de Lulú	1989	8
☐	Editar Copiar Borrar	9	El corazón helado	2007	8

1.2.5 UPDATE LIBRO

Código del Service:

Recibe por parámetro el objeto libro, lo valida, y lo guarda en la BDD mediante el repositorio de libros.

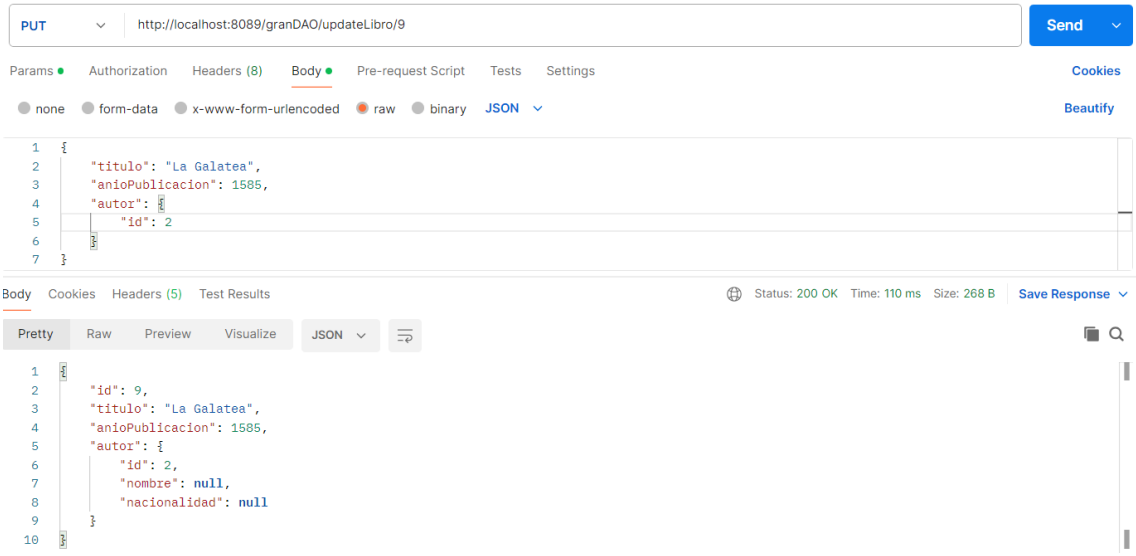
```java
//Actualiza un libro en la base de datos
public Libros updateLibro(@Valid Libros libro) {  1 usage  mariaasir *
    return repositorioLibros.save(libro);
}
```

Código del controlador:

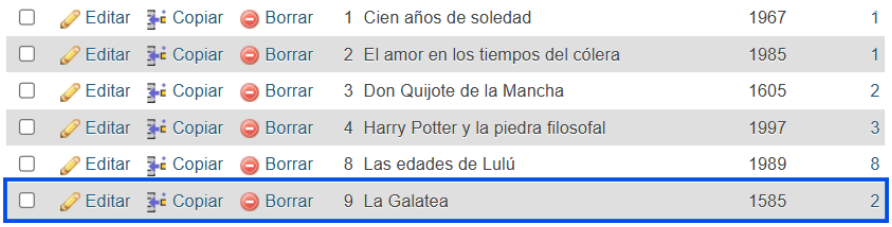
Recibe el ID y el objeto libro y busca si el libro existe. Si no existe, lanza una excepción. Si existe, actualiza sus campos y llama al Service para actualizar el libro. Devuelve el objeto libro mediante ResponseEntity.

```java
//PUT LIBRO --> UPDATE OBJETO LIBRO
@PutMapping("/updateLibro/{id}")  mariaasir *
public ResponseEntity<Libros> updateLibro(@PathVariable int id, @RequestBody @Valid Libros libro) {
    Libros libroExistente = service.getLibroById(id);         //Busca si el libro existe por su ID
    if (libroExistente == null) {
        return ResponseEntity.notFound().build(); // Si el libro no existe, devuelve un error NOT FOUND
    }

    //Actualiza los campos del libro existente con los datos introducidos
    libroExistente.setTitulo(libro.getTitulo());
    libroExistente.setAnioPublicacion(libro.getAnioPublicacion());
    libroExistente.setAutor(libro.getAutor());

    Libros libroActualizado = service.updateLibro(libroExistente);  //Llama al Service para actualizar el libro
    return ResponseEntity.ok().body(libroActualizado); //Devuelve el Objeto del libro actualizado
}
```

Comprobación en PostMan:



Comprobación de la modificación en la BDD:



1.2.6 DELETE LIBRO

Código del Service:

Recibe el ID del libro a eliminar por parámetro, llama al repositorio de libros y elimina el libro de la BDD.

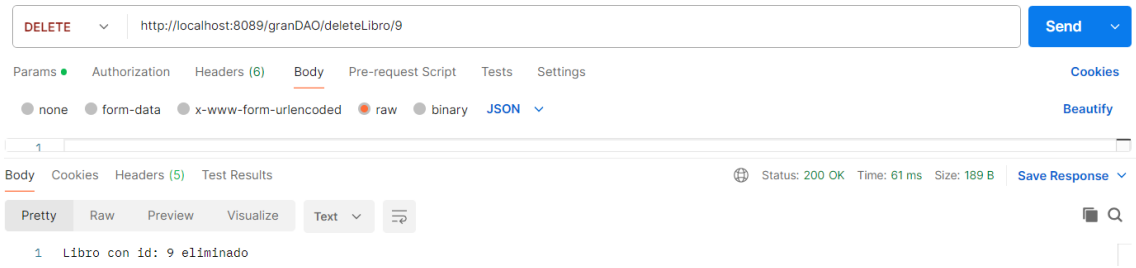
```java
//Elimina un libro de la base de datos por su ID
public void deleteLibro(int id) {  1 usage  mariaasir
    repositorioLibros.deleteById(id);
}
```

Código del controlador:

Recibe el ID por parámetro y llama al Service para eliminar el libro. Muestra un mensaje de que el libro se ha eliminado con éxito.

```java
//DELETE --> DELETE DE UN OBJETO LIBRO
@DeleteMapping("/deleteLibro/{id}")  mariaasir
public ResponseEntity<String> deleteLibro(@PathVariable int id) {
    //Llama al Service para eliminar el libro de la BDD
    service.deleteLibro(id);
    String mensaje = "Libro con id: " + id + " eliminado";  //Muestra un mensaje de que el libro se ha eliminado con éxito
    return ResponseEntity.ok().body(mensaje);
}
```

Comprobación en PostMan:



Comprobación de la eliminación en la BDD:

☐	Editar	Copiar	Borrar	1	Cien años de soledad	1967	1
☐	Editar	Copiar	Borrar	2	El amor en los tiempos del cólera	1985	1
☐	Editar	Copiar	Borrar	3	Don Quijote de la Mancha	1605	2
☐	Editar	Copiar	Borrar	4	Harry Potter y la piedra filosofal	1997	3
☐	Editar	Copiar	Borrar	8	Las edades de Lulú	1989	8

2. BASES DE DATOS NO RELACIONALES NOSQL (MONGODB)

2.1 CIUDADES

2.1.1 GET ALL CIUDADES

Código del Service:

Llama al repositorio de Ciudades y devuelve una lista de todas las ciudades guardadas en la BDD.

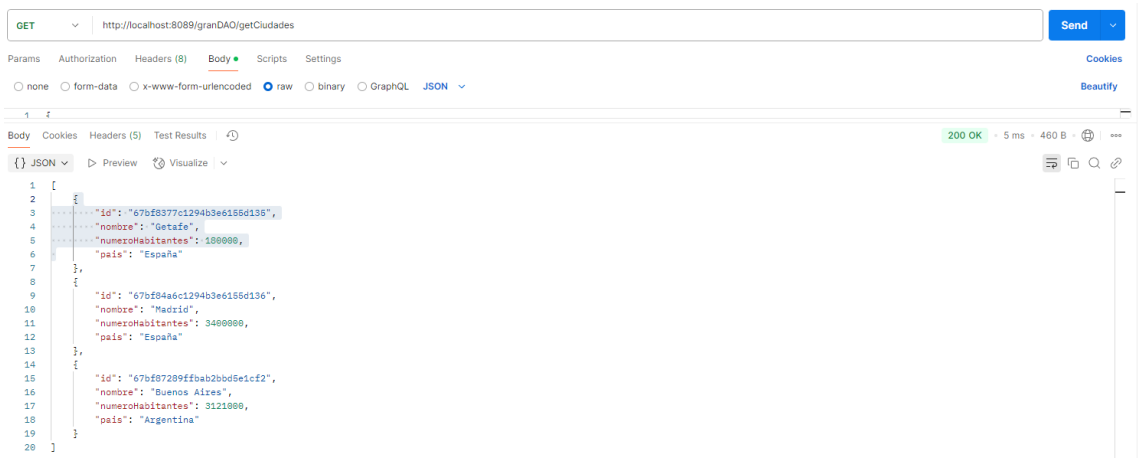
```
//Obtener todas las ciudades
public List<Ciudad> obtenerTodasLasCiudades() {  1 usage   Unai
    return repositorioCiudades.findAll();
}
```

Código del controlador:

Llama al método *obtenerTodasLasCiudades* del Service para obtener todas las ciudades de la BDD.

```
//Obtener todas las ciudades
@GetMapping("/getCiudades")   Unai
public ResponseEntity<List<Ciudad>> obtenerTodasLasCiudades() {
    return ResponseEntity.ok(service.obtenerTodasLasCiudades());
}
```

Comprobación en PostMan:



2.1.2 GET CIUDAD BY ID

Código del Service:

Desde el repositorio busca la ciudad con el ID introducido por parámetro. Si no lo encuentra devuelve NULL.

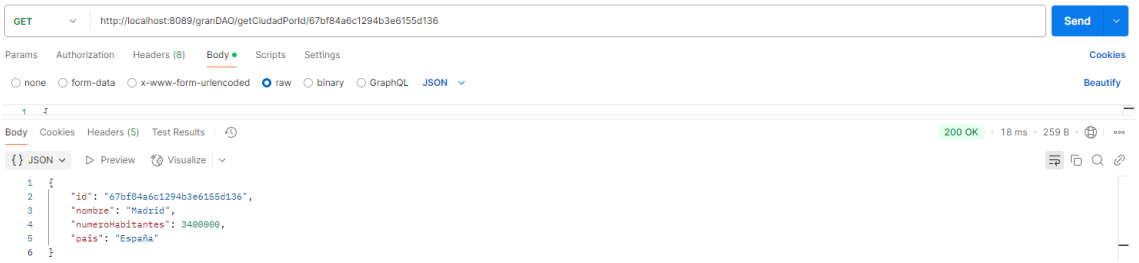
```
// Obtener una ciudad por su ID
public Ciudad obtenerCiudadPorId(String id) {  1 usage   Unai
    return repositorioCiudades.findById(id).orElse( other: null);
}
```

Código del controlador:

Llama al Service para obtener la ciudad por el ID introducido.

```
//Obtener una ciudad por su ID
@GetMapping("/getCiudadPorId/{id}")   Unai
public ResponseEntity<Ciudad> obtenerCiudadPorId(@PathVariable String id) {
    return ResponseEntity.ok(service.obtenerCiudadPorId(id));
}
```

Comprobación en PostMan:



2.1.3 POST CIUDAD

Código del Service:

Recibe un Objeto ciudad por parámetro y lo guarda en la BDD a través del repositorio.

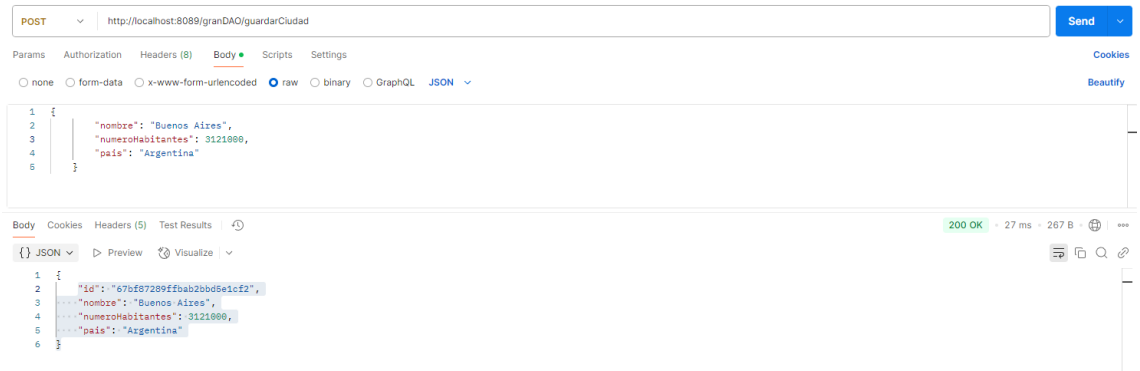
```
//Guardar una ciudad (crear o actualizar)
public Ciudad guardarCiudad(Ciudad ciudad) {  2 usages   Unai
    return repositorioCiudades.save(ciudad);
}
```

Código del controlador:

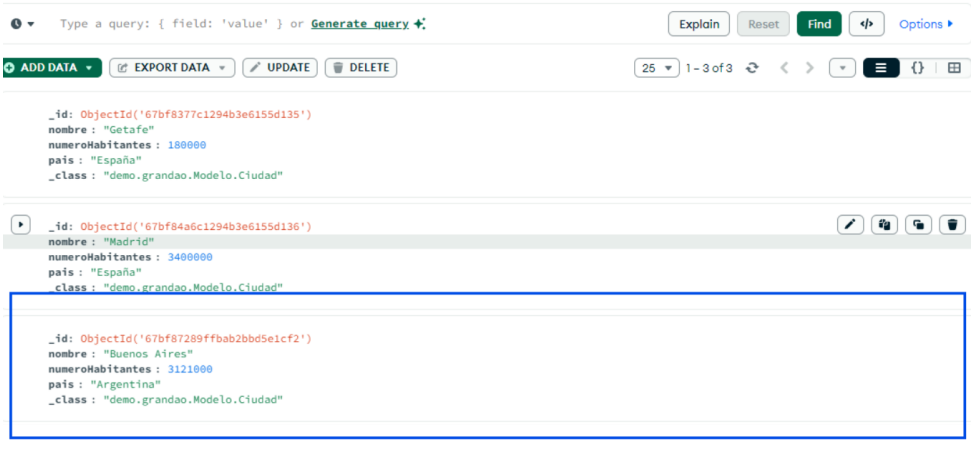
Llama al método *guardarCiudad* del Service.

```
//Crear una nueva ciudad
@PostMapping("/postCiudad")   Unai
public ResponseEntity<Ciudad> crearCiudad(@RequestBody Ciudad ciudad) {
    return ResponseEntity.ok(service.guardarCiudad(ciudad));
}
```

Comprobación en PostMan:



Comprobación de la inserción en la BDD:



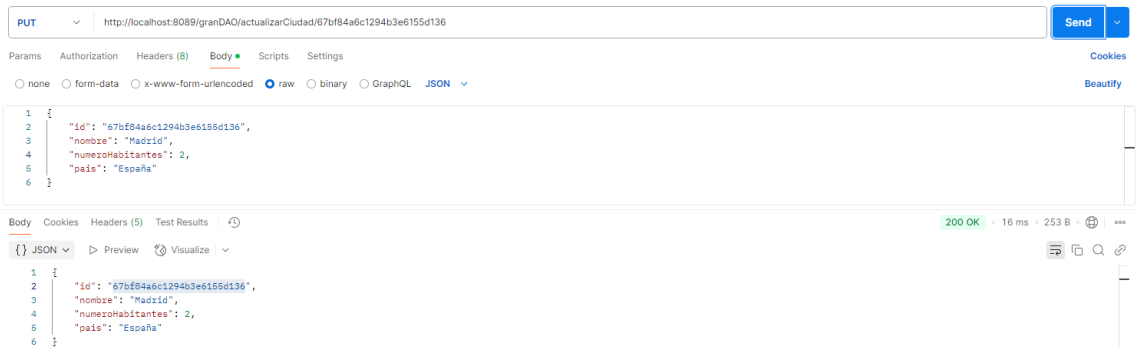
2.1.4 UPDATE CIUDAD

Código del controlador:

Recibe el ID de la ciudad y el JSON de la misma actualizado. Comprueba que hay una ciudad con ese ID y la guarda en la BDD mediante el método *guardarCiudad* del Service.

```
//Actualizar una ciudad existente
@PutMapping("/updateCiudad/{id}")  Unai
public ResponseEntity<Ciudad> actualizarCiudad(@PathVariable String id, @RequestBody Ciudad ciudad) {
    ciudad.setId(id); // Asegurar que el ID coincida
    return ResponseEntity.ok().body(service.guardarCiudad(ciudad));
}
```

Comprobación en PostMan:



Comprobación de la modificación en la BDD:

2.1.5 DELETE CIUDAD

Código del Service:

Recibe el ID de la ciudad a eliminar y la elimina de la BDD mediante el repositorio.

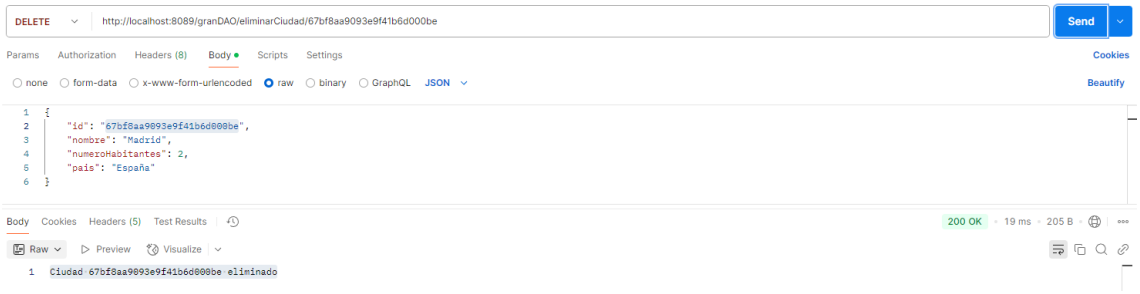
```
//Eliminar una ciudad por su ID
public void eliminarCiudadPorId(String id) {  1 usage  Unai
    repositorioCiudades.deleteById(id);
}
```

Código del controlador:

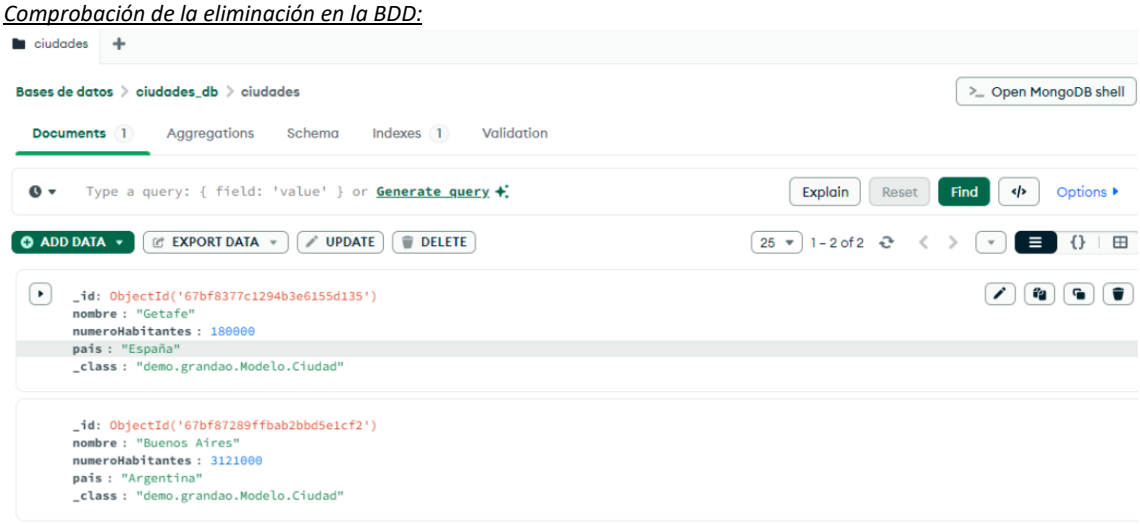
Recibe el ID de la ciudad a eliminar, llama al método *eliminarCiudadPorId* del Service y muestra un mensaje si se ha eliminado correctamente.

```
//Eliminar una ciudad por su ID
@DeleteMapping("/deleteCiudad/{id}")  Unai *
public ResponseEntity<String> eliminarCiudad(@PathVariable String id) {
    service.eliminarCiudadPorId(id);
    return ResponseEntity.ok( body: "Ciudad con ID: " + id + " eliminada");
}
```

Comprobación en PostMan:



Comprobación de la eliminación en la BDD:



3. FICHEROS XML

En este caso, tendremos un fichero XML que se encargará de guardar usuarios con dos parámetros : Nombre y Contraseña.

3.1 USUARIOS

3.1.1 GET USUARIOS

Código del DAO:

Genera una nueva instancia de JAXBContext para leer la lista de usuarios desde la clase UsuariosList. Con Unmarshaller, transforma el XML en objetos. Obtiene los datos de ARCHIVO_XML y los transforma en una instancia de UsuariosList. Devuelve el listado de usuarios.

```
public List<Usuarios> leerUsuarios() throws JAXBException {  3 usages   mariaasir
    JAXBContext context = JAXBContext.newInstance(UsuariosList.class);
    Unmarshaller unmarshaller = context.createUnmarshaller();
    UsuariosList wrapper = (UsuariosList) unmarshaller.unmarshal(new File(ARCHIVO_XML));
    return wrapper.getUsuarios();
}
```

Código del Service:

Llama al DAO para leer todos los usuarios. Si hay error al leer el fichero XML, devuelve una lista vacía.

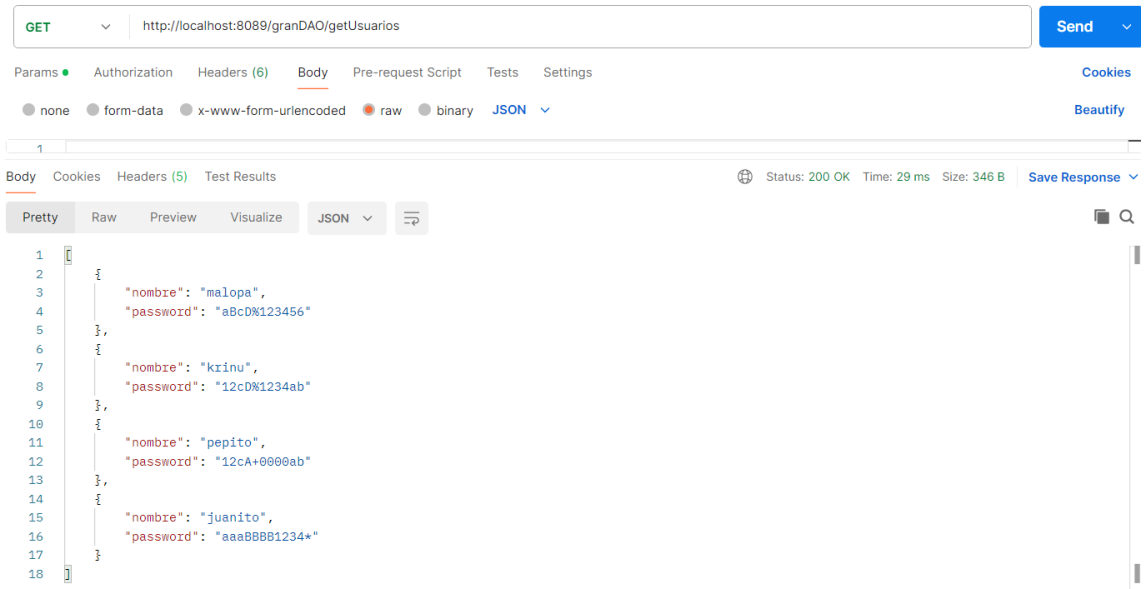
```
public List<Usuarios> getUsuarios() {  2 usages   mariaasir *
    try {
        return repositorioUsuarios.leerUsuarios(); //Llama al método que lee los usuarios desde el XML
    } catch (JAXBException e) {
        e.printStackTrace();
        return new ArrayList<>(); //Devuelve una lista vacía si hay un error
    }
}
```

Código del controlador:

Llama al Service para obtener todos los usuarios del fichero XML.

```
//GET ALL USUARIOS --> SELECT *
@GetMapping("/getUsuarios")   mariaasir
public ResponseEntity<List<Usuarios>> getUsuarios() throws JAXBException {
    //Llama al Service para obtener todos los usuarios desde el fichero XML
    return ResponseEntity.ok(service.getUsuarios()); //Recibe una respuesta con todos los usuarios
}
```

Comprobación en PostMan:



3.1.2 GET USUARIO BY NOMBRE

Código del DAO:

Genera una nueva instancia de JAXBContext para leer la lista de usuarios desde la clase UsuariosList. Con Unmarshaller, transforma el XML en objetos. Obtiene los datos de ARCHIVO_XML y los transforma en una instancia de UsuariosList. Recorre la lista de usuarios y compara el nombre introducido por parámetro con los nombres de la lista. Devuelve el usuario encontrado.

```
public List<Usuarios> leerUsuariosPorNombre(String nombre) throws JAXBException {  1 usage  new *
    JAXBContext context = JAXBContext.newInstance(UsuariosList.class);
    Unmarshaller unmarshaller = context.createUnmarshaller();
    UsuariosList wrapper = (UsuariosList) unmarshaller.unmarshal(new File(ARCHIVO_XML));

    List<Usuarios> usuariosFiltrados = new ArrayList<>();
    for (Usuarios usuario : wrapper.getUsuarios()) {
        if (usuario.getNombre().equalsIgnoreCase(nombre)) {
            usuariosFiltrados.add(usuario);
        }
    }
    return usuariosFiltrados;
}
```

Código del Service:

Llama al DAO para buscar el usuario. Si la lista de usuarios esta vacía devuelve un valor nulo, si no está vacía, devuelve el primer usuario encontrado.

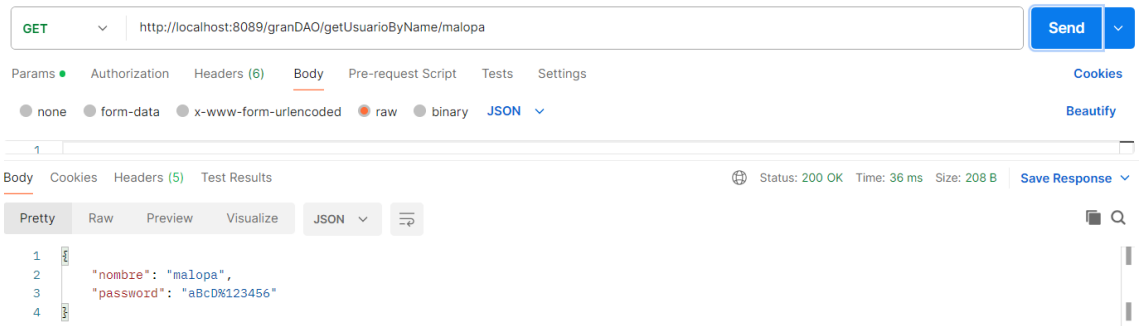
```
//Obtiene un usuario por su nombre desde el archivo XML
public Usuarios getUsuarioByName(String nombre) throws JAXBException {  2 usages  mariaasir *
    List<Usuarios> usuarios = repositorioUsuarios.leerUsuariosPorNombre(nombre);
    return usuarios.isEmpty() ? null : usuarios.get(0);
}
```

Código del controlador:

Llama al Service para encontrar el usuario.

```
//GET USUARIO BY NOMBRE
@GetMapping("/getUsuarioByName/{nombre}")  mariaasir
@Cacheable
public ResponseEntity<Usuarios> getUsuarioByName(@PathVariable String nombre) throws JAXBException {
    //Llama al Service para obtener el usuario por su Nombre
    return ResponseEntity.ok(service.getUsuarioByName(nombre));
}
```

Comprobación en PostMan:



3.1.3 POST USUARIO

Código del DAO:

Genera una nueva instancia de JAXBContext para manejar la lista de usuarios desde la clase UsuariosList. Con Marshaller, transforma la lista a formato XML. A continuación, almacena el XML en ARCHIVO_XML empleando la clase UsuariosList.

```
public void guardarUsuarios(List<Usuarios> usuarios) throws JAXBException {  3 usages  mariaasir *
    JAXBContext context = JAXBContext.newInstance(UsuariosList.class);
    Marshaller marshaller = context.createMarshaller();
    marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
    marshaller.marshal(new UsuariosList(usuarios), new File(ARCHIVO_XML));
}
```

Código del Service:

Obtiene todos los usuarios y los guarda en una lista. Si la lista de usuarios es nula devuelve una lista vacía. Si no es nula, guarda el nuevo usuario en la lista. Guarda la lista de usuarios en el fichero XML mediante el método *guardarUsuarios* de el DAO.

```
public void guardarUsuario(Usuarios usuario) throws JAXBException {  1 usage  mariaasir *
    List<Usuarios> usuarios = getUsuarios();  // Obtiene los usuarios existentes

    //Si la lista de usuarios es null, inicializa una lista vacía
    if (usuarios == null) {
        usuarios = new ArrayList<>();
    }
    usuarios.add(usuario);  //Agrega el nuevo usuario a la lista
    repositorioUsuarios.guardarUsuarios(usuarios);

}
```

Código del controlador:

Llama al Service para guardar el Objeto usuario recibido por parámetro.

```
//POST DE UN OBJETO USUARIO
@PostMapping("/postUsuario")  mariaasir *
public ResponseEntity<Usuarios> postUsuario(@RequestBody @Valid Usuarios usuario) throws JAXBException {
    //Llama al Service para agregar un nuevo usuario al fichero XML
    service.guardarUsuario(usuario);
    return ResponseEntity.ok(usuario);  //Devuelve el Objeto del usuario creado
}
```

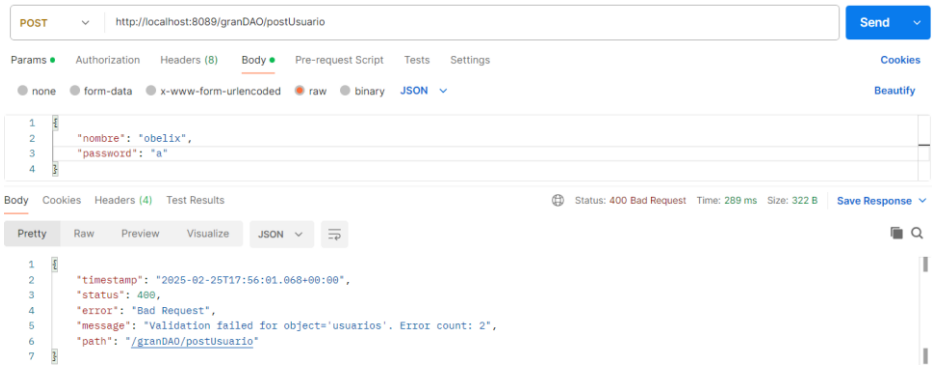
Validaciones:

Para añadir un usuario tendremos que tener en cuenta una serie de validaciones. El nombre del usuario no puede estar vacío. La contraseña debe tener entre 8 y 20 caracteres y debe contener al menos 1 mayúscula, 1 minúscula, 1 número y un carácter especial. También valida que no haya ningún usuario con el mismo nombre.

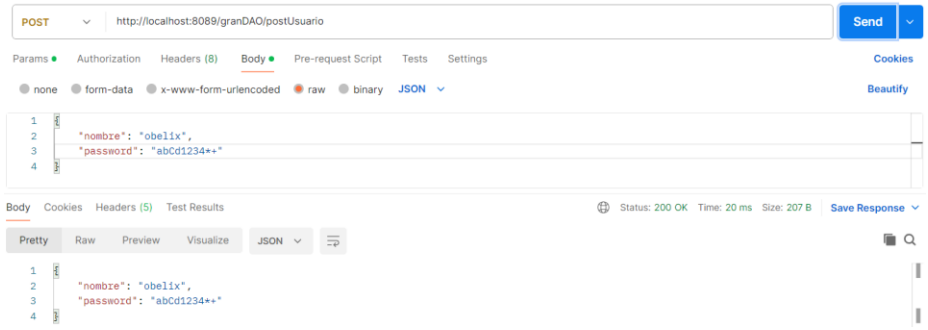
```
@XmlRootElement  27 usages  mariaasir *
@Table(name = "usuarios", uniqueConstraints = @UniqueConstraint(columnNames = "nombre"))
public class Usuarios {

    @NotBlank(message = "El nombre no puede estar vacío")  3 usages
    @Column(name = "nombre", unique = true, nullable = false)
    private String nombre;

    @NotBlank(message = "La contraseña no puede estar vacía")  3 usages
    @Size(min = 8, max = 20, message = "La contraseña debe tener entre 8 y 20 caracteres")
    @Pattern(
            regexp = "^(?=.*[A-Z])(?=.*[a-z])(?=.*\\d)(?=.*[@#$%^&+=!*]).{8,20}$",
            message = "La contraseña debe contener al menos una mayúscula, una minúscula, un número y un carácter especial (@#$%^&+=!*)"
    )
    @Column(name = "password", nullable = false)
    private String password;
```

Comprobación en PostMan:

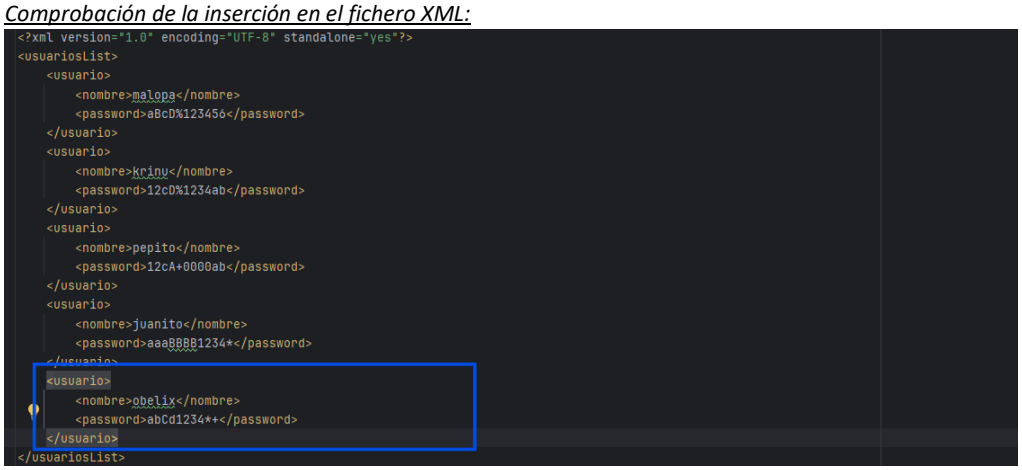
Primero realizaremos un post en el que la contraseña no tenga 8 caracteres o más y veremos que nos muestra un error, comprobando así que la validación al hacer insertar un Libro funciona correctamente.



Ahora realizaremos un post con todos los datos correctos.



Comprobación de la inserción en el fichero XML:



3.1.4 UPDATE USUARIO

Código del DAO:

Recibe el nombre del usuario a actualizar, y el objeto Usuario. Lee la lista de usuarios y lo guarda en una nueva lista. Recorre la lista y comprueba que alguno de los usuarios guardados coincide con el nombre introducido. Si coincide, actualiza los datos del usuario y con el método guardarUsuarios guarda la lista actualizada.

```
//Actualizar un usuario en el XML
public void updateUsuario(String nombre, Usuarios usuarioNuevo) throws JAXBException {  1 usage   mariaasir *
    List<Usuarios> usuarios = leerUsuarios();

    for (Usuarios usuario : usuarios) {
        if (usuario.getNombre().equals(nombre)) {
            usuario.setNombre(usuarioNuevo.getNombre());
            usuario.setPassword(usuarioNuevo.getPassword());
            guardarUsuarios(usuarios); //Guarda la lista actualizada
            return;
        }
    }
}
```

Código del Service:

Recibe el nombre por parámetro, y el nuevo usuario. Llama al DAO para actualizar el usuario.

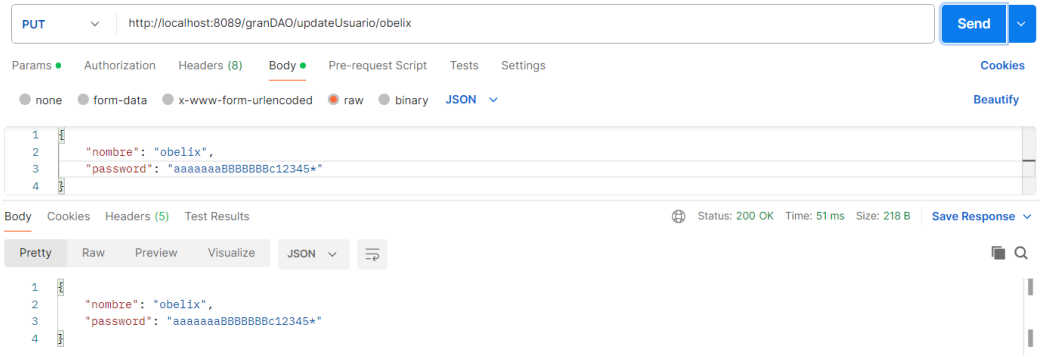
```
//Actualiza los datos de un usuario en el archivo XML
public void actualizarUsuario(String nombre, Usuarios usuarioNuevo) throws JAXBException {  1 usage   mariaasir
    repositorioUsuarios.updateUsuario(nombre, usuarioNuevo);
}
```

Código del controlador:

Recibe el nombre de usuario por parámetro. Busca el usuario por su nombre y si existe, lo guarda en un nuevo usuario llamado usuarioExistente. Si es nulo, devuelve un error. Si el usuario existe, modifica sus parámetros y llama al Service para actualizarlo. Devuelve el objeto usuario actualizado.

```
//PUT USUARIO --> UPDATE OBJETO USUARIO
@PutMapping("/updateUsuario/{nombre}")  mariaasir *
public ResponseEntity<Usuarios> actualizarUsuario(@PathVariable String nombre, @RequestBody @Valid Usuarios usuario) throws JAXBException {
    Usuarios usuarioExistente = service.getUsuarioByName(nombre);   //Busca si el usuario existe por su nombre
    if (usuarioExistente == null) {
        return ResponseEntity.notFound().build();           // Si el usuario no existe, devuelve un error NOT FOUND
    }

    //Actualiza los campos del usuario existente con los datos introducidos
    usuarioExistente.setNombre(usuario.getNombre());
    usuarioExistente.setPassword(usuario.getPassword());

    service.actualizarUsuario(nombre, usuario); //Llama al Service para actualizar el usuario
    return ResponseEntity.ok(usuario); //Devuelve el Objeto del usuario actualizado

}
```

Comprobación en PostMan:



Comprobación de la modificación en el fichero XML:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<usuariosList>
    <usuario>
        <nombre>malopa</nombre>
        <password>aBcD%123456</password>
    </usuario>
    <usuario>
        <nombre>krinu</nombre>
        <password>12cD%1234ab</password>
    </usuario>
    <usuario>
        <nombre>pepito</nombre>
        <password>12cA+0000ab</password>
    </usuario>
    <usuario>
        <nombre>juanito</nombre>
        <password>aaaBBBB1234*</password>
    </usuario>
    <usuario>
        <nombre>obelix</nombre>
        <password>aaaaaaaBBBBBBBc12345*</password>
    </usuario>
</usuariosList>
```

3.1.5 DELETE USUARIO

Código del DAO:

Recibe el nombre del usuario a eliminar. Lee los usuarios y los guarda en una nueva lista. Comprueba que haya algún usuario con ese nombre y si lo hay, lo elimina. Una vez hecho esto, guarda la lista de usuarios actualizada.

```java
public void deleteUsuario(String nombre) throws JAXBException {  1 usage   mariaasir *
    List<Usuarios> usuarios = leerUsuarios();
    usuarios.removeIf(usuario -> usuario.getNombre().equals(nombre));
    guardarUsuarios(usuarios); //Guarda la lista actualizada
}
```

Código del Service:

Llama al DAO de usuarios para eliminar el usuario.

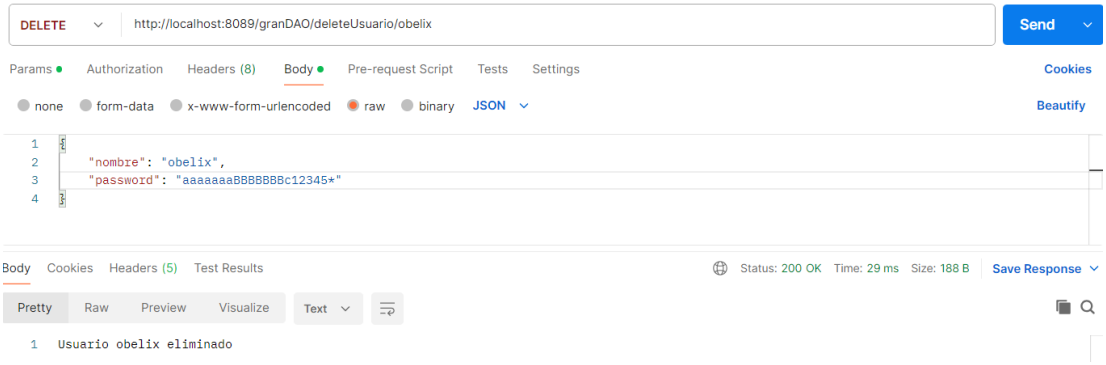
```java
// Eliminar un usuario
public void deleteUsuario(String nombre) throws JAXBException {  1 usage   mariaasir
    repositorioUsuarios.deleteUsuario(nombre);
}
```

Código del controlador:

Recibe por parámetro el nombre del usuario. Llama al Service para eliminar el usuario. Si se ha eliminado con éxito, devuelve un mensaje de que el usuario ha sido eliminado.

```java
//DELETE --> DELETE DE UN OBJETO USUARIO
@DeleteMapping("/deleteUsuario/{nombre}")  mariaasir
public ResponseEntity<String> deleteUsuario(@PathVariable String nombre) throws JAXBException {
    //Llama al Service para eliminar el usuario del fichero XML
    service.deleteUsuario(nombre);
    String mensaje = "Usuario " + nombre + " eliminado"; //Muestra un mensaje de que el usuario se ha eliminado con exito
    return ResponseEntity.ok().body(mensaje);
}
```

Comprobación en PostMan:



Comprobación de la eliminación en el fichero XML:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<usuariosList>
    <usuario>
        <nombre>malopa</nombre>
        <password>aBcD%123456</password>
    </usuario>
    <usuario>
        <nombre>krinu</nombre>
        <password>12cD%1234ab</password>
    </usuario>
    <usuario>
        <nombre>pepito</nombre>
        <password>12cA+0000ab</password>
    </usuario>
    <usuario>
        <nombre>juanito</nombre>
        <password>aaaBBBB1234*</password>
    </usuario>
</usuariosList>
```

4. FICHEROS Y DIRECTORIOS

4.1 COCHES

4.1.1 GET ALL COCHES

Código del DAO:

Crea un archivo en la ruta indicada. Si el archivo no existe o está vacío , devuelve una lista vacía. Si el archivo existe, introduce la lista de coches dentro del archivo.

```java
// Leer la lista de coches desde el fichero
public List<Coche> leerCoches() throws IOException, ClassNotFoundException {  3 usages  Unai
    File file = new File(FILE_PATH);
    if (!file.exists() || file.length() == 0) {
        return new ArrayList<>(); // Si el archivo no existe o está vacío, devuelve una lista vacía
    }
    try (FileInputStream fileIn = new FileInputStream(file);
         ObjectInputStream objectIn = new ObjectInputStream(fileIn)) {
        return (List<Coche>) objectIn.readObject();
    }
}
```

Código del Service:

Llama al método de *leerCoches* del DAO de coches.

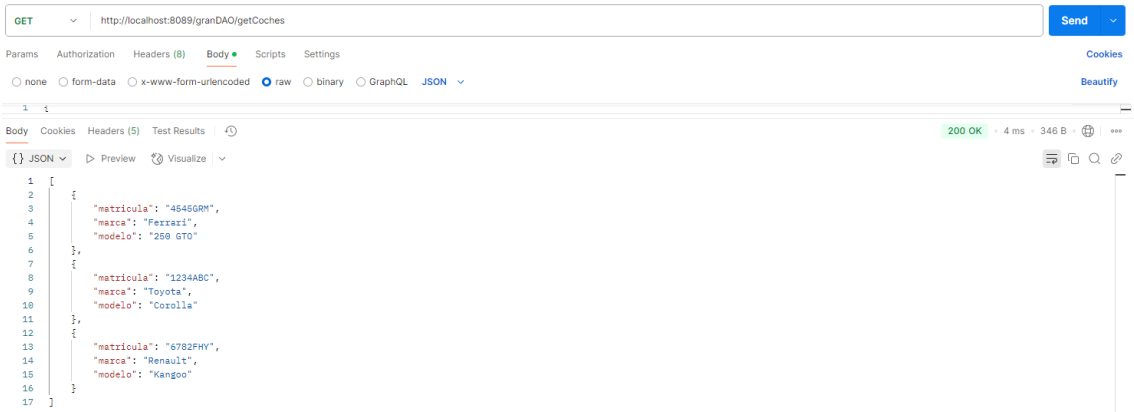
```
// Leer la lista de coches desde el fichero
public List<Coche> obtenerCoches() throws IOException, ClassNotFoundException {  1 usage   Unai
    return repositorioCoches.leerCoches();
}
```

Código del controlador:

Crea una lista de coches y recibe todos los coches desde el Service. Devuelve la lista de los coches con ResponseEntity.

```
// Endpoint para leer todos los coches
@GetMapping("/getCoches")   Unai
public ResponseEntity<List<Coche>> obtenerCoches() throws IOException, ClassNotFoundException {
    List<Coche> coches = service.obtenerCoches();
    return ResponseEntity.ok(coches);
}
```

Comprobación en PostMan:



4.1.2 GET COCHE BY MATRÍCULA

Código del DAO:

Recibe la matrícula de un coche. Crea una lista de coches, con todos los coches que hay en la BDD. Recorre la lista de los coches y busca el coche que contenga esa matrícula. Una vez encontrado, devuelve el Objeto Coche.

```
//Buscar un coche por matricula
public Coche buscarCochePorMatricula(String matricula) throws IOException, ClassNotFoundException {  1 usage   new *
    List<Coche> coches = leerCoches(); //Lee la lista de coches
    Coche cocheEncontrado = new Coche();
    for (Coche coche : coches) {
        if (coche.getMatricula().equals(matricula)) {
            cocheEncontrado = coche;
        }
    }
    return cocheEncontrado;
}
```

Código del Service:

Recibe la matrícula del coche a buscar. Llama al método *buscarCochePorMatricula* de la clase DAO de coches.

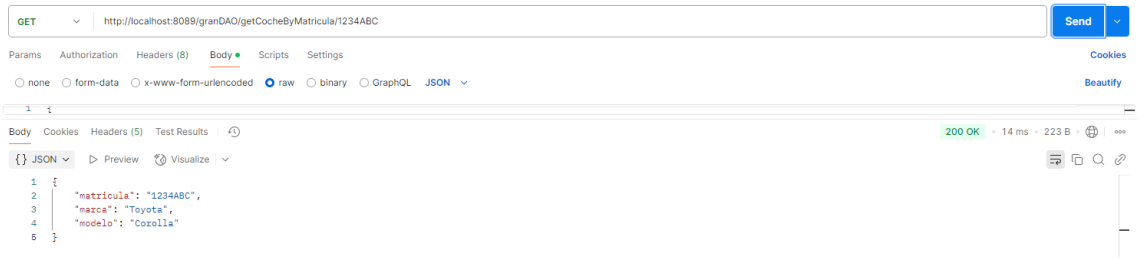
```
// Buscar un coche por matrícula
public Coche buscarCochePorMatricula(String matricula) throws IOException, ClassNotFoundException {  1 usage   Unai *
    return repositorioCoches.buscarCochePorMatricula(matricula);
}
```

Código del controlador:

Recibe la matrícula por parámetro. Llama al Service para buscar un coche por su matrícula.

```
//Endpoint para leer un coche por matricula
@GetMapping("/getCocheByMatricula/{matricula}")  Unai *
public ResponseEntity<Coche> obtenerCochePorMatricula(@PathVariable String matricula) throws IOException, ClassNotFoundException {
    return ResponseEntity.ok(service.buscarCochePorMatricula(matricula));
}
```

Comprobación en PostMan:



4.1.3 POST COCHE

Código del DAO:

Encontramos dos métodos:

guardarCoches: Recibe una lista de coches y la guarda en el fichero mediante el writeObject, eliminando el contenido anterior del fichero.

insertarCoche: Crea una lista de coches y guarda todos los coches que ya estaban en el fichero. Además, añade el coche nuevo a la lista. Por último, llama al método *guardarCoches* para guardar la lista actualizada con el nuevo coche.

```
//Guardar una lista de coches en el fichero
public void guardarCoches(List<Coche> coches) throws IOException {  1 usage  Unai
    try (FileOutputStream fileOut = new FileOutputStream(FILE_PATH);
         ObjectOutputStream objectOut = new ObjectOutputStream(fileOut)) {
        objectOut.writeObject(coches); // Escribe la lista en el archivo
    }
}

//Añadir un coche a la lista existente
public void insertarCoche(Coche coche) throws IOException, ClassNotFoundException {  1 usage  new *
    List<Coche> coches = leerCoches(); //Lee la lista existente
    coches.add(coche); //Añade el nuevo coche
    guardarCoches(coches); //Guarda la lista actualizada
}
```

Código del Service:

Llama al DAO para insertar el coche.

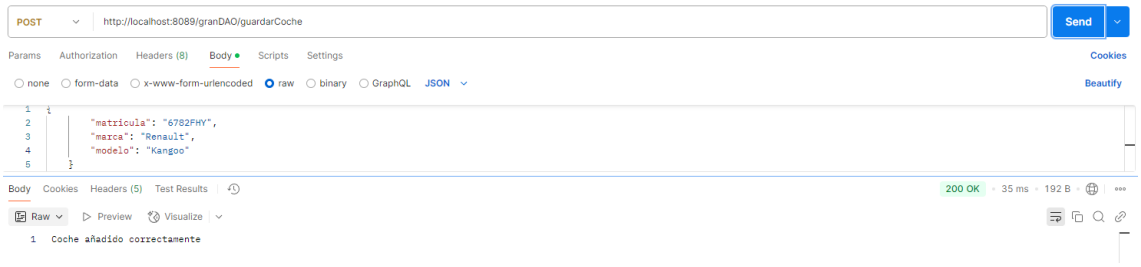
```
// Añadir un coche a la lista existente
public void insertarCoche(Coche coche) throws IOException, ClassNotFoundException {  1 usage  Unai
    repositorioCoches.insertarCoche(coche);
}
```

Código del controlador:

Llama al Service para guardar el nuevo coche y muestra un mensaje de que el coche se ha insertado correctamente.

```
// Endpoint para guardar un nuevo coche
@PostMapping("/guardarCoche")  Unai
public ResponseEntity<String> guardarCoche(@RequestBody Coche coche) throws IOException, ClassNotFoundException {
    service.insertarCoche(coche);
    return ResponseEntity.ok( body: "Coche añadido correctamente");
}
```

Comprobación en PostMan:



Comprobación en el fichero DAT: