

UNIDAD 4. DESARROLLO DE COMPONENTES CON SPRING

Contenido

1. SPRING FRAMEWORK	2
2. SPRING BOOT PARA LA CONSTRUCCIÓN DE UNA API REST	2
2.1. Desarrollo de una aplicación Spring Boot usando Spring Initializr	3
2.2. Creación de una API REST básica	3
2.3. Creación de una API REST de un Java Bean.....	4
2.4. Uso de Path Variables.....	6
2.5. Uso de Query Params.....	6
3. DESARROLLO DE UN CRUD CON SPRING JPA Y SPRINT REST	7
3.1. Arquitectura del proyecto	8
3.2. Configuración de MySQL.....	9
3.3. Creación de la entidad JPA Empleado	10
3.4. Implementación de la capa DAO.....	11
3.5. Testing de la Api REST con postman.....	11
4. BEANS. INYECCIÓN DE DEPENDENCIAS	13
5. CONSTRUCCIÓN DE VISTAS HTML CON SPRING Y THYMELEAF	13

1. SPRING FRAMEWORK

Spring es un framework JavaEE muy popular para crear aplicaciones web y empresariales. Spring ha evolucionado para convertirse en un ecosistema completo que soporta múltiples necesidades de desarrollo, como el acceso a datos, la creación de microservicios, y el manejo de seguridad. Se basa en principios como la **Inversión de Control (IoC)** y la **Inyección de Dependencias (DI)**, lo que permite a los desarrolladores construir aplicaciones más modulares, testeables y mantenibles.

Los componentes clave de Spring son:

- **Spring Core:** El corazón del framework, que ofrece funcionalidades de IoC y DI para gestionar los objetos y sus dependencias.
- **Spring MVC:** Permite construir aplicaciones web siguiendo el patrón Modelo-Vista-Controlador (MVC).
- **Spring Data:** Simplifica el acceso a bases de datos y la gestión de repositorios.
- **Spring Security:** Proporciona herramientas avanzadas para la seguridad de aplicaciones, como autenticación y autorización.
- **Spring Boot:** Una extensión del framework que simplifica la configuración y el despliegue de aplicaciones con "convenciones sobre configuración".
- **Spring Cloud:** Facilita el desarrollo de aplicaciones distribuidas y microservicios, integrando herramientas para la comunicación, configuración, descubrimiento de servicios, y resiliencia.

Y tiene múltiples ventajas el uso de Spring:

- **Modularidad:** Permite desarrollar aplicaciones desacopladas mediante IoC y DI, haciendo que los componentes sean independientes y reutilizables.
- **Facilidad de Configuración:** Con Spring Boot, se eliminan muchas configuraciones manuales complejas gracias a configuraciones predeterminadas que "funcionan".
- **Ecosistema Completo:** Proporciona soluciones para desarrollo web, acceso a datos, integración con mensajería, seguridad, monitorización, y más.
- **Soporte para Microservicios:** Spring Boot y Spring Cloud ofrecen herramientas para crear y gestionar arquitecturas modernas basadas en microservicios.
- **Testabilidad:** Las aplicaciones creadas con Spring son altamente testeables, ya que permite utilizar fácilmente mock objects y pruebas unitarias gracias a su diseño modular.

2. SPRING BOOT PARA LA CONSTRUCCIÓN DE UNA API REST

Spring Boot aborda el problema de que las aplicaciones Spring necesitan una configuración compleja eliminando la necesidad de configurar manualmente la configuración repetitiva. De manera que Spring Boot analizará el entorno de tu aplicación y configurará automáticamente componentes de Spring según lo que encuentre en el **classpath** y las configuraciones.

Se puede desarrollar una aplicación Spring Boot mediante Spring Initializr, Spring Starter Project en STS ("Spring Tool Suite") o usando Spring Boot CLI.

2.1. Desarrollo de una aplicación Spring Boot usando Spring Initializr

Con la plataforma web <https://start.spring.io/> podemos inicializar un proyecto con las dependencias deseadas y tendrá las configuraciones automáticamente. En nuestro caso vamos a configurarlo para que sea una API REST, añadiendo la dependencia Spring Web:

The screenshot shows the Spring Initializr web interface. It has a sidebar on the left with sections: Project, Language, Spring Boot, Project Metadata, and Packaging. The main area on the right shows the selected options and a list of dependencies.

Project: ☐ Gradle - Groovy ☐ Gradle - Kotlin ☒ Java ☐ Kotlin ☐ Groovy

Language: ☒ Maven

Spring Boot: ☐ 3.4.2 (SNAPSHOT) ☒ 3.4.1 ☐ 3.3.8 (SNAPSHOT) ☐ 3.3.7

Project Metadata:

- Group:
- Artifact:
- Name:
- Description:
- Package name:

Packaging: ☒ Jar ☐ War

Java: ☐ 23 ☒ 21 ☐ 17

Dependencies:

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

Al pulsar Generar nos descargará un archivo con el esqueleto de la aplicación Spring.

2.2. Creación de una API REST básica

Las **API REST** (*"Representational State Transfer"*) permiten la comunicación entre sistemas de manera eficiente, utilizando HTTP como protocolo y principios como la **escalabilidad, la simplicidad y la interoperabilidad**. Estas características hacen que las API REST sean ideales para conectar aplicaciones modernas, ya que los clientes (como aplicaciones móviles o web) pueden interactuar con servidores para gestionar datos o ejecutar operaciones.

Este servicio REST del servidor tendrá múltiples endpoints donde accederán los frontend. Un **endpoint** en una API REST es una URL específica que actúa como un punto de acceso para que los clientes interactúen con el servidor. Representa un **recurso** (como un usuario, producto, pedido, etc.) y define cómo se puede acceder o manipular dicho recurso a través de los métodos HTTP estándar:

- **GET:** Para obtener información del recurso.
- **POST:** Para crear un nuevo recurso.
- **PUT:** Para actualizar un recurso existente.
- **DELETE:** Para eliminar un recurso.

Ejemplo de API REST HolaMundo, que implementa un HTTP GET en el endpoint `"/hola-mundo"`:

```
package com.ejemploSpring.demo;

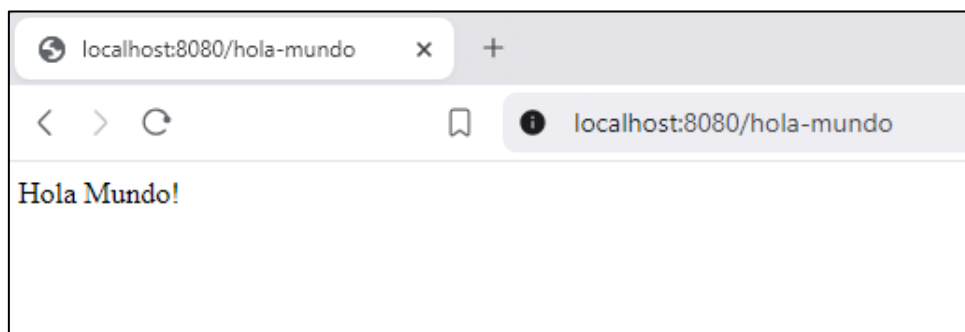
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HolaMundoController {
    // GET HTTP Method
```

```
// http://localhost:8080/hola-mundo
@GetMapping("/hola-mundo")
public String holaMundo() {
    return "Hola Mundo!";
}
```

- El código anterior utiliza la nueva anotación `@RestController` de Spring 4, que marca la clase como un controlador en el que cada método **devuelve un objeto de dominio** en lugar de una vista. Es la abreviatura de `@Controller` y `@ResponseBody` juntas.
- Anotación **`@GetMapping` para asignar peticiones HTTP GET** a métodos específicos.
- Si todos los métodos se mapearan al mismo recurso se puede añadir `@RequestMapping` en la propia clase, de manera que si se añade un mapeo a un método se concatena al endpoint establecido en el `RequestMapping` de la clase. Por tanto, **`RequestMapping` define un endpoint común** para todos los métodos definidos en la clase.

Si ejecutamos el main se lanzará este servicio en un servidor Tomcat embebido escuchando en el puerto 8080. Para lanzar una petición GET podemos utilizar el navegador introduciendo el endpoint: <http://localhost:8080/hola-mundo>



2.3. Creación de una API REST de un Java Bean

Un **Java Bean** es una clase Java que sigue un conjunto específico de convenciones de diseño, utilizadas principalmente para encapsular datos y facilitar su manipulación en entornos de desarrollo. Sus características son:

- Constructor público sin parámetros (si se va a utilizar JPA).
- Atributos privados y propiedades accesibles mediante métodos getter y setter.

```
package com.ejemploSpring.demo;

public class Estudiante {
    private String nombre;
    private String apellidos;

    public Estudiante(String nombre, String apellidos) {
        this.nombre = nombre;
        this.apellidos = apellidos;
    }
}
```

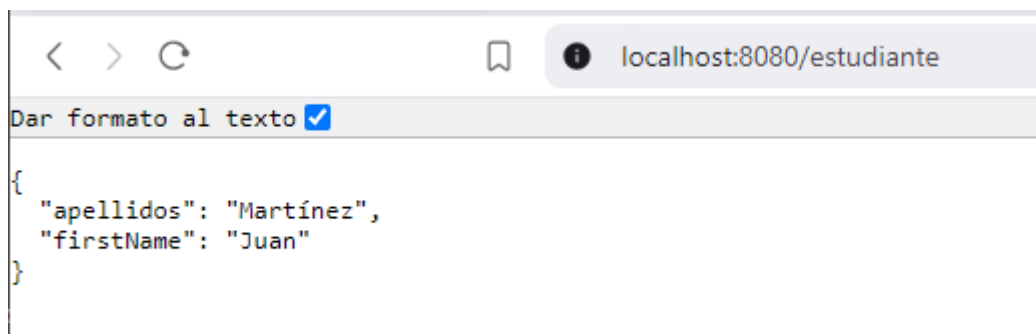
```
public String getNombre() {
    return nombre;
}

public void setNombre(String nombre) {
    this.nombre = nombre;
}

public String getApellidos() {
    return apellidos;
}

public void setApellidos(String apellidos) {
    this.apellidos = apellidos;
}
}
```

El objeto Estudiante debe convertirse a JSON. Gracias al soporte del conversor de mensajes HTTP de Spring, no es necesario realizar esta conversión manualmente, sino que se utiliza internamente la API Jackson para convertir la instancia de Estudiante a JSON:



```
{
  "apellidos": "Martínez",
  "firstName": "Juan"
}
```

También podemos devolver una lista que se convertirá automáticamente a JSON:

```
@GetMapping("/estudiantes")
public List<Estudiante> getEstudiantes() {
    List<Estudiante> estudiantes=new ArrayList<>();
    estudiantes.add(new Estudiante("Aitor","Tilla"));
    estudiantes.add(new Estudiante("Aitor","Menta"));
    estudiantes.add(new Estudiante("Benito","Camelo"));
    return estudiantes;
}
```

Por otro lado, para implementar un **POST** es similar, pero el tipo de retorno es un **ResponseEntity** (paquete HTTP Response) y los parámetros se reciben dentro del cuerpo del mensaje (@RequestBody):

```
@PostMapping("/estudiantes")
public ResponseEntity<Empleado> saveEmpleado(@RequestBody Empleado
empleado) {
    return new
ResponseEntity<>(empleadoService.saveEmpleado(empleado),
HttpStatus.CREATED);
}
```

¿Por qué usamos ResponseEntity en lugar de directamente devolver el objeto empleado?

Porque Spring por defecto envuelve la respuesta en un HTTP Response 200 OK, y si queremos tener más control sobre lo que se devuelve, como por ejemplo, devolver un HTTP 201 CREATED, hay que envolver el objeto en un ResponseEntity.

Destacan los siguientes códigos HTTP:

2xx – Éxito (Solicitud procesada con éxito):

- **200 OK.** La solicitud fue exitosa y la respuesta contiene el resultado. Por ejemplo, GET o PUT de empleados.
- **201 Created.** La solicitud fue exitosa y se creó el recurso. Ejemplo, POST de creación de empleado.
- **204 No content.** La solicitud fue exitosa pero no hay contenido. Ejemplo, DELETE de empleado.

4xx – Errores del cliente.

- **400. Bad Request.** Solicitud inválida, por ejemplo, un JSON mal formado.
- **401. Unauthorized.** El cliente no está autenticado para realizar esa operación.
- **403. Forbidden.** El cliente está autenticado pero no tiene permiso para realizar la operación.
- **404. Not Found.** El recurso no existe. Ejemplo: no existe empleado con ese id.
- **409. Conflict.** Se produce un conflicto debido al estado actual del recurso. Ejemplo, ya existe un empleado con ese email para poder registrarlo.

5xx – Errores del servidor.

- **500. Internal Server Error.** Excepción no controlada.
- **501. Not implemented.** El servidor no tiene implementada esa funcionalidad. Ejemplo: un método HTTP no soportado.
- **503 Service Unavailable.** El servidor no está disponible, por ejemplo, sobrecarga o está apagado.

2.4. Uso de Path Variables

En una API REST, las Path Variables son partes dinámicas de la URL que se utilizan para **identificar un recurso específico y es obligatorio**.

Ejemplo. Es más legible /estudiante/Juan/Perez que /estudiante?nombre=Juan&apellidos=Perez

```
@GetMapping("/estudiante/{id}")
public Estudiante getEstudianteByNombre(@PathVariable("id") long id) {
    return service.findById(id);
}
```

Ejemplo de uso: <http://localhost:8080/estudiante/Javier/Uribe>

2.5. Uso de Query Params

Una Query param se utiliza para filtrar por alguna característica. En otras palabras, los Query Params son para complementar la búsqueda mientras que mediante los Path Params establecemos el recurso que estamos buscando.

El Query param comienza con el signo interrogante "?" y el resto de parámetros con el signo ampersand("&").

```
@GetMapping
public List<Cliente> findAll(@RequestParam(value = "marca", required =
false) String marca,
                           @RequestParam(value = "nombre", required
= false) String nombre) {
    if (marca != null) {
        return clienteService.findByMarca(marca);
    }
    if (nombre != null) {
        return clienteService.findByNombre(nombre);
    }
    return clienteService.getAllClientes();
}
```

IMPORTANTE: Si queremos que el query param sea opcional se marca **required=false**.

Ejemplo de uso: <http://localhost:8080/estudiante?nombre=Javier>

2.6. Serialización de objetos con JSON

Spring utiliza internamente la librería Jackson para serializar los objetos JSON.

Un error común son los ciclos que se producen cuando un objeto contiene un objeto que contiene al padre. Para evitar el bucle se debe añadir la etiqueta:

```
@JsonIgnore
```

Y también hay que tener en cuenta que en JSON tampoco mapea la herencia automáticamente. Para ello hay que especificar la propiedad donde se va a mapear. Ejemplo: tipoProducto.

```
@JsonTypeInfo(use = JsonTypeInfo.Id.NAME, property = "tipoProducto")
```

Y también hay que especificar los valores a los que se mapearán las subclases:

```
@JsonSubTypes({
    @JsonSubTypes.Type(value = Pizza.class, name = "pizza"),
    @JsonSubTypes.Type(value = Pasta.class, name = "pasta")
})
```

Cuando lancemos una petición POST habrá que especificar el tipo de producto:

```
{
    "nombre": "Margarita",
    "tipoProducto": "pizza",
    "precio": "12.5",
}
```

3. DESARROLLO DE UN CRUD CON SPRING JPA Y SPRINT REST

Se va a desarrollar una aplicación completa utilizando Initializr que lleva Spring Web, Spring JPA, MySQL Driver y Lombok:

The screenshot shows the Spring Initializr web application. It has a sidebar on the left with sections: Project, Language, Spring Boot, Project Metadata, and Packaging. The main area on the right shows the selected dependencies: Spring Web, Spring Data JPA, MySQL Driver, and Lombok. The Project section shows Maven selected. The Language section shows Java selected. The Spring Boot section shows 3.4.1 selected. The Project Metadata section shows fields for Group (com.example), Artifact (demo), Name (demo), Description (Demo project for Spring Boot), and Package name (com.example.demo). The Packaging section shows Jar selected. The Dependencies section shows a list of dependencies with their versions and descriptions.

CUIDADO CON EL fichero pom.xml que genera la plataforma, ya que la dependencia de Lombok no funciona y hay que cambiarla a la que conocemos de Maven:

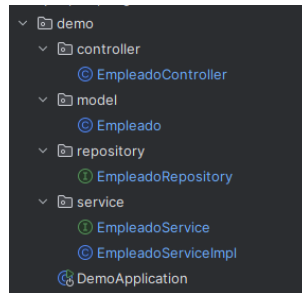
```
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <scope>provided</scope>
</dependency>
```

3.1. Arquitectura del proyecto

La arquitectura del proyecto se basa en la arquitectura MVC con el patrón DAO vista hasta ahora, pero adaptándose a las necesidades de las aplicaciones webs modernas.

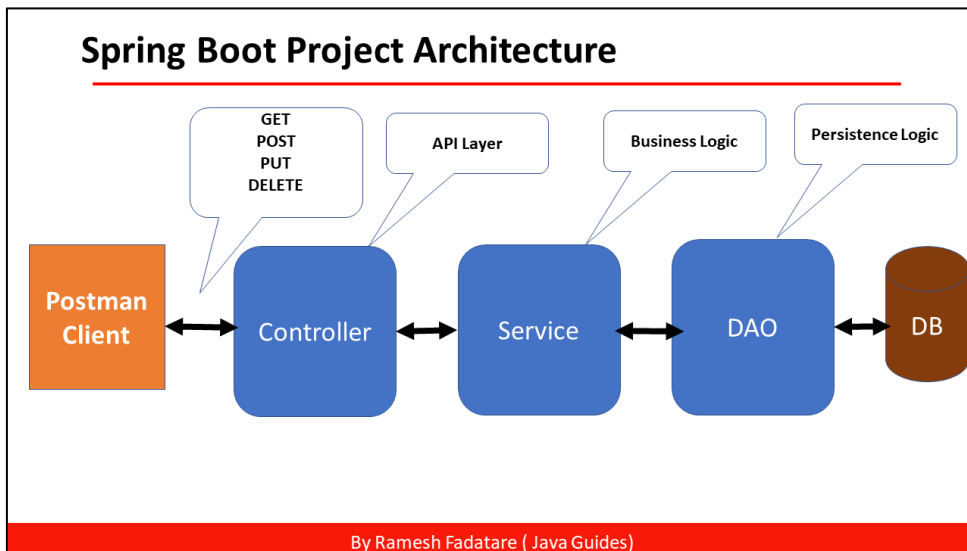
- **Model.** Representa la lógica de negocio, pero si utiliza JPA, también representará los datos de la aplicación.
- **Repository.** Contiene las interfaces de repositorio que heredan de JpaRepository y serían el equivalente al DAO.
- **Service.** Es el coordinador de la lógica de negocio y el que interactúa con el Dao (Repository) para llevar a cabo la persistencia. Es el equivalente al Controlador que veíamos.
- **Controller.** Cumple el rol de intermediario pero cambia con respecto a lo que conocíamos. Ya que ahora se encarga de manejar las solicitudes HTTP provenientes del clientes (API REST) y de darle a la vista lo que se necesita.
- **Vista.** Si usamos JSP con Thymeleaf, serían los templates que contienen las plantillas HTML que se rellenan con los datos proporcionados por el Controller.

Ejemplo de estructura de paquetes aplicando la arquitectura Spring MVC:



Aquí se muestra la traza.

1. El cliente realiza una petición REST.
2. El Controller gestiona la petición REST, haciendo uso del Service y rellena los datos que requiere la vista.
3. El Service gestiona la lógica de negocio e interactúa con el DAO para hacer la persistencia.
4. El DAO (Repository) gestiona la persistencia automáticamente con la clase JpaRepository.



3.2. Configuración de MySQL

Spring Boot autoconfigura un Data Source (fuente de datos) para simplificar el proceso, evitando tener que crear un fichero persistence.xml. Para ello, hay que modificar el fichero application.properties:

```
spring.application.name=demo
spring.datasource.url = jdbc:mysql://localhost:3306/demo
useSSL=false&serverTimezone=UTC&useLegacyDatetimeCode=false
spring.datasource.username = root
spring.datasource.password = admin
## Hibernate Properties
# The SQL dialect makes Hibernate generate better SQL for the chosen
database
spring.jpa.properties.hibernate.dialect =
org.hibernate.dialect.MySQLDialect
# Hibernate ddl auto (create, create-drop, validate, update)
spring.jpa.hibernate.ddl-auto = update
```

Las anotaciones hibernate cambian en Spring y "create" hace el equivalente a drop-and-create.

3.3. Creación de la entidad JPA Empleado

```
import jakarta.persistence.*;
import lombok.Data;

@Data
@Entity
public class Empleado {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    private String nombre;
    private String apellido;
    @Column(nullable = false)
    private String email;
}
```

La anotación **@Data** nos la proporciona **Lombok** y se usa para evitar el proceso repetitivo de definir getters y setters. Si también queremos que nos genere constructor con parámetros y sin parámetros se añade las etiquetas:

```
@Data
@AllArgsConstructor //Genera constructor con parámetros
@NoArgsConstructor //Genera constructor sin parámetros
```

Hay que tener en cuenta para el testing que si yo quiero que implemente el equals y el hashCode hay que añadir la etiqueta:

```
@EqualsAndHashCode
```

Cuidado: Cuando testeamos objetos que contienen listas, en JPA las mapea a PersistentBag, una clase donde no funciona el equals correctamente por lo que habrá que volver a convertirlo en el test a list:

```
@Test
void testRegistrarCliente() {
    Cliente cliente = crearCliente();
    clienteService.save(cliente);
    Cliente clienteBD = clienteService.findClienteById(1);
    clienteBD.setCoches(new ArrayList<>(clienteBD.getCoches()));
    cliente.setCoches(new ArrayList<>(cliente.getCoches()));
    assertEquals(cliente, clienteBD);
}
```

También hay que tener en cuenta que se pueden producir bucles infinitos en los métodos toString y el equals, por lo que habrá que excluirlos con anotaciones:

```
@Entity
@Data
@AllArgsConstructor //Genera constructor con parámetros
@NoArgsConstructor //Genera constructor sin parámetros
@EqualsAndHashCode
public class Coche {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String matricula;
    private String marca;
}
```

```
private String modelo;  
private LocalDate fecha;  
@ManyToOne(cascade = CascadeType.ALL)  
@ToString.Exclude  
@EqualsAndHashCode.Exclude  
private Cliente propietario;  
}
```

3.4. Implementación de la capa DAO

La interfaz JpaRepository define metodos para todas las operaciones CRUD sobre la entidad, y una implementación por defecto del JpaRepository llamada SimpleJpaRepository. Esto implica que **Spring crea instancias de SimpleJpaRepository automáticamente durante el runtime cuando detecta un bean que extiende de JpaRepository**. Por tanto, no hay que implementar en el DAO las operaciones básicas de JPA.

Si quisiéramos añadir algún método adicional, JPA los implementa automáticamente si seguimos la convención del prefijo:

- **existsByX**: Devuelve un booleano indicando si hay algún registro que cumple la condición.
- **findByX**: Recupera uno o más registros que cumplen la condición.
- **countByX**: Devuelve el número de registros que cumplen la condición.
- **deleteByX**: Elimina los registros que cumplen la condición.

Por ejemplo, el método existsByEmail no existe pero lo podemos definir de esta manera y será implementado automáticamente:

```
public interface EmpleadoRepository extends JpaRepository<Empleado,  
Long> {  
    boolean existsByEmail(String email);  
}
```

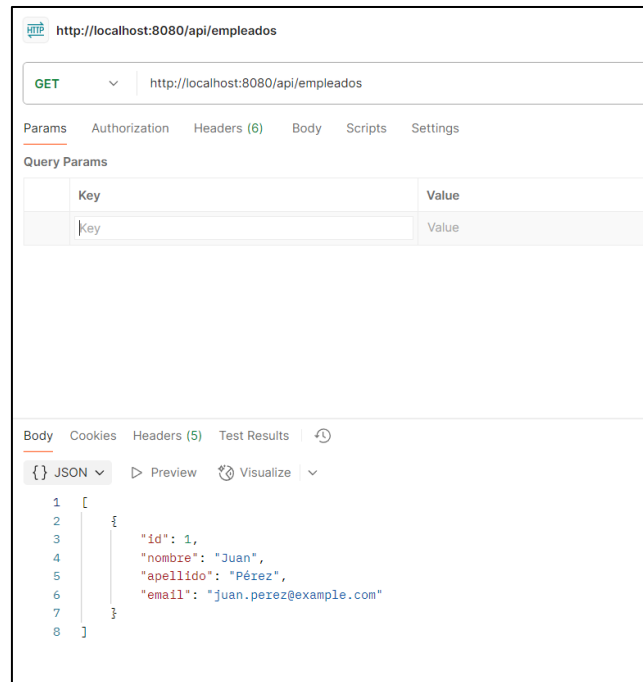
Así podemos hacer más robusta la aplicación, impidiendo que se registre un empleado que ya exista con ese email, devolviendo un código 409:

```
@Override  
public Empleado saveEmpleado(Empleado empleado) {  
    if (empleadoRepository.existsByEmail(empleado.getEmail())) {  
        throw new ResponseStatusException(HttpStatus.CONFLICT, "Ya  
existe un empleado con el email:" + empleado.getEmail());  
    }  
    return empleadoRepository.save(empleado);  
}
```

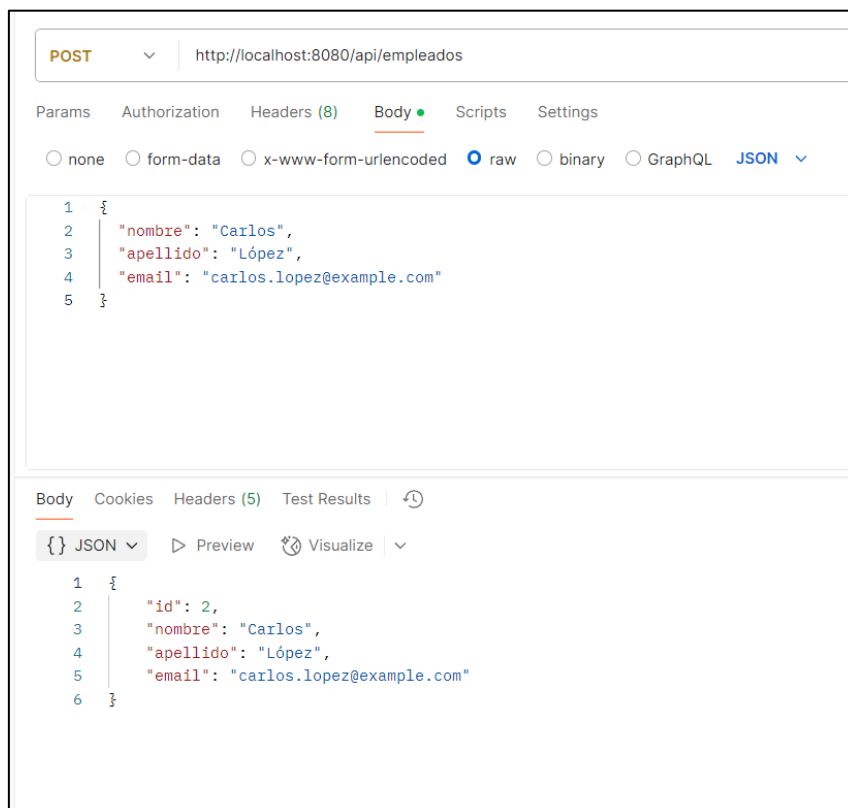
3.5. Testing de la Api REST con postman

Podemos enviar una petición HTTP mediante la aplicación POSTMAN.

Ejemplo de HTTP GET (en params podemos establecer los query params).



Pero sobre todo, es interesante para hacer peticiones POST. En este ejemplo, metemos en formato JSON los parámetros que se necesitan para crear un empleado y que serán parseados con `@RequestBody` en nuestra API REST:



4. BEANS. INYECCIÓN DE DEPENDENCIAS

La **inyección de dependencias** permite que Spring se encargue de la creación y gestión de los objetos requeridos, lo que facilita la organización del código y mejora la reutilización. En lugar de que una clase tenga que gestionar la creación de sus dependencias, Spring se encarga de inyectarlas en el momento adecuado, lo que ayuda a mantener el código limpio, modular y fácil de probar.

En el siguiente ejemplo, tenemos una clase `EmpleadoServiceImpl`, que requiere una instancia de `EmpleadoRepository` para realizar operaciones sobre la entidad `Empleado`. Usamos la anotación `@Autowired` para que Spring se encargue de la creación y asignación de esta dependencia automáticamente. De esta manera, ya no es necesario implementar un constructor que inicialice el repository.

```
@Service
public class EmpleadoServiceImpl implements EmpleadoService{

    @Autowired // Inyección automática de la dependencia
    private EmpleadoRepository empleadoRepository;
```

Para que `Autowired` pueda inicializar la dependencia, esta dependencia debe ser un **bean (o componente)**. Características de un bean:

- Indica que la clase está destinada a contener **lógica empresarial o de servicios**.
- Spring gestiona el bean con el **patron Singleton** para que solo haya 1 instancia en toda la aplicación.
- Serán beans los siguientes componentes:
 - `@Repository` (capa de bases de datos, equivale a nuestro anterior DAO en MVC)
 - `@RestController` (capa servicio REST)
 - `@Configuration` (Configuraciones)
 - `@Service` (capa de lógica de negocio y coordinador, equivale a nuestro anterior Controlador en MVC)

5. CONSTRUCCIÓN DE VISTAS HTML CON SPRING Y THYMELEAF

Además, de devolver objetos en formato JSON, también se pueden devolver páginas vistas HTML. Para ello es necesario utilizar un motor de plantillas para Java que permite generar contenido HTML dinámico desde el servidor. Uno de los motores más utilizados es **Thymeleaf**.

Dependencia:

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

Creamos una nueva clase que representará la vista HTML.

```
package com.ejemploSpring.demo;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
```

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;

import java.util.ArrayList;
import java.util.List;

@Controller
@RequestMapping("/mostrar")
public class EstudianteVistaController {

    @GetMapping("estudiantes")
    public String mostrarEstudiantes(Model model) {
        List<Estudiante> estudiantes = new ArrayList<>();
        estudiantes.add(new Estudiante("Aitor", "Tilla"));
        estudiantes.add(new Estudiante("Mario", "Neta"));
        estudiantes.add(new Estudiante("Benito", "Camelo"));
        model.addAttribute("estudiantes", estudiantes);
        return "lista-Estudiantes";
    }

    @GetMapping("estudiante/{nombre}/{apellidos}")
    public String mostrarEstudiante(@PathVariable("nombre") String
nombre,
                                @PathVariable("apellidos") String
apellidos,
                                Model model) {
        model.addAttribute("estudiante", new Estudiante(nombre,
apellidos));
        return "datos-Estudiante";
    }
}
```

- Ahora no es un @RestController, que devuelve JSON, sino que son @Controller.
- Para **pasar datos desde el Controller hasta la vista html se utiliza Model**. Se almacenan como pares clave-valor.
- Los métodos devuelven un String. Este String representa la página web que será la salida de la llamada, pero no es estática, es generada dinámicamente con los datos que lleva el Model. Por tanto, **este String debe coincidir con el nombre del HTML guardado en resources/templates**. Ejemplo: resources/templates/datos-Estudiante.html

Ejemplo de datos-Estudiante.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Datos del estudiante</title>
    <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap
.min.css">
</head>
<body>
<h1>Datos de un estudiante</h1>
<h3>Nombre :</h3> <span th:text="${estudiante.nombre}"> Nombre </span>
<br>
<h3>Apellidos :</h3> <span th:text="${estudiante.apellidos}">
Apellidos </span>
</body>
</html>
```

Ejemplo de lista-Estudiantes.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Lista de estudiantes</title>
  <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap
.min.css">
</head>
<body>
<h1>Listado de Estudiantes</h1>
<table>
  <thead>
    <tr>
      <th> Nombre</th>
      <th> Apellidos</th>
    </tr>
  </thead>
  <tbody>
    <tr th:each="estudiante : ${estudiantes}">
      <td><span th:text="${estudiante.nombre}"> Nombre </span></td>
      <td><span th:text="${estudiante.apellidos}"> Apellidos
</span></td>
    </tr>
  </tbody>
</table>
</body>
</html>
```

Thymeleaf usa expresiones `${}` para insertar datos dinámicos y para poder usarlo se puede insertar una directiva como atributo en cualquier elemento HTML. Las directivas más importantes son:

th:text. Se usa para insertar texto dinámico dentro de un elemento.

```
<p th:text="${cliente.nombre}"></p>
```

th:each. Se usa para recorrer colecciones.

```
<ul>
  <li th:each="cliente : ${clientes}"
th:text="${cliente.nombre}"></li>
</ul>
```

th:if. Se usa para mostrar un elemento solo si se cumple una condición.

```
<p th:if="${cliente.admin}">Este cliente es administrador</p>
```

th:switch. Permite evaluar condiciones múltiples.

```
<div th:switch="${cliente.tipo}">
  <p th:case="'admin'">Cliente Administrador</p>
```

```
<p th:case="'regular'">Cliente Regular</p>
<p th:case="*">Cliente Desconocido</p>
</div>
```