

# Taller Spring API Rest con JdbcTemplate

## 1.- Crear un proyecto nuevo con **spring inicializr**

- Spring.io.inicializr



**Project**  
☐ Gradle - Groovy ☐ Gradle - Kotlin ☒ **Maven**

**Language**  
☒ **Java** ☐ Kotlin ☐ Groovy

**Spring Boot**  
☐ 3.4.0 (SNAPSHOT) ☐ 3.4.0 (M3) ☐ 3.3.5 (SNAPSHOT) ☒ **3.3.4**  
☐ 3.2.11 (SNAPSHOT) ☐ 3.2.10

**Project Metadata**  
Group   
Artifact   
Name   
Description   
Package name   
Packaging ☒ **Jar** ☐ War  
Java ☐ 23 ☐ 21 ☒ **17**

**Dependencies** ADD DEPENDENCIES... CTRL + B  
*No dependency selected*

## Añadir dependencias

**Dependencies** ADD DEPENDENCIES... CTRL + B

**Spring Web** **WEB**  
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

**Spring Data JDBC** **SQL**  
Persist data in SQL stores with plain JDBC using Spring Data.

**Spring Boot DevTools** **DEVELOPER TOOLS**  
Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

**MySQL Driver** **SQL**  
MySQL JDBC driver.

Generar proyecto:

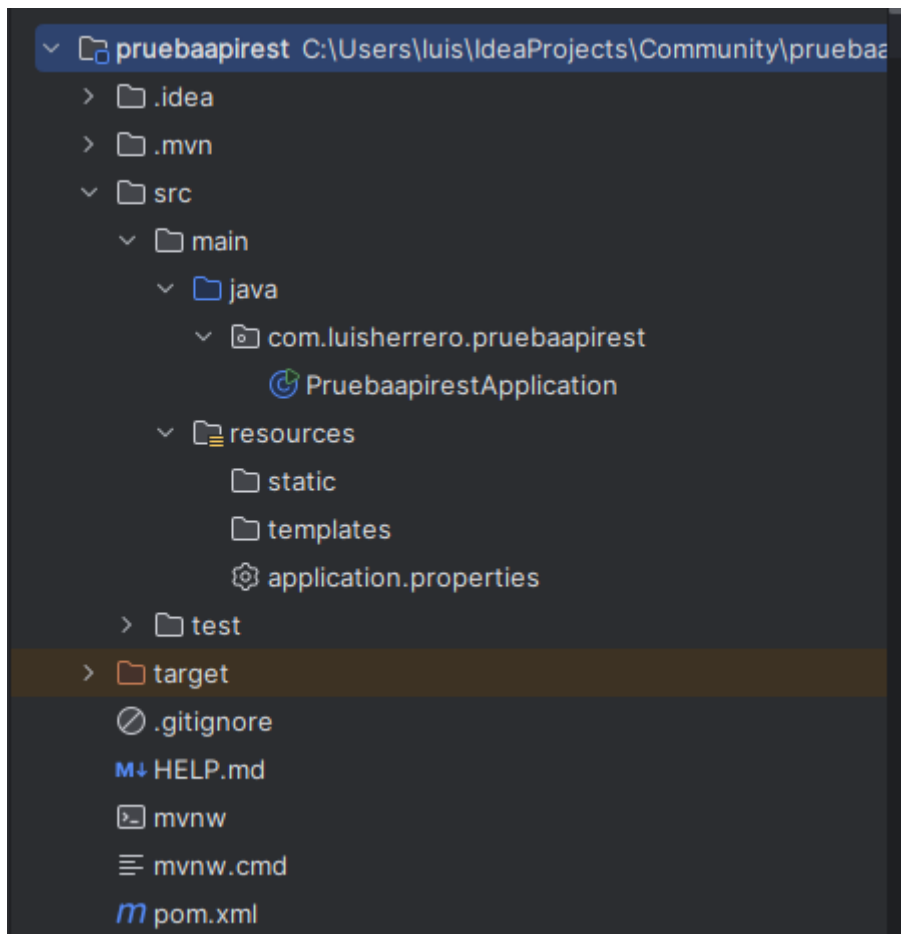
GENERATE CTRL + ⌘

EXPLORE CTRL + SPACE

SHARE...

Una vez generado el proyecto, tenemos en descargas un archivo ZIP que deberemos descomprimir y abrir desde IntelliJ Idea.

Tras abrir el proyecto, veremos que la estructura es la siguiente:



La clase **PruebaapiRESTApplication** contiene el método main que se encargará de lanzar a ejecución el servicio.

El fichero **application.properties** permite configurar propiedades del proyecto Spring, entre otras cosas la conexión a la base de datos (el datasource). Para configurar una conexión a una base de datos **mysql**, estas serían las propiedades que habría que configurar (dando los valores adecuados a la conexión que queramos hacer):

```
spring.application.name=pruebaapiREST
spring.datasource.url=jdbc:mysql://localhost:3306/una_BD
```

```
spring.datasource.username=mi_user_mysql
spring.datasource.password=mi_password_mysql
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

Ahora, ya podemos ejecutar el proyecto. Si todo ha ido bien, estos serán los mensajes de Log que se devuelven tras haber inicializado el servicio:

```
[pruebaapires] [ restartedMain] c.l.p.PruebaapiresApplication : Starting PruebaapiresApplication using Java 22.0.1 with
[pruebaapires] [ restartedMain] c.l.p.PruebaapiresApplication : No active profile set, falling back to 1 default profile:
[pruebaapires] [ restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : Devtools property defaults active! Set 'spring.devtools.a
[pruebaapires] [ restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : For additional web related logging consider setting the '
[pruebaapires] [ restartedMain] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JDBC repositories in DEFAULT mo
[pruebaapires] [ restartedMain] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 28 ms. Found
[pruebaapires] [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port 8080 (http)
[pruebaapires] [ restartedMain] o.apache.catalina.core.StandardService : Starting service [Tomcat]
[pruebaapires] [ restartedMain] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.30]
[pruebaapires] [ restartedMain] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
[pruebaapires] [ restartedMain] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 3
[pruebaapires] [ restartedMain] com.zaxxer.hikari.HikariApplicationContext : HikariPool-1 - Starting...
[pruebaapires] [ restartedMain] com.zaxxer.hikari.pool.HikariPool : HikariPool-1 - Added connection com.mysql.cj.jdbc.Connect
[pruebaapires] [ restartedMain] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.
[pruebaapires] [ restartedMain] o.s.b.d.a.OptionalLiveReloadServer : LiveReload server is running on port 35729
[pruebaapires] [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path '/'
[pruebaapires] [ restartedMain] c.l.p.PruebaapiresApplication : Started PruebaapiresApplication in 6.619 seconds (proces
```

En una de las líneas se indica que se ha iniciado para el proyecto un **servidor de aplicaciones Tomcat en el puerto 8080**. Podremos así iniciar conexiones **http** por el puerto **8080** para que se pueda consumir el servicio de la API Rest (cuando la implementemos).

3.- Vamos a implementar algo muy básico en la API Rest:

Que al hacerle una **petición** desde el navegador (cliente), responda con un saludo.

Los métodos de la API que se comunican con los clientes por http son métodos de **controlador o controller**. Por eso, para tener una buena organización de proyecto, deben colocarse en una clase dentro de un package de nombre **controller** (aunque podría tener cualquier otro nombre). Creamos por tanto un package controller y, dentro de él, una clase de nombre **PruebaController**.

Para que **PruebaController** se comporte como clase con métodos controlador, hay que anotarla con la anotación **RestController** de Spring.

```
@RestController no usages
public class PruebaController {
}
```

Ahora, incluiremos dentro de la clase este método, con las anotaciones que se indican:

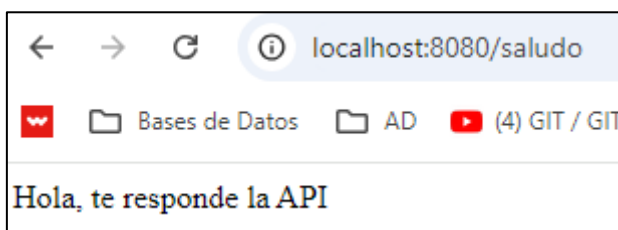
```
@GetMapping("/saludo") no usages
public String pruebaSaludo1() {
    return "Hola, te responde la API ";
}
```

La anotación **@GetMapping("/saludo")** hace que la API responderá a peticiones http desde clientes. La URL de la petición deberá ser la URL base de la API (en nuestro caso, por ahora **http://equipo\_de\_la\_api:8080** , seguido de **/saludo**.

- El motivo de esa URL es que hemos definido un **endpoint** de valor **/saludo**.
- Lo devuelto por **return** será lo que se envía al cliente.

Ahora reiniciamos el proyecto, con lo que lanzamos a ejecución el servicio con las nuevas condiciones.

Para probar, ejecutamos el navegador (Chrome por ejemplo) y en URL de navegación escribimos **http://localhost:8080/saludo**. Veremos la respuesta en la pantalla del navegador es:



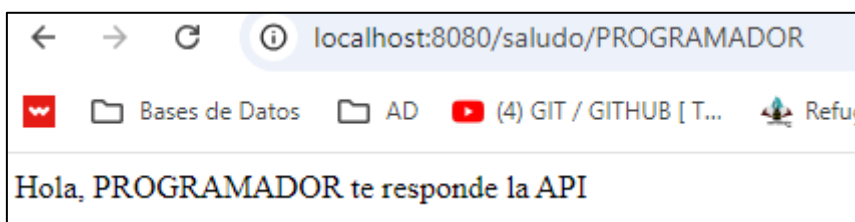
Ahora, vamos a incluir otro método controlador. En este caso, desde el cliente se va a enviar un valor en un parámetro dentro de la petición y en el controlador se va a usar ese parámetro.

```
@GetMapping("/saludo/{nom}") no usages
public String pruebaSaludo2(@PathVariable String nom) {
    return "Hola, "+nom+" te responde la API ";
}
```

Ahora, el **endpoint** es **/saludo/{nom}**. Para hacer peticiones a este método el cliente tendrá que añadir un valor después de **saludo/**.

Al poner en el **endpoint** algo entre llaves, significa que es un valor recibido en un parámetro (en este caso, en nom). En el método, se declara ahora un parámetro de tipo String con el mismo nombre que el declarado en el **endpoint**. Este parámetro se debe anotar con **@PathVariable** para indicar que se recibe en la petición GET http y se corresponde con {nom}.

Reiniciamos la API y probamos en el navegador.



#### 4.- Desarrollo de la API Rest para acceder a la base de datos concursomusica

Lo realizado hasta ahora simplemente ha tenido como objetivo entender como un cliente consume un servicio API Rest. Pero una API Rest trabaja normalmente con información almacenada en bases de datos. En nuestro caso, usaremos la base de datos **conkursomusica** de **MySQL**.

En primer lugar, en el proyecto estableceremos las propiedades adecuadas en **application.properties**.

Además de las propiedades que hemos puesto anteriormente, para que lo conozcamos, vamos a incluir estas dos propiedades:

```
# Puerto en el que se ejecutará la aplicación
server.port=4000
# URL base o prefijo para los endpoints
server.servlet.context-path=/
```

Aunque ya están comentadas, cuando probemos mas adelante, veremos que efecto tienen.

En el proyecto, para tenerlo bien estructurado, deberíamos tener cuatro paquetes al menos:

- **model**
- **repository**
- **service**
- **controller**

En el package **model** se tienen las clases que modelan las tablas de la base de datos. Añadimos la clase **Grupo** que debe tener los atributos que mapean las columnas de la tabla **grupos**.

```
private int codGrupo;
private String nombre;
private String localidad;
private String estilo;
private boolean esGrupo;
private Integer annoGrab;
private LocalDate fechaEstreno;
private String compania;
```

Y añadimos los **getter/setter**.

Si quisiéramos trabajar con más tablas de la BD, tendríamos que añadir las clases de modelo correspondientes.

En el package **repository**, vamos primero a incluir un interface con una definición de métodos que realizarán funcionalidades básicas con la tabla Grupos. El interface se debería llamar **GrupoRepository**:

```
public interface GrupoRepository { no usages
    // Método para obtener todos los grupos
    List<Grupo> findAll(); no usages
    // Método para obtener un grupo por su ID
    Grupo findById(int id); no usages
    // Método para obtener grupos de una localidad
    Grupo findByLocalidad(String loc); no usages
    // Método para guardar un nuevo grupo
    int save(Gruo grupo); no usages
    // Método para actualizar un grupo existente
    int update(Gruo grupo); no usages
    // Método para eliminar un grupo por su ID
    int deleteById(int id); no usages
}
```

Ahora, también en **repository**, creamos una clase **GrupoRepositoryImpl** que implementa el anterior interface. La clase debe estar anotada con **@Repository**.

```
@Repository
public class GrupoRepositoryImpl implements GrupoRepository {
}
```

Implementamos los métodos del interface. Declaramos un objeto miembro de tipo **JdbcTemplate** en la clase y un constructor para iniciar ese **JdbcTemplate**.

```
import java.util.List;
@Repository no usages
public class GrupoRepositoryImpl implements GrupoRepository{

    private final JdbcTemplate jdbcTemplate; 2 usages

    public GrupoRepositoryImpl(JdbcTemplate jdbcTemplate) { no usages
        this.jdbcTemplate = jdbcTemplate;
    }

    @Override no usages
    public List<Grupo> findAll() {
        return List.of();
    }
}
```

Con **jdbcTemplate** podremos enviar instrucciones SQL al SGBD, en este caso a MySQL.

Vamos a ver en primer lugar como debe ser el código del método **save** para almacenar un Grupo en la base de datos.

```
@Override no usages
public int save(Gruo grupo) {
    String sql="INSERT INTO grupos (nombre, localidad, estilo, esgrupo, annoGrab, fechaEstreno, compania) " +
        "VALUES (?, ?, ?, ?, ?, ?, ?)";
    return jdbcTemplate.update(sql,
        grupo.getNombre(), grupo.getLocalidad(), grupo.getEstilo(), grupo.isEsGrupo(),
        grupo.getAnnoGrab(), grupo.getFechaEstreno(), grupo.getCompania());
}
```

Vemos que el método recibe un objeto grupo. Con los datos del grupo, se crea una SQL parametrizada. El método **update** de **JdbcTemplate** permite enviar a servidor instrucciones de actualización de datos. Al método se le pasa siempre una consulta parametrizada y a continuación los datos sustituibles en la instrucción parametrizada. El método devuelve cuantas filas se han actualizado al ejecutar la instrucción.

Ahora vamos a ver cómo se puede codificar el método **findAll()**. Es tan simple como esto:

```
@Override no usages
public List<Grupo> findAll() {
    return jdbcTemplate.query("SELECT * FROM grupos", new GrupoRowMapper());
}
```

Como vemos, se ejecuta una **query** a la que se pasa una consulta SELECT (String) y un objeto **GrupoRowMapper**. El método **query** devuelve una lista de objetos, en este caso de objetos Grupo. La clase **GrupoRowMapper** tenemos que desarrollarla. Es una clase de tipo **RowMapper** que sirve para especificar como se deben procesar o mapear los resultados de una consulta de grupos (al igual que cuando procesamos un **ResultSet**). Esta clase podemos hacerla interna dentro de la clase **GrupoRepositoryImpl**:

```
private static class GrupoRowMapper implements RowMapper<Grupo> { 1 usage
    @Override no usages
    public Grupo mapRow(ResultSet rs, int rowNum) throws SQLException {
        Grupo grupo = new Grupo();
        grupo.setCodGrupo(rs.getInt( columnLabel: "codgrupo"));
        grupo.setNombre(rs.getString( columnLabel: "nombre"));
        grupo.setLocalidad(rs.getString( columnLabel: "localidad"));
        grupo.setEstilo(rs.getString( columnLabel: "estilo"));
        grupo.setEsGrupo(rs.getBoolean( columnLabel: "esgrupo"));
        grupo.setAnnoGrab(rs.getInt( columnLabel: "annoGrab"));

        // Convertir la columna DATE a LocalDate
        LocalDate fechaEstreno = rs.getDate( columnLabel: "fechaEstreno") != null
            ? rs.getDate( columnLabel: "fechaEstreno").toLocalDate() : null;
        grupo.setFechaEstreno(fechaEstreno);

        grupo.setCompania(rs.getString( columnLabel: "compania"));
        return grupo;
    }
}
```

Como vemos:

- **GrupoRowMapper** implementa el interface **RowMapper** aplicado sobre objetos de la clase **Grupo**.
- Al implementar **RowMapper** obliga a implementar el método **mapRow**.
- Cuando **JdbcTemplate** envía consultas SELECT de grupo usando este **RowMapper**, el método va a recibir, por cada fila, el **ResultSet** y el **número de fila** a procesar.
- Dentro del código de **mapRow** se crea un objeto **Grupo** con los datos del **ResultSet**.
- El método devuelve un objeto **Grupo** correspondiente a cada fila procesada de un **ResultSet**.

Ahora que ya tenemos dos métodos de repositorio, vamos a desarrollar dos métodos de controlador que los usen. En el package **controller** añadimos una clase **GrupoController**.

Nos saltamos **la capa de Servicio** para simplificar. La capa de Servicio es la que implementa la lógica de negocio. Dada la simplicidad de nuestra API, en esta capa tendríamos métodos absolutamente iguales a los de repositorio.



```

@RestController no usages
@RequestMapping("/grupos")
public class GrupoController {
    @Autowired 7 usages
    private GrupoRepository grupoRepository;

    @GetMapping no usages
    public List<Grupo> getAllGrupos() {
        return grupoRepository.findAll();
    }

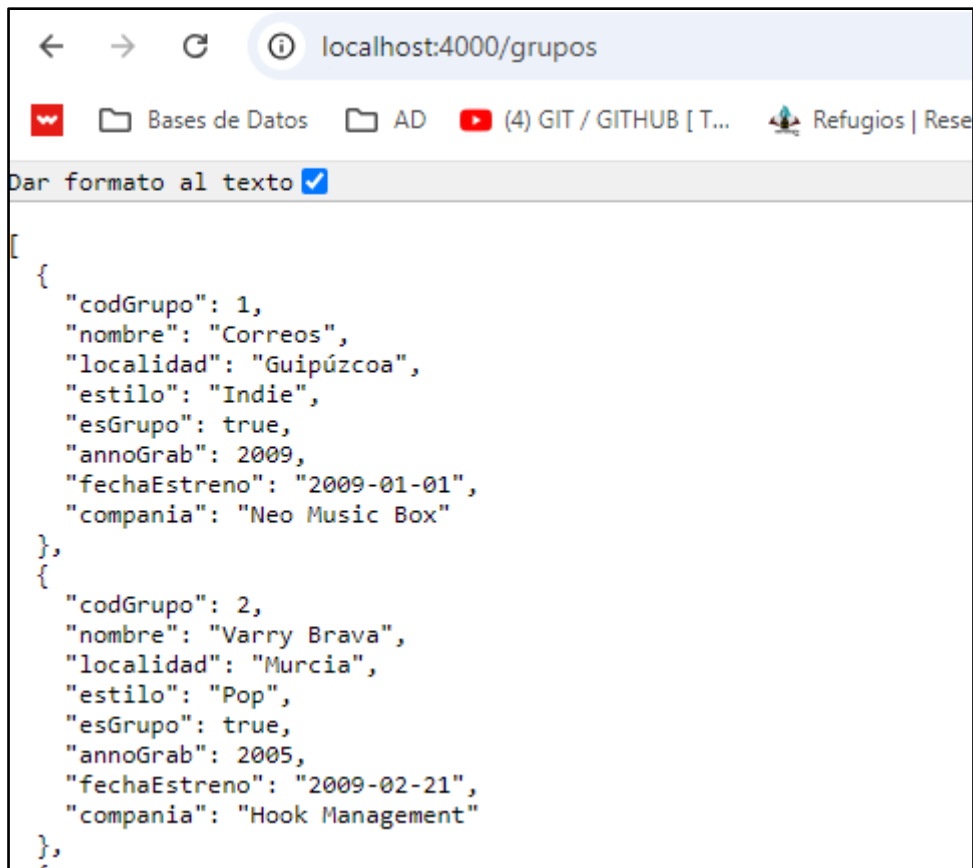
    @PostMapping no usages
    public String createGrupo(@RequestBody Grupo grupo) {
        int n=grupoRepository.save(grupo);
        if(n==1)
            return "Grupo creado correctamente";
        else
            return "El grupo no se ha podido crear";
    }
}

```

La clase se anota con **@RestController**.

- La anotación **@RequestMapping** indica que los endpoints de todos los métodos del controlador comienzan por **/grupos**.
- La anotación **@Autowired** hace que se inyecte un objeto de la clase anotada.
- El método **getAllGrupos** responde a las peticiones GET en el endpoint **/grupos**.
- El método **getAllGrupos** devuelve una lista de grupos.
- El método **createGrupo** responde a peticiones POST en el endpoint **/grupos**. El método **createGrupo** recibe en la petición POST un objeto grupo en formato JSON.

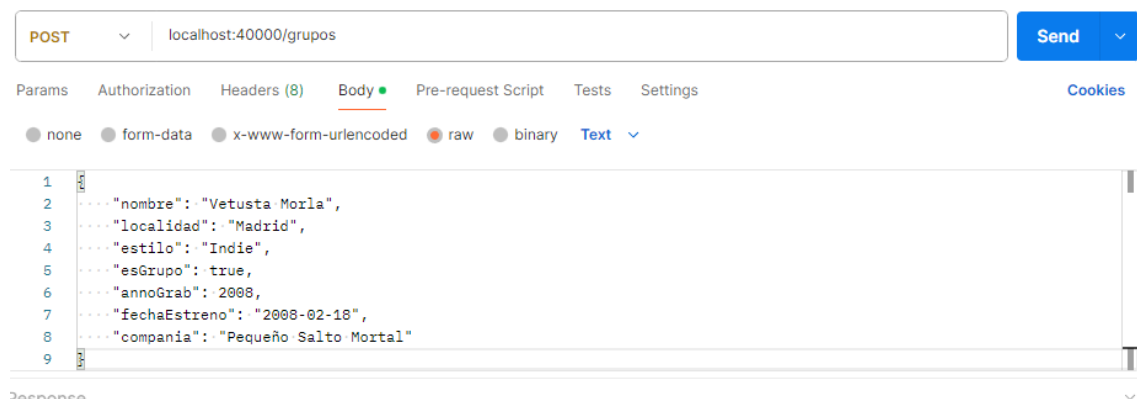
Podemos probar el endpoint **/grupos** para peticiones GET con Chrome usando la URL <http://localhost:4000/grupos>. Cabe recordar que en el fichero de propiedades especificamos que el puerto en el que responde el servicio es el 4000.



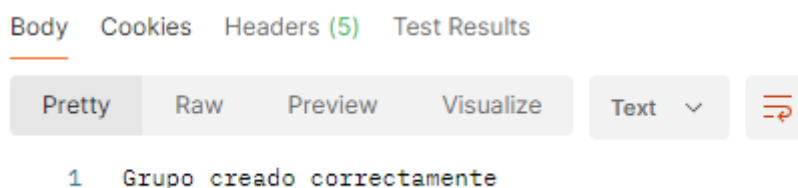
A screenshot of a web browser window. The address bar shows 'localhost:4000/grupos'. The page content displays a JSON array of two music groups. The first group has 'codGrupo': 1, 'nombre': 'Correos', 'localidad': 'Guipúzcoa', 'estilo': 'Indie', 'esGrupo': true, 'annoGrab': 2009, 'fechaEstreno': '2009-01-01', and 'compania': 'Neo Music Box'. The second group has 'codGrupo': 2, 'nombre': 'Varry Brava', 'localidad': 'Murcia', 'estilo': 'Pop', 'esGrupo': true, 'annoGrab': 2005, 'fechaEstreno': '2009-02-21', and 'compania': 'Hook Management'. A checkbox 'Dar formato al texto' is checked.

```
[
  {
    "codGrupo": 1,
    "nombre": "Correos",
    "localidad": "Guipúzcoa",
    "estilo": "Indie",
    "esGrupo": true,
    "annoGrab": 2009,
    "fechaEstreno": "2009-01-01",
    "compania": "Neo Music Box"
  },
  {
    "codGrupo": 2,
    "nombre": "Varry Brava",
    "localidad": "Murcia",
    "estilo": "Pop",
    "esGrupo": true,
    "annoGrab": 2005,
    "fechaEstreno": "2009-02-21",
    "compania": "Hook Management"
  }
]
```

El método POST no lo podemos probar con un navegador. Hay varias formas de probarlo. Vamos a usar la aplicación POSTMAN.



Hacemos la petición con SEND y esta debería ser la respuesta:



Mejorar el método POST para que en el mensaje http de respuesta se envíe un código HTTP 201 (**Created**) y el grupo creado en el cuerpo del mensaje cuando se

ha insertado correctamente el grupo y que se envíe un código HTTP 500 de error y un texto en el cuerpo cuando la inserción ha sido errónea.

```
@PostMapping no usages
public ResponseEntity<?> createGrupo2(@RequestBody Grupo grupo) {
    if(grupoRepository.save(grupo)==1)
        return ResponseEntity.status(HttpStatus.CREATED).body(grupo);
    else
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("No se pudo insertar el grupo.");
}
```

Ahora hacemos una mejora al método **getAll** para que incluya código de respuesta OK cuando devuelve uno o varios grupos y código de respuesta cuando no hubiera grupos.

```
@GetMapping no usages
public ResponseEntity<?> getAllGrupos() {
    List<Grupo> grupos=grupoRepository.findAll();
    if(grupos.isEmpty())
        return ResponseEntity.notFound().build();
    else
        return ResponseEntity.ok(grupos);
}
```

Ahora vamos a codificar correctamente los métodos que nos faltan en el repositorio:

```
@Override 3 usages
public Grupo findById(int id) {
    return jdbcTemplate.queryForObject(sql: "SELECT * FROM grupos WHERE codgrupo = ?",
        new GrupoRowMapper(), id);
}
```

```
@Override no usages
public List<Grupo> findByLocalidad(String loc) {
    return jdbcTemplate.query(sql: "SELECT * FROM grupos WHERE localidad = ?",
        new GrupoRowMapper(), loc);
}
```

```

@Override 1 usage
public int update(Grupo grupo) {
    try {
        return jdbcTemplate.update(
            sql: "UPDATE grupos SET nombre = ?, localidad = ?, estilo = ?, esgrupo = ?, annoGrab = ?, " +
                "fechaEstreno = ?, compania = ? WHERE codgrupo = ?",
            grupo.getNombre(), grupo.getLocalidad(), grupo.getEstilo(), grupo.isEsGrupo(),
            grupo.getAnnoGrab(), grupo.getFechaEstreno(), grupo.getCompania(), grupo.getCodGrupo()
        );
    }
    catch (Exception e){
        return -1;
    }
}

```

```

@Override 1 usage
public int deleteById(int id) {
    return jdbcTemplate.update( sql: "DELETE FROM grupos WHERE codgrupo = ?", id);
}

```

Y ahora vamos añadiendo otros métodos al controlador que se corresponden con los añadidos al repositorio.

Método GET para devolver un grupo a partir del id recibido.

```

@GetMapping("/{id}") no usages
public ResponseEntity<?> getGrupoById(@PathVariable int id) {
    Grupo grupo = grupoRepository.findById(id);
    if (grupo != null) {
        return ResponseEntity.ok(grupo);
    } else {
        return ResponseEntity.notFound().build();
    }
}

```

Método GET para devolver los grupos de una localidad recibida.

```

@GetMapping("/{localidad/{loc}") no usages
public ResponseEntity<?> getGrupoByLocalidad(@PathVariable String loc) {
    List<Grupo> grupos= grupoRepository.findByLocalidad(loc);
    if(grupos.isEmpty())
        return ResponseEntity.notFound().build();
    else
        return ResponseEntity.ok(grupos);
}

```

Método PUT para actualizar los datos de un grupo cuyos datos se han recibido en el cuerpo de la petición PUT y cuyo id de grupo se ha recibido en la petición.

```

@PutMapping("/{id}") no usages
public ResponseEntity<Grupo> updateGrupo(@PathVariable int id, @RequestBody Grupo grupo) {
    Grupo existingGrupo = grupoRepository.findById(id);
    if (existingGrupo != null) {
        grupo.setCodGrupo(id);
        grupoRepository.update(grupo);
        return ResponseEntity.ok(grupo);
    } else {
        return ResponseEntity.notFound().build();
    }
}

```

Método DELETE para eliminar el grupo cuyo id se ha recibido en la petición.

```

@DeleteMapping("/{id}") no usages
public ResponseEntity<Void> deleteGrupo(@PathVariable int id) {
    Grupo existingGrupo = grupoRepository.findById(id);
    if (existingGrupo != null) {
        grupoRepository.deleteById(id);
        return ResponseEntity.noContent().build();
    } else {
        return ResponseEntity.notFound().build();
    }
}

```

Y ya podemos probar todos los endpoints.