

*“AÑO DE LA RECUPERACIÓN DE LA ECONOMÍA
PERUANA”*

**UNIVERSIDAD NACIONAL DEL CENTRO
DEL PERÚ**



**FACULTAD DE INGENIERÍA DE SISTEMAS
SPRING API-GESTIÓN DOCENTE**

DOCENTE:

SUASNÁBAR TERREL, Jaime

PRESENTADO POR:

ESPINOZA MORALES, Keyla Xiomara

LEON GARCIA, David Daniel

HUANCAYO - PERÚ

2025

DESARROLLO API - GESTIONDOCENTES

Desarrollar una API RESTFUL para la gestión de docentes utilizando Jakarta EE y Spring. La aplicación permitirá realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminary Otras) sobre una entidad llamada Docente, que estará asociada a una base de datos MySQL.

PASO 1.

En el siguiente link: <https://start.spring.io/> para crear el proyecto en spring, para lo cual debe tener las siguientes características:

- **Project:** Maven
- **Language:** Java
- **Spring Boot:** 3.5.3
- **Project Metadata:**
 - Group, com.ddleon; Artifact, gestiondocentes; Name, gestiondocentes; Description, Demo project for Spring Boot; Package name, com.ddleon.gestiondocentes.
- **Dependencies:** Spring Web, Spring Data JPA y MySQL Driver.

Cuando se tiene todas las características señaladas como se muestra en la figura 1, continuando, se le da click en "GENERATE", después se creará el archivo comprimido listo para descargarlo y descomprimirlo en tu ordenador.

Figura 1

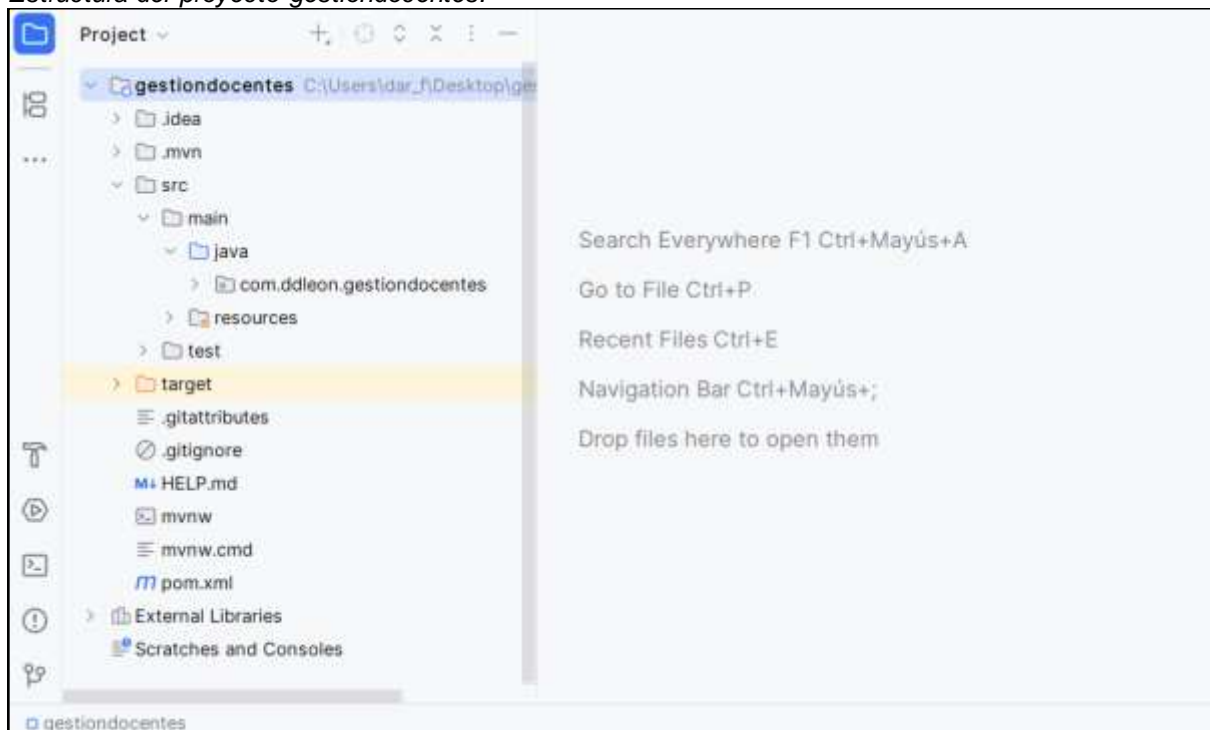
Creación del proyecto en spring y sus respectivas características consideradas.

The screenshot shows the Spring Initializr web interface. The 'Project' section has 'Gradle - Groovy' selected. The 'Language' section has 'Java' selected. The 'Spring Boot' section has '3.5.3' selected. The 'Project Metadata' section shows the following details: Group: com.ddleon, Artifact: gestiondocentes, Name: gestiondocentes, Description: Demo project for Spring Boot, Package name: com.ddleon.gestiondocentes, Packaging: Jar, and Java version: 21. The 'Dependencies' section shows 'Spring Web' (WEB), 'Spring Data JPA' (SQL), and 'MySQL Driver' (SQL) selected. At the bottom, there are buttons for 'GENERATE' (CTRL + G), 'EXPLORE' (CTRL + SPACE), and a menu icon.

Cuando se genere la carpeta comprimida, se descarga y descomprime, para luego abrir el proyecto **gestiondocentes** con **IntelliJ IDEA**. Entonces se genera la siguiente estructura que muestra la figura 2, es importante tener configurado las variables de entorno y dentro de la misma el **path**, tanto para **maven** y **jdk**.

Figura 2

Estructura del proyecto gestiondocentes.



La estructura está compuesta por **pom.xml**, tiene la estructura principal del proyecto, que contiene todas las dependencias de Spring, entonces cada que se necesite exportar librerías se tendrá que agregar dentro de dependencias. En build hace referencia a la configuración cuando se realice la compilación.

En la carpeta **src** se tiene todo el código fuente, se divide en dos partes, **main** y **test**, en **main** se coloca todo el código del proyecto, en el segundo tiene todo el testing. Dentro de **main** se tiene **java** y **resources**, dentro de **java** se incorpora el código fuente del proyecto y dentro de **resources** se coloca los archivos de configuración, como lo son archivos html o de frontend.

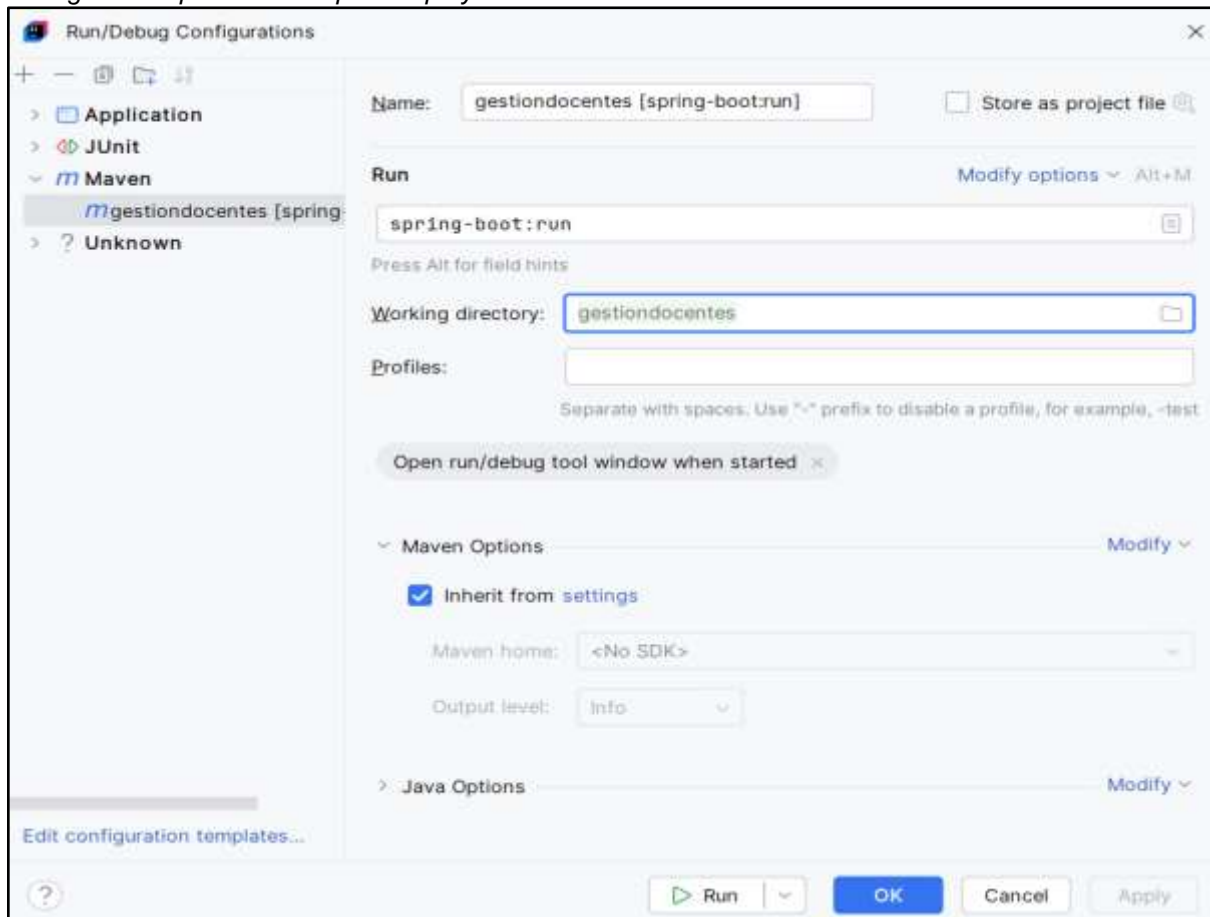
PASO 2.

Es importante que **Run / Debug Configuration** este configurado para poder ejecutar el proyecto **gestiondocentes**, en name debe aparecer **gestiondocentes [Spring-boot:run]**, en run, **spring-boot:run** y en working directory, **gestiondocentes**, luego se le da click en aplicar para guardar la configuración y finalmente en OK para cerrar la ventana, todo esto se ve en la figura 3 que se muestra a continuación.

Posteriormente se empezará con la configuración del proyecto para poder establecer la conexión con la base de datos MySQL.

Figura 3

Configuración para el arranque del proyecto con maven.

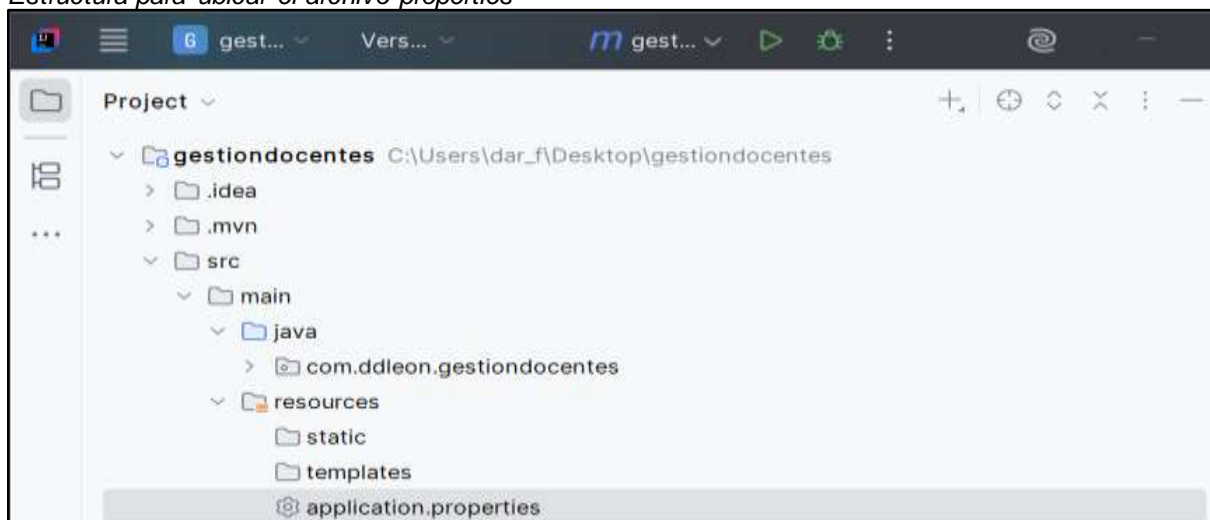


PASO 3.

Se ubica el archivo **src/main/resources/application.properties** en el proyecto.

Figura 4

Estructura para ubicar el archivo properties



Se incorpora el código que configure y así establezca la conexión con la base de datos MySQL, el cual tiene por nombre de la aplicación **gestiondocentes** además que está trabajando por el puerto **3306** y a la par se conectará con la base de datos **gestiondocentesdb**, también se debe considerar el nombre de usuario y la contraseña, el usuario para este caso es **root** y la contraseña **LeG@1998**.

Figura 5

Código para establecer la conexión con la base de datos gestiondocentesdb en MySQL.

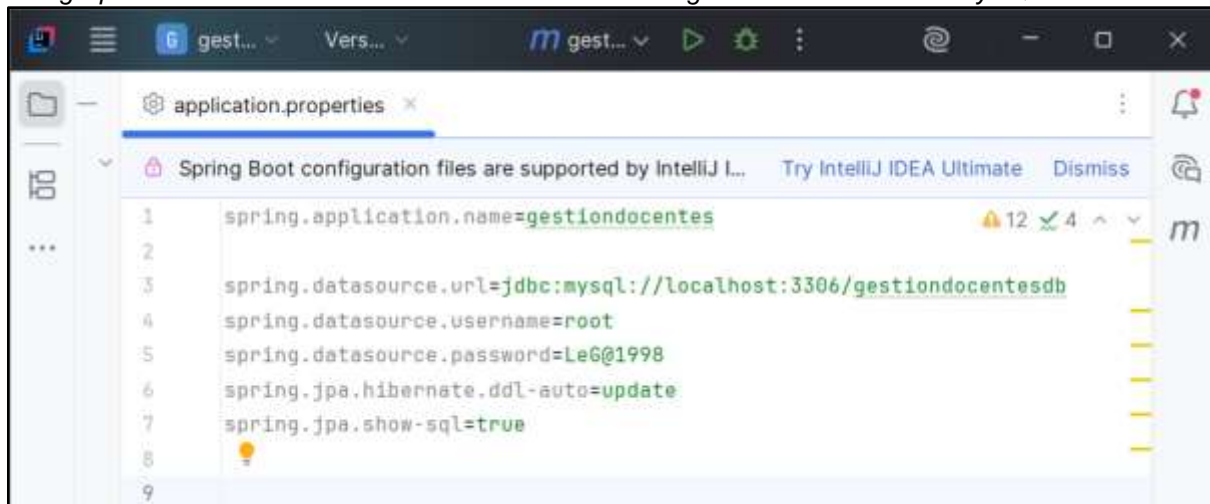
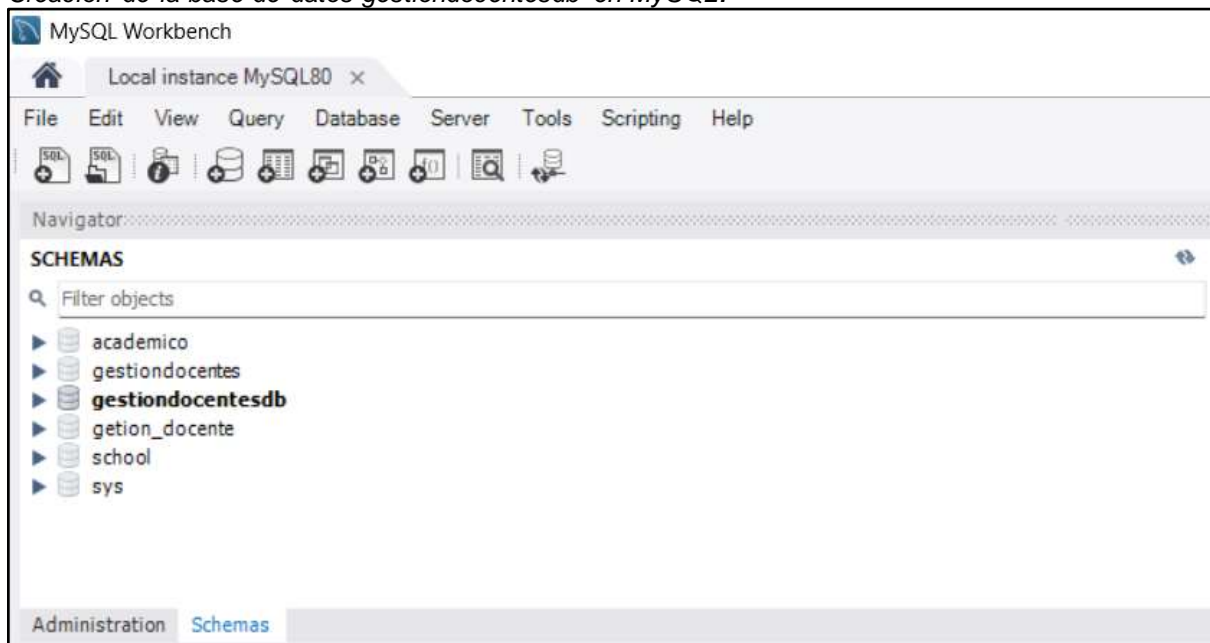


Figura 6

Creación de la base de datos gestiondocentesdb en MySQL.



PASO 4.

Se crea el paquete **Models** dentro de **com.ddleon.gestiondocentes**, y dentro de **Models** se crea la clase **Teacher** y se tiene el siguiente código que se muestra en la figura 7.

Figura 7

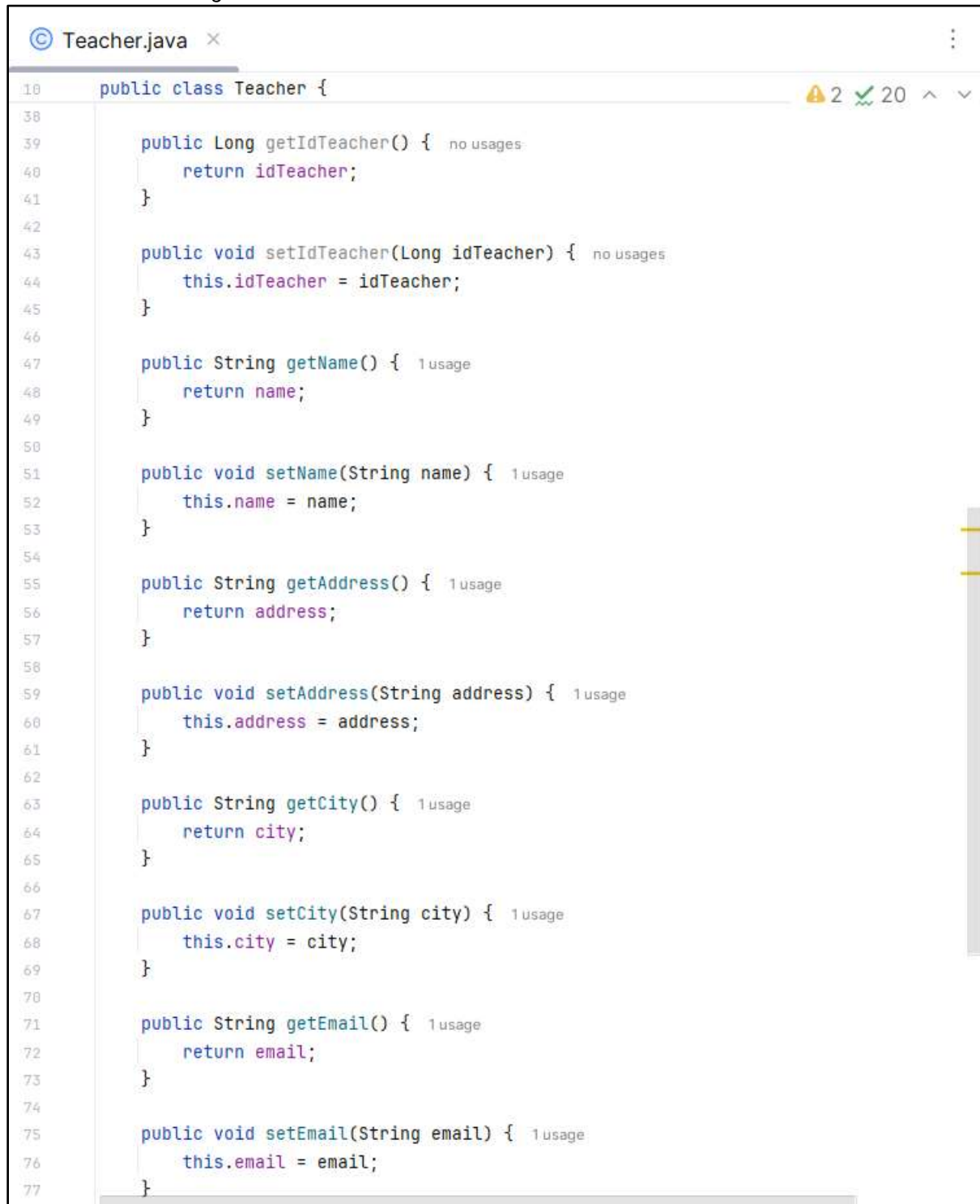
Código de la clase Teacher.



```
1 package com.ddleon.gestiondocentes.Models;
2
3 import jakarta.persistence.*;
4 import jakarta.validation.constraints.*;
5
6 import java.time.LocalDate;
7
8 @Entity 26 usages
9 @Table(name="teacher")
10 public class Teacher {
11
12     @Id 2 usages
13     @GeneratedValue(strategy = GenerationType.IDENTITY)
14     private Long idTeacher;
15
16     @NotBlank(message = "El nombre del docente es obligatorio") 2 usages
17     private String name;
18
19     @NotBlank(message = "La dirección del docente es obligatoria") 2 usages
20     private String address;
21
22     @NotBlank(message = "La ciudad del docente es obligatoria") 2 usages
23     private String city;
24
25     @NotBlank(message = "El email del docente es obligatorio") 2 usages
26     @Email(message = "El formato del email es inválido")
27     private String email;
28
29     @NotNull(message = "La fecha de nacimiento es obligatoria") 2 usages
30     @Past(message = "La fecha de nacimiento debe ser anterior a la fecha actual")
31     private LocalDate birthday;
32
33     @Min(value = 0, message = "El tiempo de servicio no puede ser negativo") 2 usages
34     private int serviceTime;
35
36     // Getters and Setters
37
38
39     public Long getIdTeacher() { no usages
40         return idTeacher;
41     }
```

Figura 8

Continuación del código de la clase Teacher.



The image shows a screenshot of a code editor window titled "Teacher.java". The code is for a Java class named "Teacher". The editor has a line number margin on the left, a status bar at the top right showing 2 warnings and 20 checks, and a vertical scrollbar on the right. The code defines several methods for a teacher object, including getters and setters for idTeacher, name, address, city, and email. Each method is annotated with its usage count in the IDE.

```
18 public class Teacher {
38
39     public Long getIdTeacher() { no usages
40         return idTeacher;
41     }
42
43     public void setIdTeacher(Long idTeacher) { no usages
44         this.idTeacher = idTeacher;
45     }
46
47     public String getName() { 1 usage
48         return name;
49     }
50
51     public void setName(String name) { 1 usage
52         this.name = name;
53     }
54
55     public String getAddress() { 1 usage
56         return address;
57     }
58
59     public void setAddress(String address) { 1 usage
60         this.address = address;
61     }
62
63     public String getCity() { 1 usage
64         return city;
65     }
66
67     public void setCity(String city) { 1 usage
68         this.city = city;
69     }
70
71     public String getEmail() { 1 usage
72         return email;
73     }
74
75     public void setEmail(String email) { 1 usage
76         this.email = email;
77     }
}
```


Figura 9

Continuación del código de la clase Teacher.

```
78
79     public LocalDate getBirthday() { 3 usages
80         return birthday;
81     }
82
83     public void setBirthday(LocalDate birthday) { 1 usage
84         this.birthday = birthday;
85     }
86
87     public int getServiceTime() { 1 usage
88         return serviceTime;
89     }
90
91     public void setServiceTime(int serviceTime) { 1 usage
92         this.serviceTime = serviceTime;
93     }
94 }
95
```

PASO 5.

Se crea el paquete **Repositories** dentro de **com.ddleon.gestiondocentes**, y dentro de **Repositories** se crea la clase **TeacherRepository** y se tiene el siguiente código que se muestra en la figura 10.

Figura 10

Código de la clase Teacher.

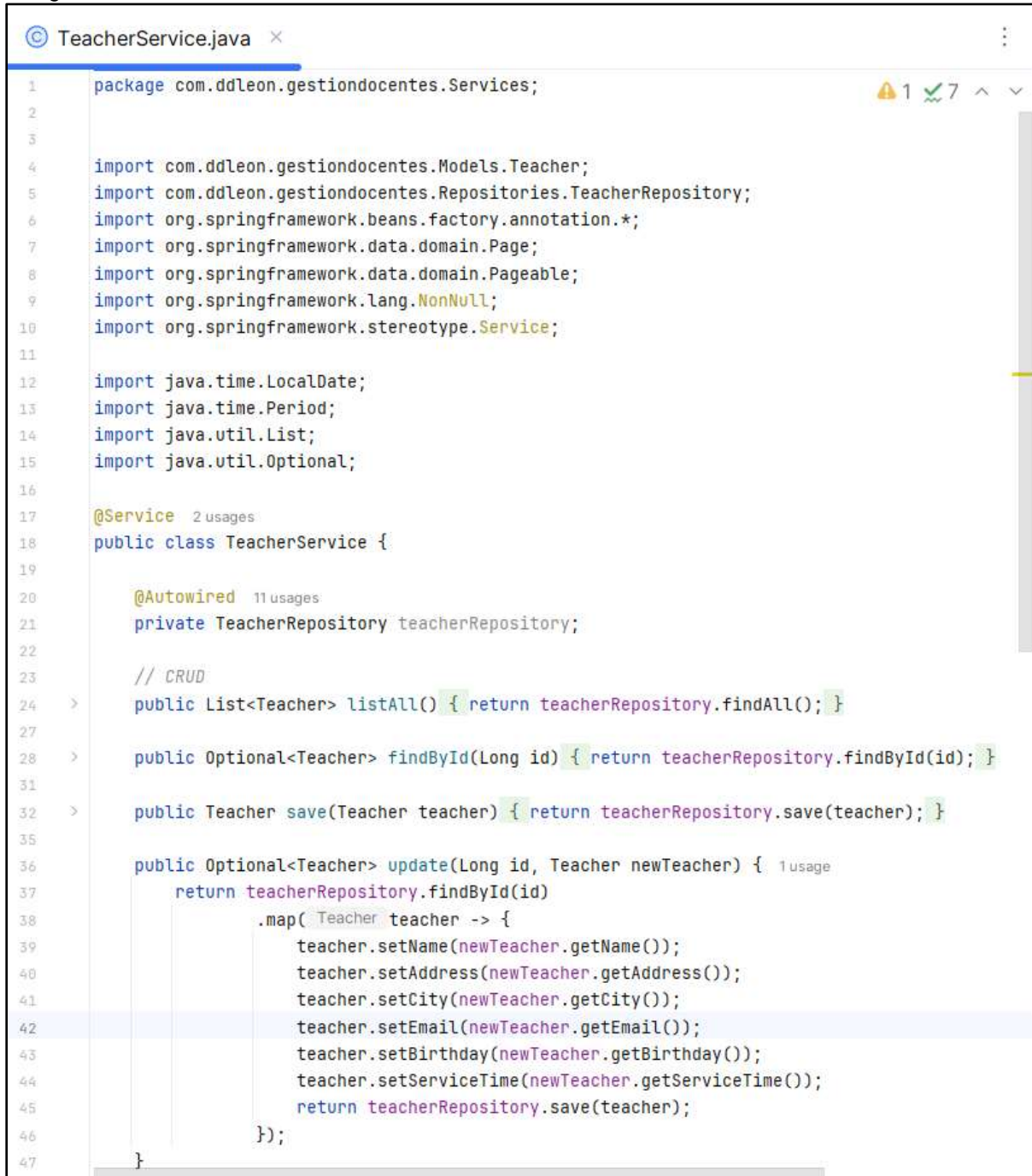
```
TeacherRepository.java x
1 package com.ddleon.gestiondocentes.Repositories;
2
3 import com.ddleon.gestiondocentes.Models.Teacher;
4 import org.springframework.data.jpa.repository.JpaRepository;
5
6 import java.util.List;
7
8 public interface TeacherRepository extends JpaRepository <Teacher, Long> { 2 usages
9
10     List<Teacher> findByCity(String city); 1 usage
11
12     List<Teacher> findByServiceTimeGreaterThanOrEqualTo(int years); 1 usage
13
14     // Page<Teacher> findAll(@NonNull Pageable pageable);
15 }
16
```

PASO 6.

Se crea el paquete **Services** dentro de **com.ddleon.gestiondocentes**, y dentro de **Services** se crea la clase **TeacherService** y se tiene el siguiente código que se muestra en la figura 11.

Figura 11

Código de la clase TeacherService.



```
1 package com.ddleon.gestiondocentes.Services;
2
3
4 import com.ddleon.gestiondocentes.Models.Teacher;
5 import com.ddleon.gestiondocentes.Repositories.TeacherRepository;
6 import org.springframework.beans.factory.annotation.*;
7 import org.springframework.data.domain.Page;
8 import org.springframework.data.domain.Pageable;
9 import org.springframework.lang.NonNull;
10 import org.springframework.stereotype.Service;
11
12 import java.time.LocalDate;
13 import java.time.Period;
14 import java.util.List;
15 import java.util.Optional;
16
17 @Service 2 usages
18 public class TeacherService {
19
20     @Autowired 11 usages
21     private TeacherRepository teacherRepository;
22
23     // CRUD
24     > public List<Teacher> listAll() { return teacherRepository.findAll(); }
27
28     > public Optional<Teacher> findById(Long id) { return teacherRepository.findById(id); }
31
32     > public Teacher save(Teacher teacher) { return teacherRepository.save(teacher); }
35
36     public Optional<Teacher> update(Long id, Teacher newTeacher) { 1 usage
37         return teacherRepository.findById(id)
38             .map( Teacher teacher -> {
39                 teacher.setName(newTeacher.getName());
40                 teacher.setAddress(newTeacher.getAddress());
41                 teacher.setCity(newTeacher.getCity());
42                 teacher.setEmail(newTeacher.getEmail());
43                 teacher.setBirthday(newTeacher.getBirthday());
44                 teacher.setServiceTime(newTeacher.getServiceTime());
45                 return teacherRepository.save(teacher);
46             });
47     }
```

Figura 12

Continuación del código de la clase TeacherService.

```
48
49 ~ public boolean delete(Long id) { !usage
50 ~     if (teacherRepository.existsById(id)) {
51         teacherRepository.deleteById(id);
52         return true;
53     }
54     return false;
55 }
56
57 // Buscar por ciudad
58 > public List<Teacher> findByCity(String city) { return teacherRepository.findByCity(city); }
59
60 // Buscar por experiencia mínima
61 ~ public List<Teacher> findByServiceTimeGreaterThanOrEqual(int years) { !usage
62     return teacherRepository.findByServiceTimeGreaterThanOrEqual(years);
63 }
64
65 // Edad promedio
66 ~ public double calculateAverageAge() { !usage
67     List<Teacher> teachers = teacherRepository.findAll();
68     if (teachers.isEmpty()) return 0.0;
69
70     int agesum = 0;
71     for (Teacher d : teachers) {
72         if (d.getBirthDay() != null) {
73             agesum += Period.between(d.getBirthDay(), LocalDate.now()).getYears();
74         }
75     }
76     return (double) agesum / teachers.size();
77 }
78
79 // Paginación
80 > public Page<Teacher> listPaged(@NonNull Pageable pageable) { return teacherRepository.findAll(pageable); }
81
82 }
```

PASO 7.

Se crea el paquete **Exceptions** dentro de **com.ddleon.gestiondocentes**, y dentro de **Exceptions** se crea la clase **GlobalExceptionHandler** y se tiene el siguiente código que se muestra en la figura

Figura 13

Código de la clase GlobalExceptionHandler.

```
GlobalExceptionHandler.java x
1 package com.ddleon.gestiondocentes.Exceptions;
2
3
4 import io.swagger.v3.oas.annotations.Hidden;
5 import org.springframework.http.HttpStatus;
6 import org.springframework.http.ResponseEntity;
7 import org.springframework.web.bind.MethodArgumentNotValidException;
8 import org.springframework.web.bind.annotation.*;
9 import org.springframework.web.context.request.WebRequest;
10
11 import java.time.LocalDateTime;
12 import java.util.HashMap;
13 import java.util.Map;
14
15 @Hidden no usages
16 @ControllerAdvice
17 public class GlobalExceptionHandler {
18
19     // Validaciones con @Valid
20     @ExceptionHandler(MethodArgumentNotValidException.class) no usages
21     @RequestMapping
22     public ResponseEntity<Object> handleValidationErrors(MethodArgumentNotValidException ex, WebRequest request) {
23         Map<String, Object> errorBody = new HashMap<>();
24         errorBody.put("timestamp", LocalDateTime.now());
25         errorBody.put("status", HttpStatus.BAD_REQUEST.value());
26         errorBody.put("error", "Validación fallida");
27
28         Map<String, String> fieldErrors = new HashMap<>();
29         ex.getBindingResult().getFieldErrors().forEach( FieldError error ->
30             fieldErrors.put(error.getField(), error.getDefaultMessage()));
31         errorBody.put("message", fieldErrors);
32         errorBody.put("path", request.getDescription( includeClientInfo: false).replace( target: "uri=", replacement: "**"));
33
34         return new ResponseEntity<>(errorBody, HttpStatus.BAD_REQUEST);
35     }
36 }
```

Figura 14

Continuación del código de la clase GlobalExceptionHandler.

```
34 // Otros errores generales
35
36 @ExceptionHandler(Exception.class) no usages
37 @RequestMapping
38 public ResponseEntity<Object> handleGeneralException(Exception ex, WebRequest request) {
39     Map<String, Object> errorBody = new HashMap<>();
40     errorBody.put("timestamp", LocalDateTime.now());
41     errorBody.put("status", HttpStatus.INTERNAL_SERVER_ERROR.value());
42     errorBody.put("error", "Error interno");
43     errorBody.put("message", ex.getMessage());
44     errorBody.put("path", request.getDescription( includeClientInfo: false).replace( target: "uri=", replacement: "**"));
45
46     return new ResponseEntity<>(errorBody, HttpStatus.INTERNAL_SERVER_ERROR);
47 }
48 }
```

EJECUCIÓN DEL PROYECTO

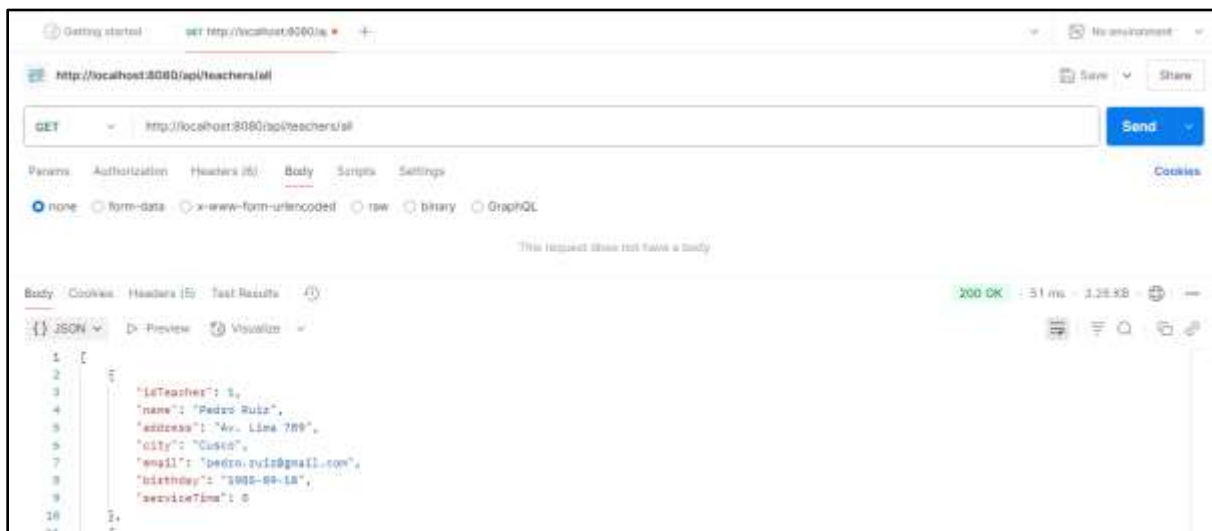
Detalla el proceso seguido para la validación y prueba de los endpoints del proyecto mediante la herramienta Postman. El objetivo principal fue asegurar el correcto funcionamiento de los servicios REST desarrollados, verificando tanto la estructura de las peticiones como las respuestas esperadas

PASO 1.

El primer paso en el proceso de validación de la API consistió en realizar una solicitud HTTP GET al endpoint ***http://localhost:8080/api/teachers/all***, cuya función principal es devolver el listado completo de docentes registrados en el sistema.

Figura 15

Listar los docentes en Postman



PASO 2.

El segundo paso consistió en probar la capacidad del sistema para recuperar los datos de un docente determinado, utilizando su identificador único. Para ello, se realizó una solicitud HTTP GET al endpoint ***http://localhost:8080/api/teachers/{id}***, donde {id} se reemplaza con el valor específico asociado al docente deseado.

Figura 16

Obtiene un docente por ID en Postman

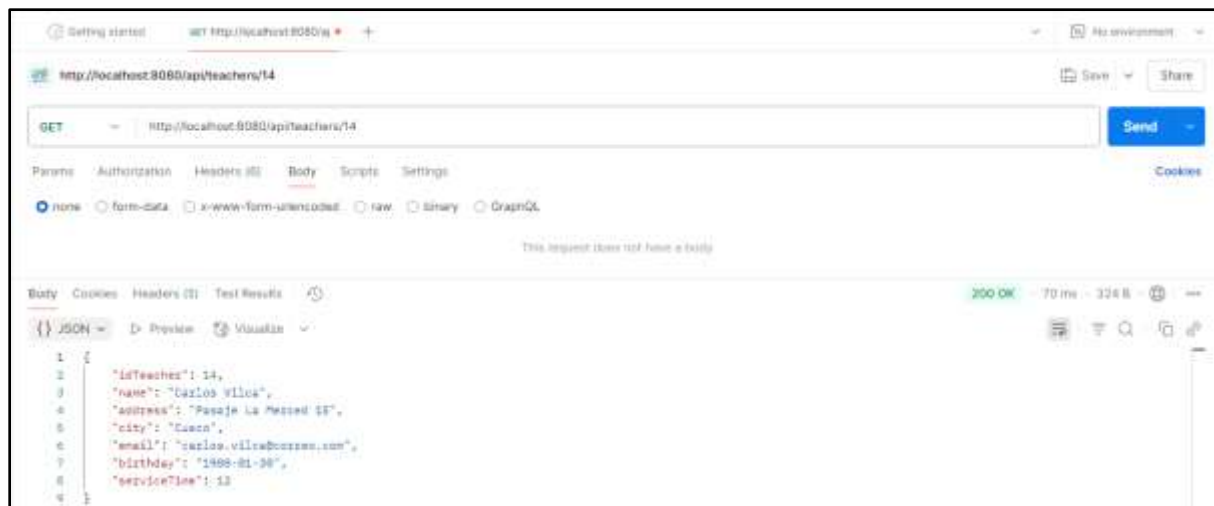


Figura 17

Obtiene un docente por ID en localhost



PASO 3.

Se procedió a realizar una solicitud HTTP POST al endpoint

`http://localhost:8080/api/teachers`. Desde Postman, se configuró la petición seleccionando el método correspondiente, y se definió el encabezado Content-Type con el valor `application/json`, indicando que el cuerpo del mensaje sería enviado en formato JSON.

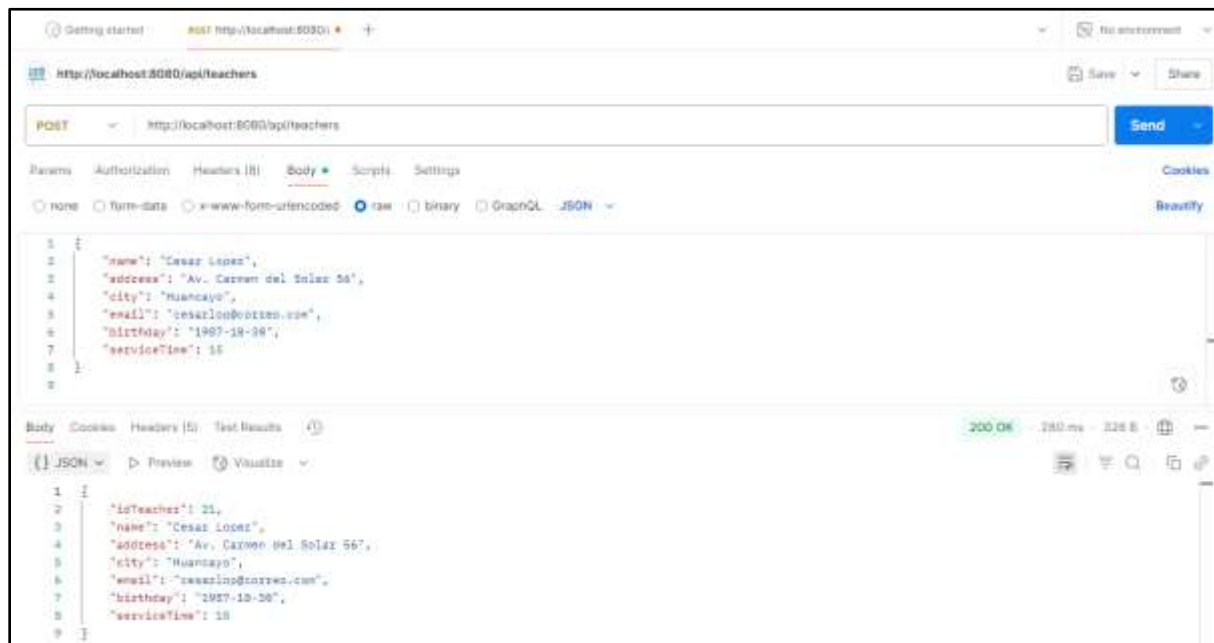
En el cuerpo de la solicitud, se incluyó un objeto representativo del nuevo docente a registrar, como el siguiente:

```
{
  "name": "Cesar Lopez",
  "address": "Av. Carmen del Solar 56",
  "city": "Huancayo",
  "email": "cesarlop@correo.com",
  "birthday": "1987-10-30",
  "serviceTime": 15
}
```


En la consola de Postman, también se evidenció que la inserción fue exitosa, mostrando los datos del nuevo docente en formato JSON, incluidos su nombre, especialidad y correo electrónico.

Figura 18

Crear un nuevo docente en Postman



PASO 4.

El paso siguiente fue una solicitud HTTP PUT hacia el endpoint

http://localhost:8080/api/teachers/{id}, donde {id} representa el identificador único del docente a modificar.

Desde Postman, se configuró la petición seleccionando el método PUT y reemplazando el parámetro de ruta por un ID válido; por ejemplo:

http://localhost:8080/api/teachers/21. Para ello se ingresa los datos a cambiar

Figura 19

Actualizar los datos de los docentes en Postman

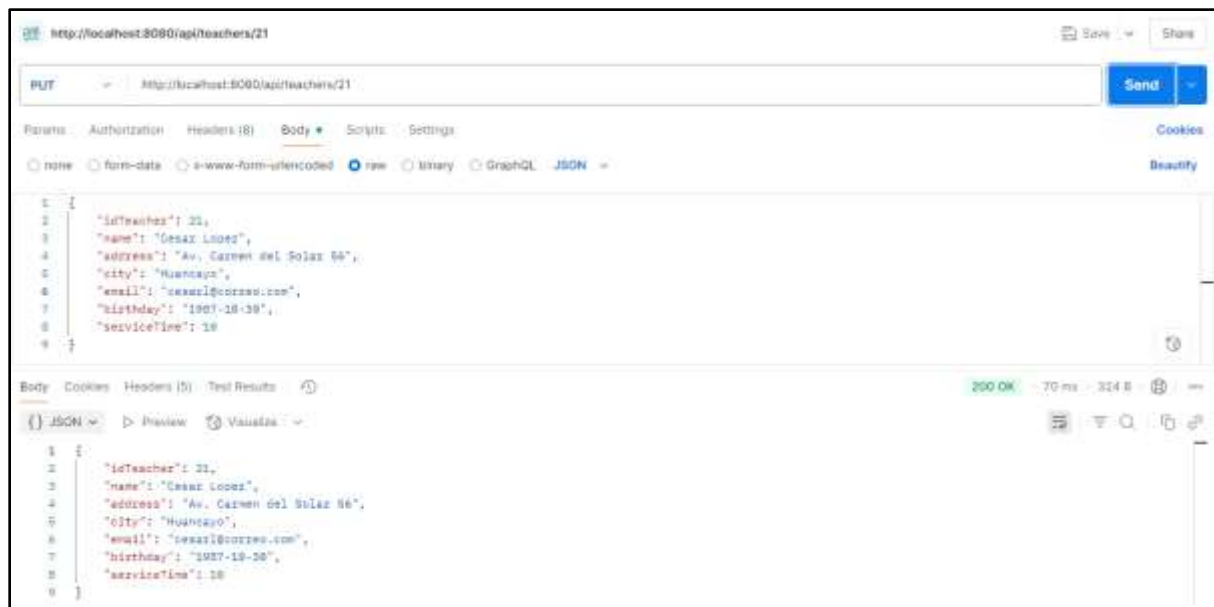


Figura 20

Verificación en el localhost



PASO 5.

El método HTTP DELETE sobre el endpoint

http://localhost:8080/api/teachers/{id}, donde {id} corresponde al identificador del docente que se desea eliminar.

Desde Postman, se configuró la petición reemplazando {id} con un valor real, como por ejemplo: ***http://localhost:8080/api/teachers/21***

Figura 21

Eliminar a uno de los docentes en Postman

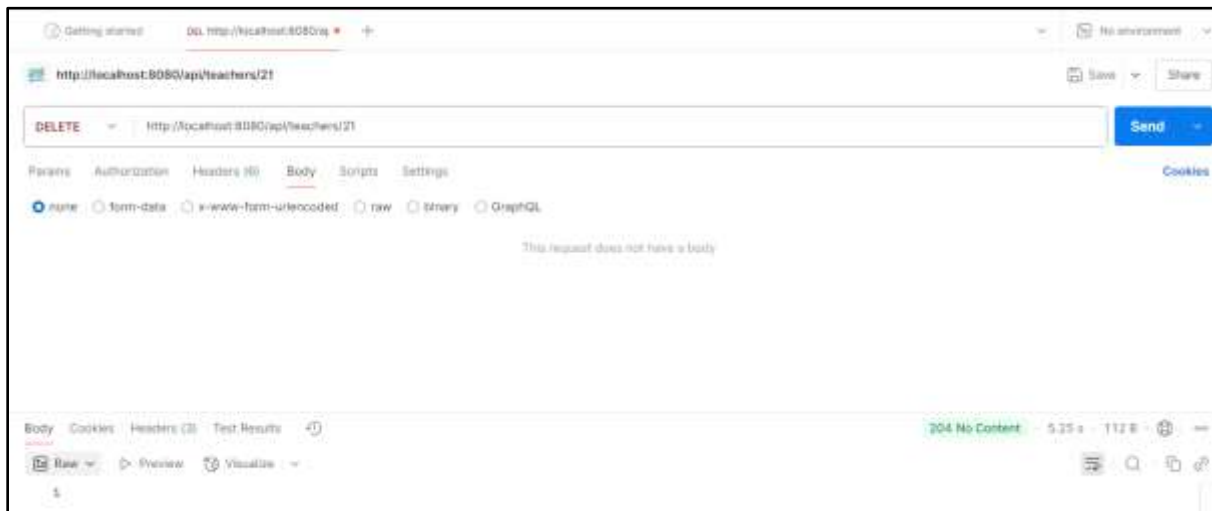
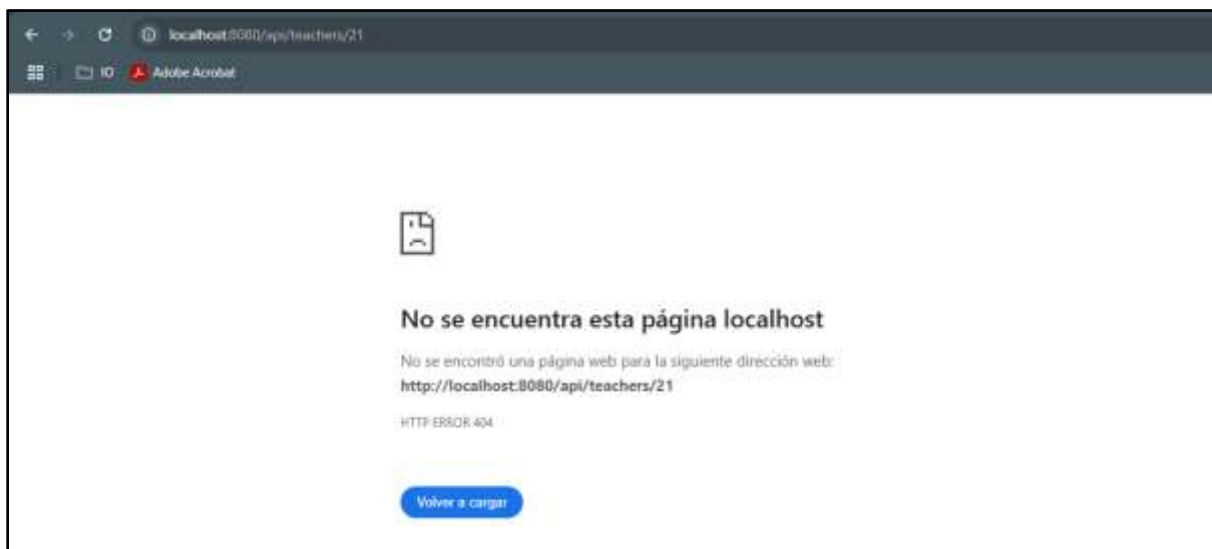


Figura 22

Verificación en el localhost



PASO 6.

El sistema proporciona un endpoint que permite obtener una lista filtrada de docentes en función de su ciudad de residencia, utilizando la ruta GET ***http://localhost:8080/api/teachers/city/{city}***. Esta funcionalidad resulta especialmente útil cuando se necesita segmentar la información por ubicación geográfica, ya sea para reportes institucionales, análisis regionales o asignaciones académicas.

Durante las pruebas, se realizó una solicitud dirigida a ***http://localhost:8080/api/teachers/city/Trujillo***

Figura 23

Listar todos los docentes que residen en una ciudad específica en Postman



Figura 24

Verificación en el localhost



PASO 7.

Para consultar los docentes que cuentan con una cantidad mínima de años de experiencia, se utiliza el endpoint GET

`http://localhost:8080/api/teachers/experience/{years}`, el cual permite filtrar los resultados en función del número de años de servicio especificado. Durante la prueba realizada, se utilizó como ejemplo el valor 10, accediendo a la ruta

`http://localhost:8080/api/teachers/experience/10`

Figura 25

Listar todos los docentes que tienen una cierta cantidad de años en Postman

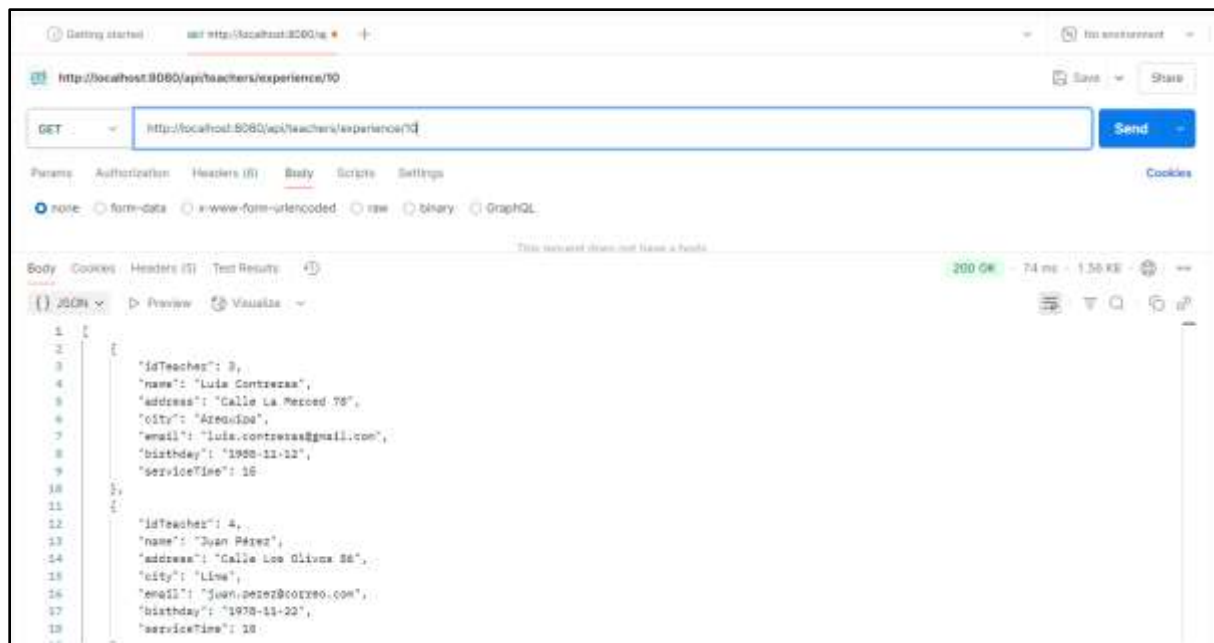


Figura 26

Verificación en el localhost



PASO 8.

Para calcular y obtener la edad promedio de todos los docentes registrados en el sistema, se emplea el endpoint: GET **`http://localhost:8080/api/teachers/average-age`**.

Esta operación permite acceder a un valor numérico que representa la media aritmética de las edades de todos los docentes almacenados en la base de datos. Durante el proceso de prueba, se configuró desde Postman una solicitud con el método GET apuntando directamente a la ruta mencionada.

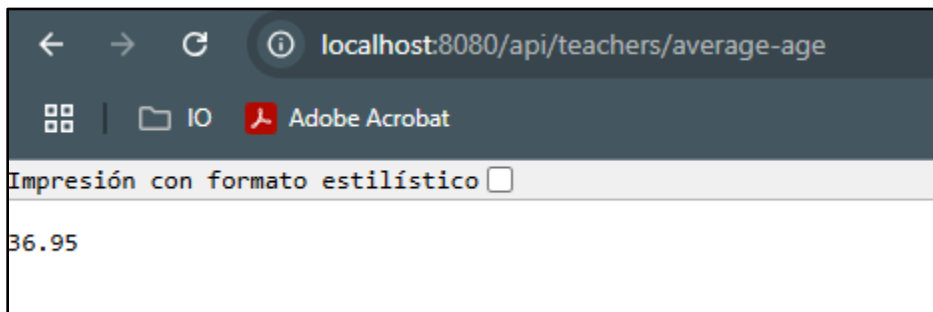
Figura 27

La edad promedio de todos los docentes registrados en Postman



Figura 28

Verificación en el localhost



PASO 9.

Swagger es una herramienta de documentación interactiva que permite describir, visualizar y probar APIs REST de forma clara y estructurada. Su propósito principal es facilitar la comprensión y el uso de una API tanto para desarrolladores como para usuarios técnicos, proporcionando una interfaz gráfica donde se pueden explorar los endpoints disponibles, sus métodos HTTP, parámetros requeridos, respuestas esperadas y códigos de estado.

<http://localhost:8080/swagger-ui/index.html>

En ella se puede observar, por ejemplo, el controlador teacher-controller, que agrupa operaciones como GET, PUT y DELETE sobre el recurso /api/teachers/{id}. Cada operación está codificada por colores (azul para GET, amarillo para PUT, rojo para DELETE), lo que facilita su identificación visual.

Figura 29

Documentación Swagger

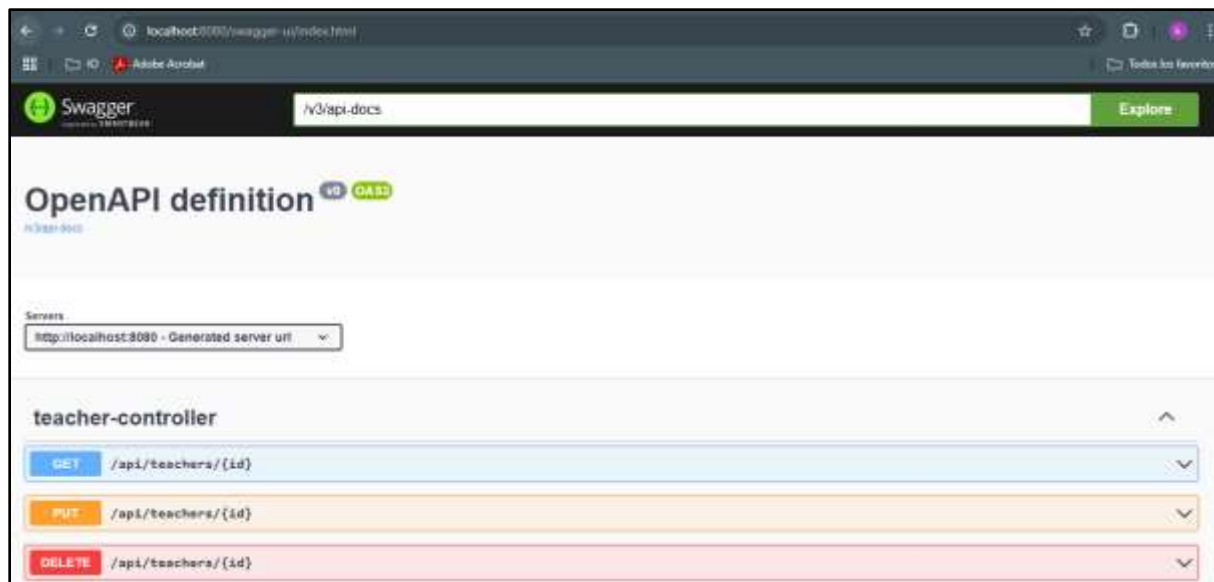


Figura 30

Get api/teachers/ {20} en Swagger



Figura 31

Get api/teachers/ {20} en Swagger Response

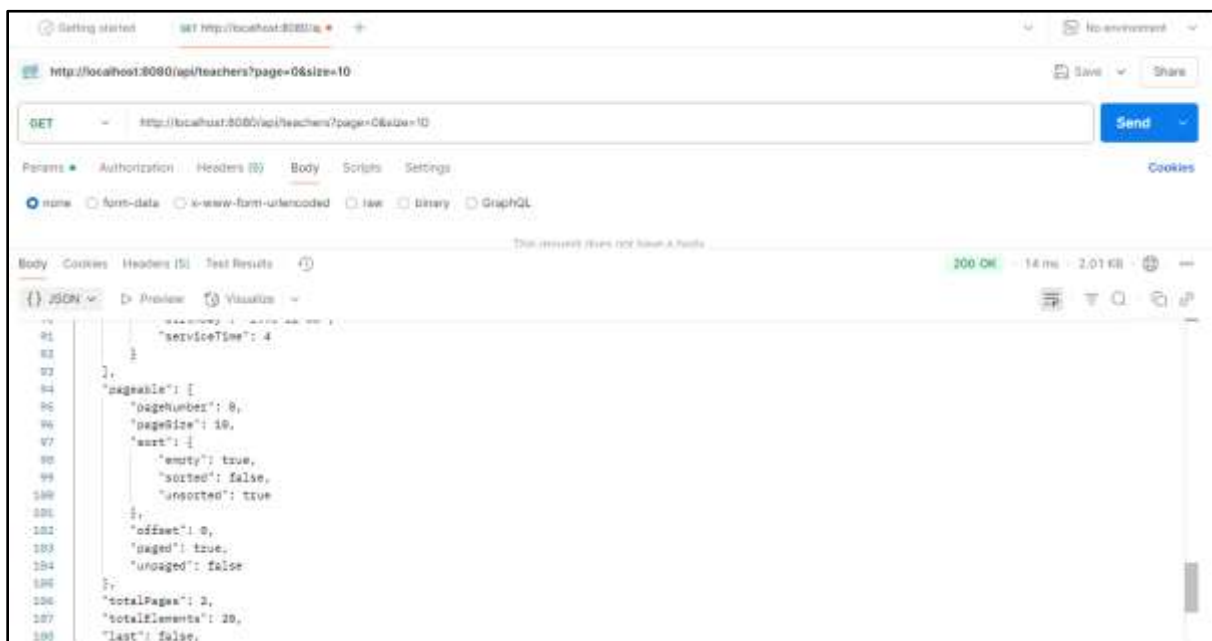


PASO 10.

Para implementar paginación en el listado de docentes, el sistema expone el endpoint GET ***http://localhost:8080/api/teachers?page=0&size=10***, donde *page=0* corresponde a la primera página (recordando que el índice es cero basado) y *size=10* indica que se devolverán diez docentes por página. Esta funcionalidad es especialmente útil cuando el volumen de datos es considerable, ya que permite dividir los resultados en bloques manejables y optimiza tanto el rendimiento como la experiencia del usuario.

Figura 32

Paginación en el listado de docentes en Postman



PASO 11.

Además de verificar que los endpoints funcionen correctamente cuando se reciben datos válidos, también es fundamental comprobar que el sistema responda adecuadamente ante solicitudes incorrectas o que incumplen las reglas de validación. Esto asegura que la API sea robusta, confiable y capaz de proteger la integridad de la información almacenada.

Durante esta prueba, se intentó crear un nuevo docente a través del endpoint POST ***http://localhost:8080/api/teachers***, pero ingresando datos que violaban varias restricciones del modelo. En el cuerpo de la solicitud se incluyeron los siguientes errores:

-
- ***Un campo birthday con una fecha futura: "2025-10-30"***
- ***Un valor negativo en el campo serviceTime: -10***
- ***Un formato inválido en el campo email: "correo.invalido.com"***

Figura 33

Verificación de Validaciones mínimas en Postman

