🧩 UT2 – Servicios Web

Módulo: Programación de Servicios y Procesos (RA4)

> **Resultado de aprendizaje 4:** Desarrolla aplicaciones que ofrecen servicios en red, utilizando librerías de clases y aplicando criterios de eficiencia y disponibilidad.

- **a)** Identificar los principales **protocolos estándar de comunicación** (HTTP, HTTPS, WebSocket, FTP...) y su uso en servicios en red.
- **b)** Comprender las **ventajas de emplear protocolos estándar**, como la interoperabilidad, seguridad y compatibilidad entre aplicaciones.
- **c)** Analizar las **librerías y frameworks** que permiten implementar servicios en red (Spring Boot, Jakarta EE, etc.).
- **d) Desarrollar y probar servicios** que ofrezcan comunicación en red (REST).
- **e) Utilizar clientes** de comunicación (Postman, clientes HTTP, aplicaciones Java) para verificar el funcionamiento de los servicios.
- **f)** Implementar mecanismos de **comunicación simultánea** que permitan atender varios clientes al mismo tiempo.
- **g) Comprobar la disponibilidad** y correcto funcionamiento de los servicios desarrollados.
- **h) Depurar y documentar** adecuadamente las aplicaciones creadas, siguiendo buenas prácticas de programación y presentación técnica.

1. Protocolos estándar en la comunicación en red

Las **aplicaciones y servicios web modernos** funcionan sobre la **capa de aplicación** del modelo TCP/IP, que es la encargada de definir cómo los programas se comunican entre sí a través de la red. En esta capa se encuentran los **protocolos de más alto nivel**, como **HTTP** y **HTTPS**, que permiten el intercambio de información entre clientes y servidores. Cuando un usuario accede a una página web o una aplicación envía datos a un servidor (por ejemplo, una API REST), esa comunicación se realiza mediante peticiones y respuestas HTTP, que viajan sobre una conexión TCP.

Por tanto, comprender el funcionamiento de **HTTP/HTTPS** es esencial para desarrollar servicios web eficientes, seguros y compatibles con otros sistemas.

El **HTTP (HyperText Transfer Protocol)** es un protocolo de la **capa de aplicación** del modelo TCP/IP. Se utiliza para la comunicación entre **clientes y servidores web**, enviando y recibiendo información en formato texto (normalmente HTML, JSON o XML).

El **HTTPS** es la versión segura de HTTP, que incorpora un canal cifrado mediante **TLS/SSL** para proteger los datos transmitidos (confidencialidad y autenticidad).

🔹 Características generales

- Funciona sobre el **protocolo TCP** (puerto 80 para HTTP y 443 para HTTPS).
- Es un protocolo **sin estado (stateless)**: cada petición es independiente.
- Utiliza un formato **petición–respuesta**:
  - El **cliente** (navegador o aplicación) envía una **solicitud HTTP**.
  - El **servidor** procesa la solicitud y devuelve una **respuesta HTTP** con un código de estado.

🔹 Métodos principales de HTTP

Método	Descripción	Uso típico
GET	Solicita información del servidor.**No modifica** los datos.	Consultar una lista o un recurso. Ejemplo: `GET /alumnos`
POST	Envía datos al servidor para **crear** un nuevo recurso. Usado en submits o envios de datos de un formulario desde paginas web.	Registrar un nuevo alumno, enviar un formulario. Ejemplo: `POST /alumnos`
PUT	Envía datos al servidor para **actualizar completamente** un recurso existente.	Reemplazar todos los datos de un alumno. Ejemplo: `PUT /alumnos/3`
PATCH	Actualiza **parcialmente** un recurso (no todos sus campos).	Cambiar solo el correo de un alumno. Ejemplo: `PATCH /alumnos/3`
DELETE	Solicita la**eliminación** de un recurso del servidor.	Eliminar un registro. Ejemplo: `DELETE /alumnos/3`

🔹 **Códigos de estado HTTP más comunes**

Cuando un servidor responde a peticiones HTTP envia en la HTTP Response un codigo.

Código	Significado	Descripción
200 OK	Éxito	La petición se procesó correctamente.
201 Created	Recurso creado	Un nuevo recurso fue creado (normalmente con POST).
204 No Content	Sin contenido	La operación se realizó pero no hay datos que devolver.
400 Bad Request	Petición incorrecta	Error de sintaxis o datos no válidos.
401 Unauthorized	No autorizado	Falta autenticación o token válido.
403 Forbidden	Prohibido	El cliente no tiene permiso para acceder al recurso.
404 Not Found	No encontrado	El recurso solicitado no existe.
500 Internal Server Error	Error interno	El servidor falló al procesar la petición.

2. Arquitectura de software para servicios web

A continuacion se describen arquitecturas y patrones utilizados en la construccion de aplicaciones basadas en servicios web.

🧩 **Arquitectura Frontend-Backend**

Frontend

Es la parte de una aplicación o página web que ve y con la que interactúa el usuario, es decir, la **interfaz**. Incluye el diseño, los botones, formularios, colores, animaciones… Se desarrolla con tecnologías como HTML, CSS y JavaScript, Typescript, Angular...

Backend

Es la parte interna y oculta que **gestiona los datos, la lógica y la comunicación con la base de datos**. Se encarga de **procesar peticiones**, autenticación, seguridad.

El backend **suele estar desplegado en un servidor** (Ejemplo: Tomcat), que es un ordenador remoto disponible en Internet. Ese servidor ejecuta la lógica de la aplicación y se conecta a una base de datos (BD) donde se guardan los datos de los usuarios, productos, etc.

Cuando el frontend necesita información (por ejemplo, iniciar sesión o mostrar una lista), envía una petición al backend, este consulta o actualiza la base de datos y devuelve la respuesta al frontend.

🧩 Patrón MVC (Modelo–Vista–Controlador)

- **Modelo**: gestiona los datos. Ejemplo: Clases en el backend que representan entidades (Empresa, Persona, Alumno...etc)
- **Vista**: interfaz o representación (en servicios web, intercambian datos en formato JSON/XML entre interfaz y backend).
- **Controlador**: gestiona las peticiones que se reciben en los endpoints y las procesa, generando respuestas (en formato JSON/XML).

Endpoints

Un endpoint es una URL específica de una API donde el frontend u otro cliente puede enviar peticiones al backend para obtener o modificar datos. Ejemplo:

- https://miapp.com/api/usuarios/5
- Ese endpoint podría devolver los datos del usuario con ID 5.

En una arquitectura MVC del backend, el Controlador es quien gestiona los endpoints. Cuando llega una petición a un endpoint, el controlador la recibe, la procesa y decide qué devolver.

⚙️ Patrón Servicio

- Capa intermedia que contiene la lógica de negocio.
- Facilita la reutilización y la separación de responsabilidades.

🗂️ Patrón DAO (Data Access Object)

- Capa encargada del acceso a datos (consultas, persistencia).
- Permite cambiar la fuente de datos sin modificar la lógica del servicio.

🔁 Relación entre MVC, Servicio y DAO

Cliente → Controlador → Servicio → DAO → Base de datos

El uso combinado de los patrones **MVC**, **Servicio** y **DAO** aporta una estructura clara, modular y escalable al desarrollo de aplicaciones y servicios web. Cada capa tiene una responsabilidad específica, lo que mejora la organización, el mantenimiento y la calidad del código.

✅ **Ventajas generales**

- **Separación de responsabilidades:** cada capa (presentación, lógica, datos) cumple una función concreta.
- **Código más limpio y mantenible:** los cambios en una parte del sistema no afectan a las demás.
- **Reutilización de componentes:** controladores, servicios y DAOs pueden reaprovecharse en distintos proyectos o módulos.
- **Facilidad de pruebas:** se pueden probar de forma independiente las capas (test unitarios de servicios o DAOs).
- **Escalabilidad:** la aplicación puede crecer añadiendo nuevas funciones sin perder coherencia.
- **Trabajo en equipo más eficiente:** distintos desarrolladores pueden encargarse de capas diferentes.
- **Adaptabilidad tecnológica:** permite cambiar la base de datos o la interfaz sin reescribir toda la aplicación.

En conjunto, estos patrones garantizan un **diseño profesional, modular y robusto**, fundamental para construir servicios web modernos y mantenibles.

3. Librerías y frameworks para servicios web

- Spring Boot: estructura de proyecto, `@RestController`, `@Service`, `@Repository`.
- Jakarta EE (referencia): contexto histórico y diferencias.
- Librerías HTTP en Java (`HttpClient`, `HttpServer`), más bajo nivel.

Spring Boot es un framework moderno que facilita la creación de aplicaciones web y servicios REST en Java. Forma parte del ecosistema **Spring Framework**, y su objetivo principal es **reducir la configuración** necesaria para poner en marcha un proyecto.

🔹 **Spring Boot: Características principales**

- **Estructura MVC clara del proyecto:** Organiza las clases en paquetes como `controller`, `service`, `repository/dao`, `model`, siguiendo el patrón MVC.
- **Anotaciones principales:**
  - `@RestController`: expone endpoints REST.
  - `@Service`: define la lógica de negocio.
  - `@Repository`: gestiona el acceso a datos (DAO).
  - `@Autowired`: inyección automática de dependencias.
- **Servidor de aplicaciones integrado (Tomcat/Jetty):** no requiere desplegar en un servidor externo, al arrancar la aplicacion se despliega automaticamente en el servidor.
- **Gestión automática de dependencias** mediante Maven o Gradle.
- **Soporte para JSON, seguridad, validación, pruebas y documentación** (Swagger, Spring Security, etc.).
- **Compatible con APIs REST y WebSockets.**

💡 **¿Por qué se usa tanto Spring Boot?**

1. **Simplicidad:** permite crear un servicio web funcional en minutos.
2. **Productividad:** automatiza tareas repetitivas y evita configuraciones manuales.

3. **Modularidad:** incluye numerosos *starters* (web, data, security, mail, websocket…) que facilita la implementacion de funcionalidades avanzadas.
4. **Compatibilidad:** se integra fácilmente con bases de datos, APIs externas y servicios cloud.
5. **Estandarización:** se ha convertido en el **estándar de facto** en el desarrollo web empresarial moderno en Java.

4. Servicio Web REST / RESTful

Un servicio web **REST** es una API que sigue el estilo arquitectónico **REST (Representational State Transfer)** para comunicar aplicaciones usando **HTTP**.

Principios básicos

- Basado en **recursos** identificados por **URL**
- **Stateless**: el servidor no guarda estado del cliente
- Uso de **métodos HTTP**:
  - GET → Leer
  - POST → Crear
  - PUT/PATCH → Actualizar
  - DELETE → Eliminar
- Formato ligero: normalmente **JSON**

Ejemplo de recurso

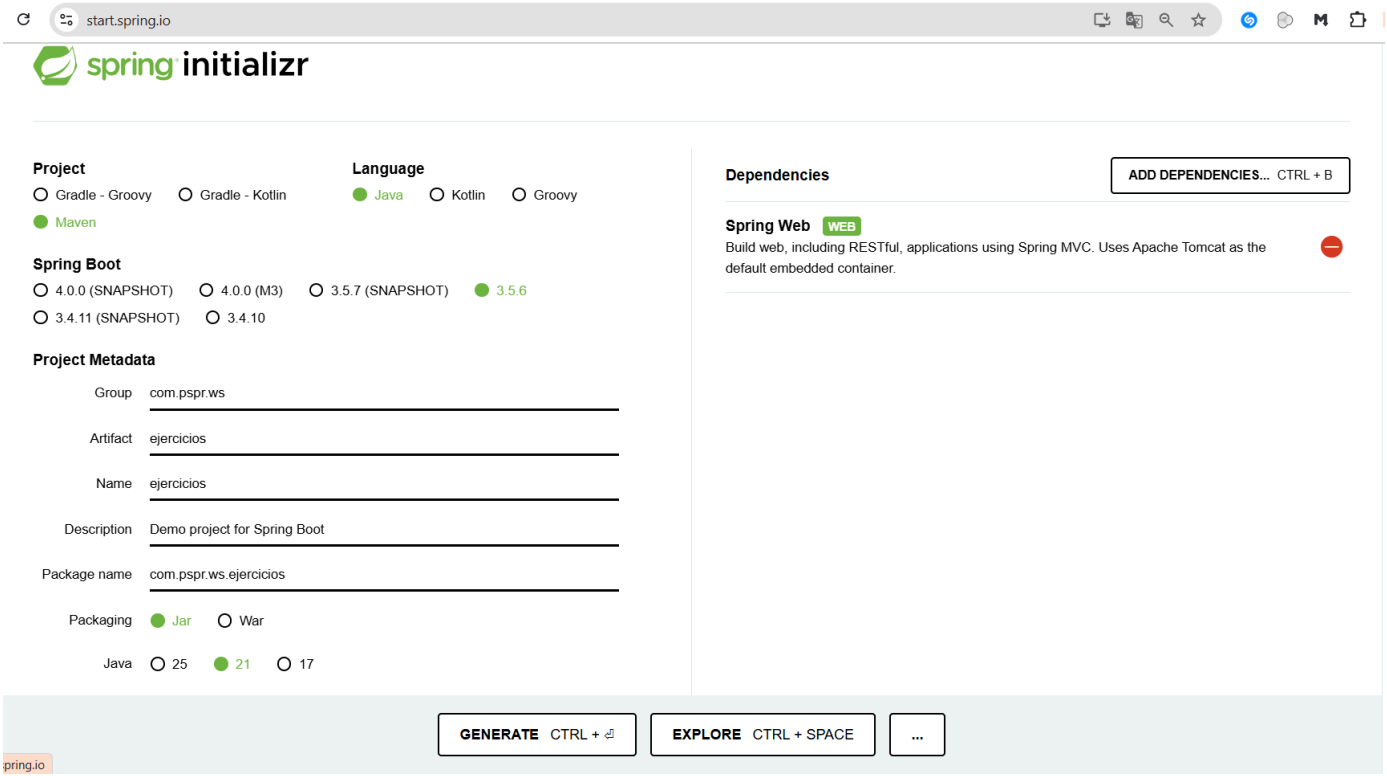
`GET /api/usuarios` → Obtener usuarios

4. Creación de un servicio web REST con Spring Boot

Objetivo: desarrollar una API CRUD sencilla.

Pasos:

1. Generar el esqueleto del proyecto desde: https://start.spring.io/
2. Dependencias básicas:
   - **Spring Web** (aplicaciones RESTful/MVC): `spring-boot-starter-web .`



3. Organizar la estructura de paquetes:

com.pspr.ws ├── controller ├── service ├── dao ├── model ├── util └── Application.java

1. Prueba que maven compila el proyecto sin errores. ▷ `mvn clean compile`
2. Copia las clases `EmpresaDAO, Empresa, ConexionBD` del ejercicio de ADT que usa conexion JDBC con la base de datos MySQL.
3. Agrega la **dependencia JDBC** en el pom.xml

```
<dependency>
      <groupId>com.mysql</groupId>
      <artifactId>mysql-connector-j</artifactId>
      <version>8.1.0</version>
</dependency>
```

Implementacion de la capa de servicio

La clase de servicio, implementa la logica. Normalmente es la que se comunica con la capa de acceso a datos (dao o repository).

```
public class EmpresaService {
    private EmpresaDAO empresaDAO;

    // Inicializar el DAO recibiendo la conexión
    public EmpresaService(Connection conexion) {
        this.empresaDAO = new EmpresaDAO(conexion);
    }

    // listar todas las empresas
    public List<Empresa> listarEmpresas() {
```

```
        try {
            return empresaDAO.listar();
        } catch (SQLException e) {
            throw new RuntimeException("Error al listar empresas", e);
        }
    }

    // Buscar empresa por ID
    public Empresa buscarPorId(int id) {
        try {
            return empresaDAO.buscarPorId(id);
        } catch (SQLException e) {
            throw new RuntimeException("Error al buscar empresa", e);
        }
    }

    //TODO: Insertar nueva empresa
    public void insertarEmpresa(Empresa empresa) {
      ...
    }

    //TODO: Actualizar empresa existente
    public void actualizarEmpresa(Empresa empresa) {
        ...
    }

    // Eliminar empresa
    public void eliminarEmpresa(int id) {
        ...
    }
}
```

Implementacion del Controlador

El controlador recibe las peticiones HTTP e invoca a los metodos de la clase servicio, para procesarlas. Recoge los resultados y los devuelve al cliente.

Fijate en las anotaciones. @GetMapping, @PostMapping, @PutMapping, @DeleteMapping. Se relacionan con los method HTTP vistos (GET, POST,PUT, DELETE).

```
@RestController
@RequestMapping("/empresas")
public class EmpresaController {

    private EmpresaService empresaService;

    public EmpresaController() {
        // TODO: Crear conexión manualmente (sin @Autowired)
        Connection conexion = ConexionBD.conectar();
        this.empresaService = new EmpresaService(conexion);
    }

    // GET - Listar todas las empresas
    @GetMapping
    public List<Empresa> listar() {
        return empresaService.listarEmpresas();
    }
```

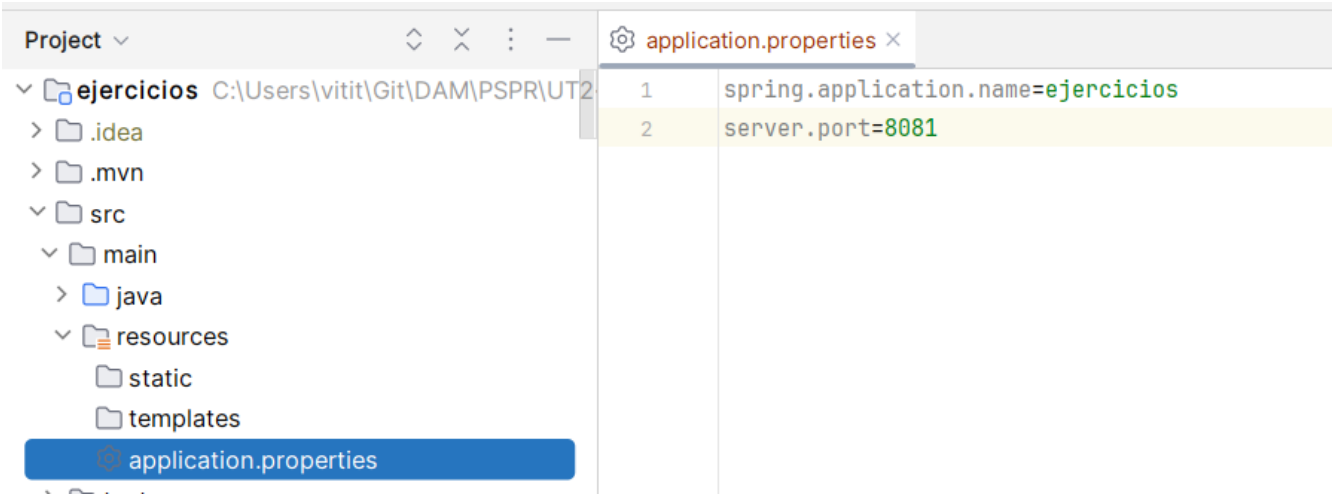
```
    // GET - Buscar empresa por ID
    @GetMapping("/{id}")
    public Empresa buscarPorId(@PathVariable int id) {
        return empresaService.buscarPorId(id);
    }

    // POST - Insertar nueva empresa
    @PostMapping
    public String insertar(@RequestBody Empresa empresa) {
        empresaService.insertarEmpresa(empresa);
        return "Empresa creada correctamente.";
    }

    // PUT - Actualizar empresa existente indicando el id y los datos
    @PutMapping("/{id}")
    public String actualizar(@PathVariable int id, @RequestBody Empresa empresa) {
        empresa.setIdEmpresa(id);
        empresaService.actualizarEmpresa(empresa);
        return "Empresa actualizada correctamente.";
    }

    // DELETE - Eliminar empresa indicando el id
    @DeleteMapping("/{id}")
    public String eliminar(@PathVariable int id) {
        empresaService.eliminarEmpresa(id);
        return "Empresa eliminada correctamente.";
    }
}
```

Compilar y "arrancar" el servicio web REST

1. **Compilacion e instalacion** de la aplicación:
   - Cada vez que realices **cambios en el pom.xml** conviene realizar Boton derecho sobre el proyecto > **Maven > Reload Project**
   - Comprueba que maven limpia y puede instalar el proyecto (incluye compilacion, tests y empaquetado en .jar) :
     - ▷ `mvn clean install`
2. **Ejecutar/Debuguear la aplicacion**:
   i. **Opcion 1**: desde consola ▷ `mvn spring-boot:run` . Esta tarea no pertenece al lifecycle de maven, es una tarea del plugin (extensiones de maven) spring-boot-maven-plugin declarada en el pom.xml
   ii. **Opcion 2**: boton derecho > Run o Debug sobre la clase Application.java (la que tiene el main())
   iii. Una vez arrancado el proyecto estará desplegado en el servidor **Tomcat** embebido de Spring, escuchando peticiones en el puerto 8080.
   iv. Abre un navegador: http://localhost:8080/empresas
   v. ebes ver un JSON con el listado de empresas. Por ejemplo:


```
[
{"idEmpresa":1,"nombre":"Tech Solutions","sector":"Software","telefono":"984123456","email":"info@techsolutions
{"idEmpresa":2,"nombre":"Radiolab","sector":"Sanidad","telefono":"985987654","email":"contacto@radiolab.es","di
]
```

3. **Cambio del puerto del servidor Tomcat**. Si lo deseamos podemos cambiar el puerto en la propiedad server.port:



📝 Ejercicio 1:

Implementar un recurso /alumnos de manera analoga a /empresas con los endpoints :

- `GET /alumnos`
- `POST /alumnos`
- `PUT /alumnos/{id}`
- `DELETE /alumnos/{id}`

Clases a implementar:

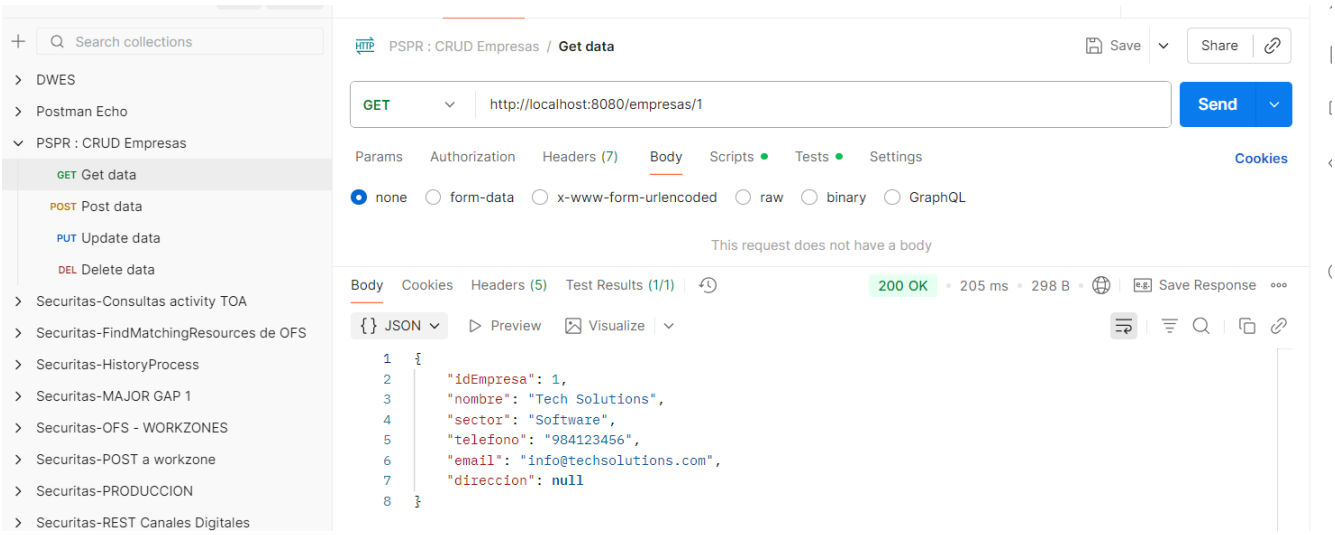
- controller/AlumnoController
- service/AlumnoService
- dao/AlumnoDAO
- model/Alumno

Pruebas del servicio

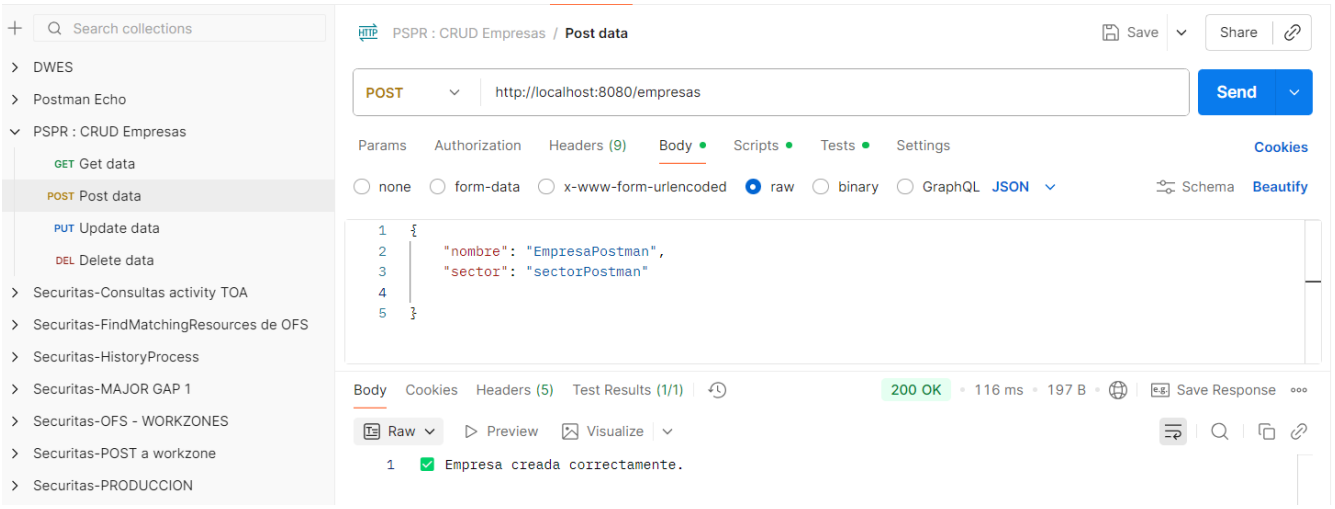
- Uso de la aplicación **Postman** https://www.postman.com/downloads/ para probar los endpoints.

Ejemplo de pruebas desde postman sobre el servicio:

- **GET /empresas/{id}** (Consulta una empresa).



- **POST /empresas/** (Crea una empresa).



📝 Ejercicio 2:

Desde postman incorpora pruebas para el recurso /alumnos

- `GET /alumnos`
- `POST /alumnos`
- `PUT /alumnos/{id}`
- `DELETE /alumnos/{id}`

Comunicación simultánea y tiempo real

Spring Boot se apoya en **Tomcat** (u otro servidor embebido) para gestionar concurrencia:

- Cada **petición HTTP** se atiende por un **hilo del pool de Tomcat**.
- Si hay más peticiones que hilos, se **ponen en cola** hasta que quede uno libre.
- Spring **no crea hilos manualmente** en controladores REST: el servidor web los gestiona.

Demostracion de gestion de la concurrencia

Crea un proyecto externo cliente. Simularemos la miles de peticiones simultaneas a un recurso con el siguiente codigo:

```
public class Main {

   static long tiempoTotal = 0;

    public static synchronized void incrementarTiempoTotal (long duracionHilo){
        tiempoTotal += duracionHilo;
    }

    public static void main(String[] args) {
        // TODO Simular 1000 peticiones simultaneas usando hilos y medir tiempo medio de respuesta //
        final int TOTAL_PETICIONES = 1000;
        List<Thread> hilos = new ArrayList<>();
        for (int i = 0; i < TOTAL_PETICIONES; i++) {
            int id = i;
            hilos.add(new Thread(() -> {
                long inicio = System.currentTimeMillis();
                try {
                    URL url = new URL("http://localhost:8080/empresas");
                    HttpURLConnection conn = (HttpURLConnection) url.openConnection();
                    conn.setRequestMethod("GET");
                    int codigo = conn.getResponseCode();
                    System.out.println("Hilo " + id + " → Respuesta: " + codigo);
                } catch (Exception e) {
                    System.err.println("Error en hilo " + id + " "+ e.getMessage());
                }
                long fin = System.currentTimeMillis();
                long duracionHilo = fin - inicio;
                incrementarTiempoTotal(duracionHilo);
            }));
        }
        for (Thread hilo: hilos){
            hilo.start();
        }
        for (Thread hilo: hilos){
            try {
                hilo.join();
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
        System.out.println("Tiempo medio de peticion: " + tiempoTotal/TOTAL_PETICIONES + "mseg");

        }
    }
```

En el servicio web, en el controlador empresas/ vamos a mostrar la informacion sobre hilos creados.

```
    @GetMapping
    public List<Empresa> listar() {
        String hiloActual = Thread.currentThread().getName();
        System.out.println("Hilo iniciado:  " + hiloActual);
        long inicio = System.currentTimeMillis();
        List<Empresa> empresas = empresaService.listarEmpresas();
        long fin = System.currentTimeMillis();
        long duracion = fin - inicio;
```

```
        System.out.println("Fin Hilo:  " + hiloActual + " duracion= " + duracion + "ms");
        return empresas;
    }
```

Parámetros de concurrencia en Tomcat

Si tu aplicación usa Tomcat embebido (por defecto en Spring Boot), puedes limitar el número de hilos del pool que atienden peticiones HTTP simultáneas.

En el fichero application.properties o application.yml:

```
# Número máximo de hilos de Tomcat
server.tomcat.threads.max=20

# Número mínimo de hilos que mantiene activos Tomcat
server.tomcat.threads.min-spare=5

# Número máximo de conexiones simultáneas (pendientes de atender)
server.tomcat.max-connections=100

# Número máximo de peticiones que pueden esperar en cola
server.tomcat.accept-count=50
```

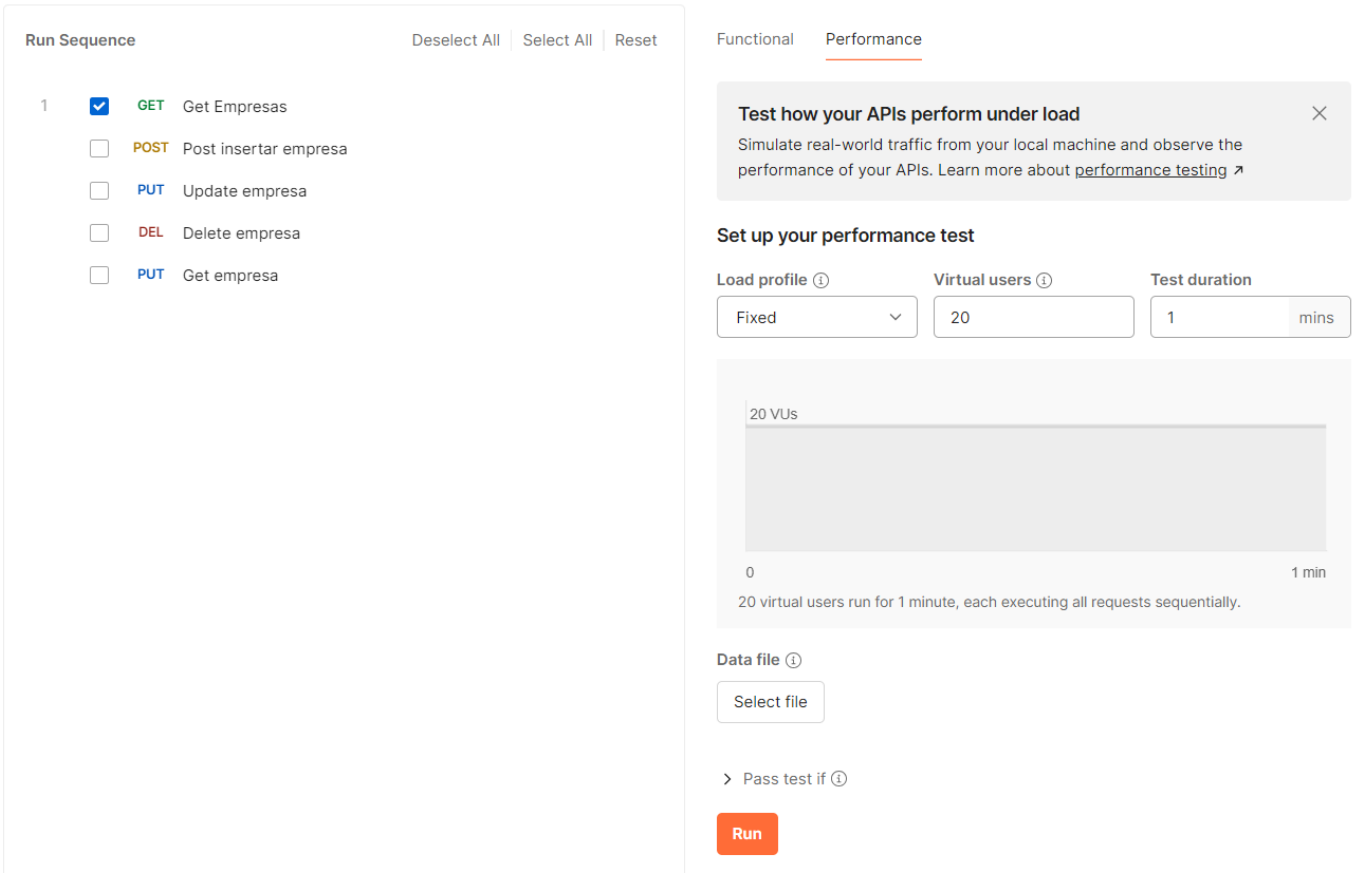
- **server.tomcat.threads.max**
  Máx. hilos atendiendo peticiones.
  → Subir si hay muchas peticiones simultáneas.
- **server.tomcat.threads.min-spare**
  Hilos “en espera”.
  → Subir si la app tarda en responder al aumentar la carga.
- **server.tomcat.max-connections**
  Máx. conexiones TCP activas.
  → Subir si se conectan muchos clientes a la vez.
- **server.tomcat.accept-count**
  Cola cuando no hay hilos libres.
  → Subir si aparecen errores 503 o peticiones rechazadas.

Verificación de disponibilidad y eficiencia

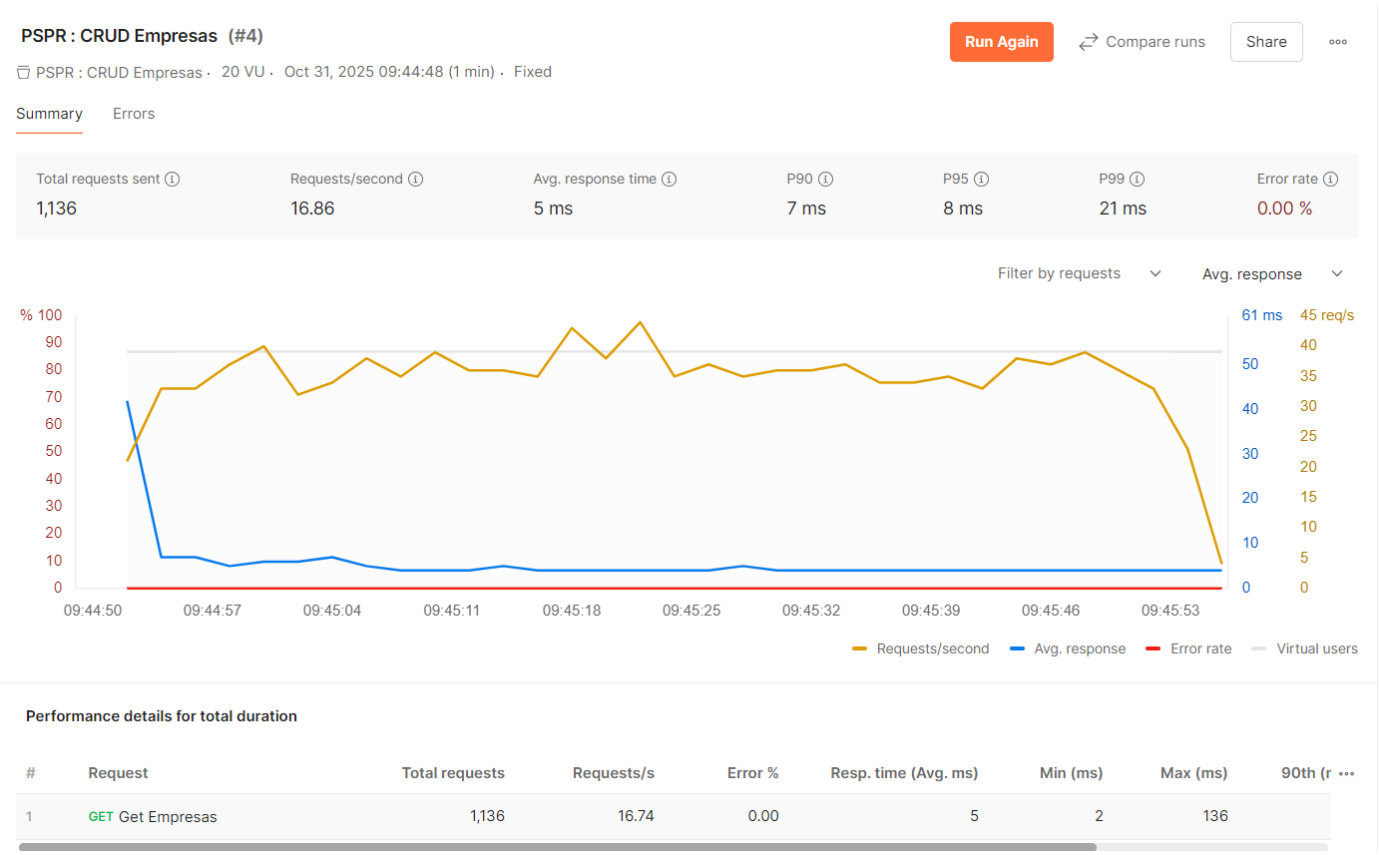
- Pruebas de carga con Postman Runner o JMeter.
- Logs de servidor, códigos de estado.

Realiza una prueba de carga desde Postman para evaluar el rendimiento. **Run collection**

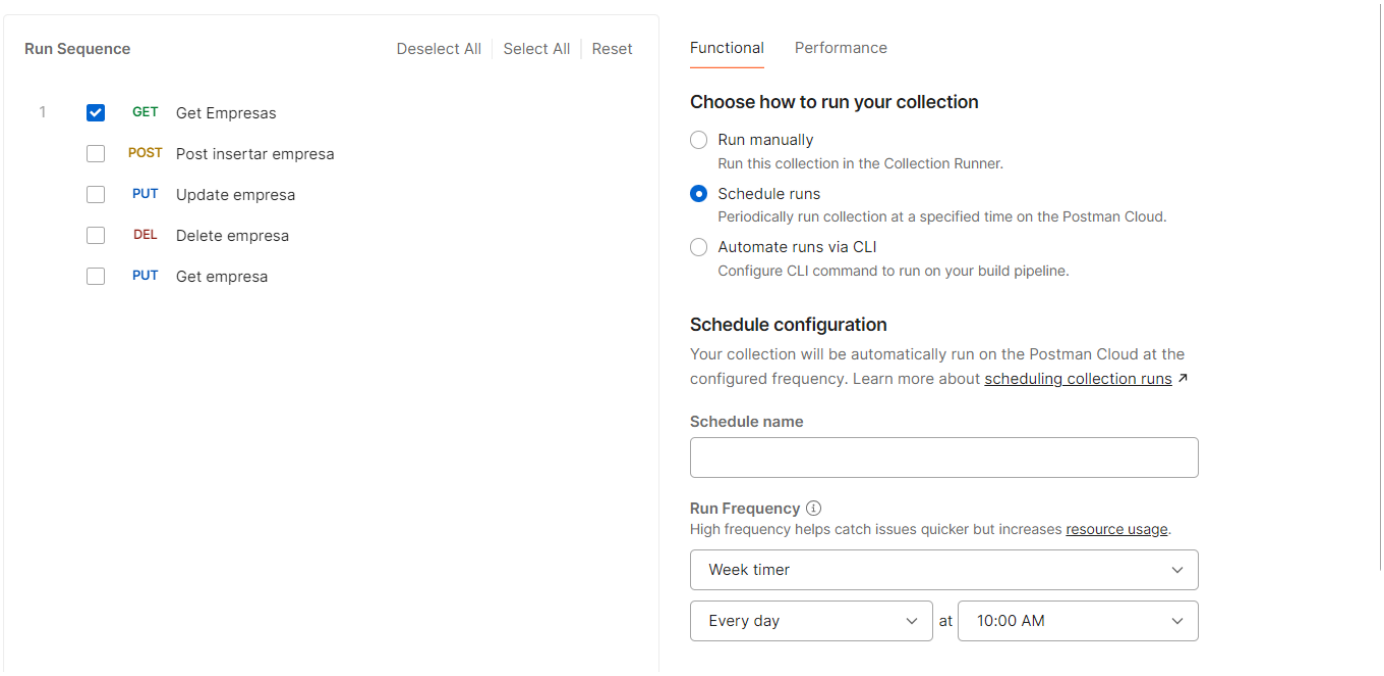
Sobre el recurso GET empresas/. Configuramos la prueba que simula 20 usuarios simultaneos solicitando el recurso indicado durante 1 min.



Se mide tasa de errores y tiempo medio de respuesta:



Disponibilidad: Estas pruebas se pueden planificar en un calendario para comprobar periodicamente la disponibilidad.

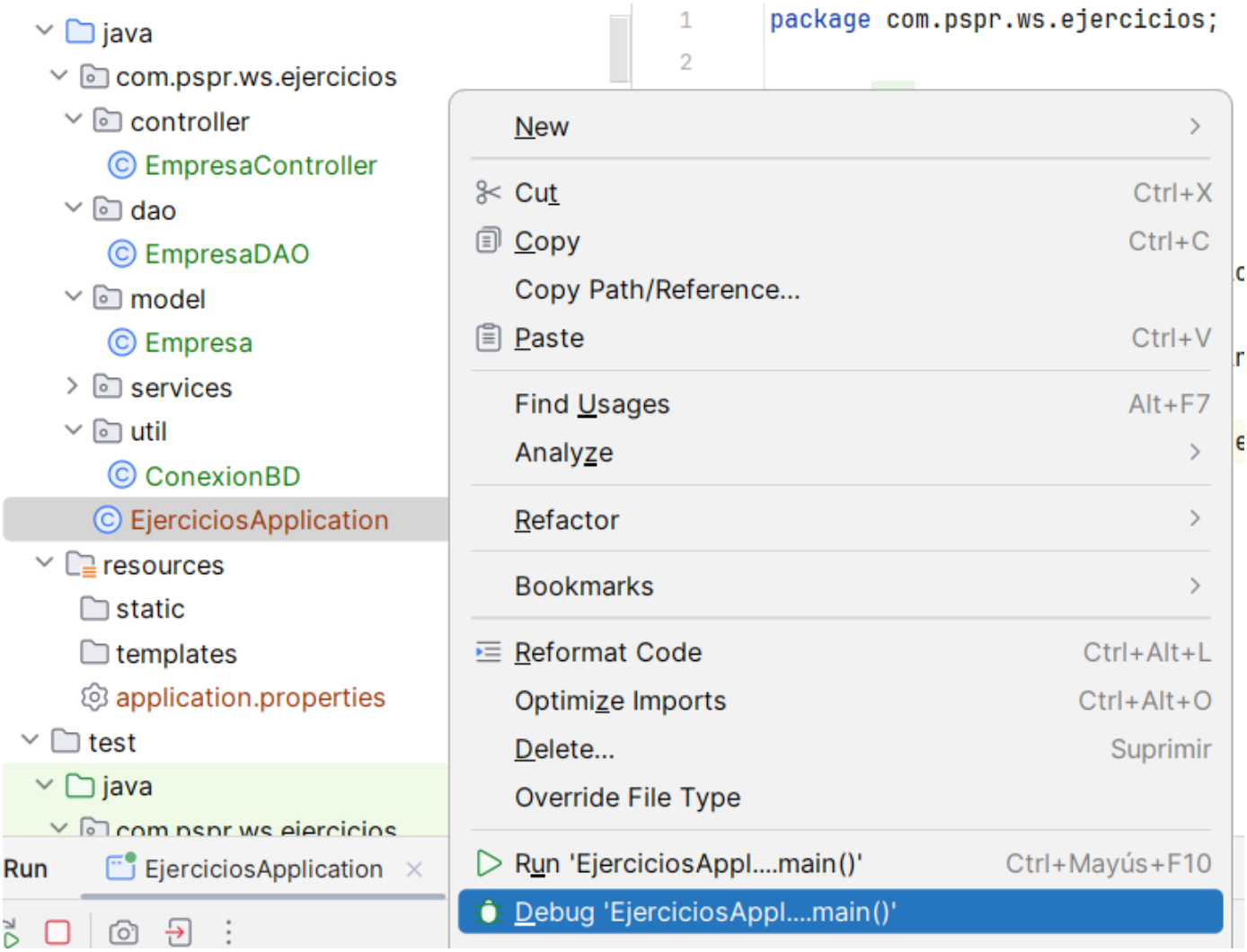


Depuración y documentación

- Depuración con IntelliJ y logs.
- Documentación automática con **Swagger/OpenAPI**.

Depuración

Para la depuracion arrancamos el servicio en modo debug. De esta manera cuando recibamos peticiones podemos añadir breakpoints en el codigo.



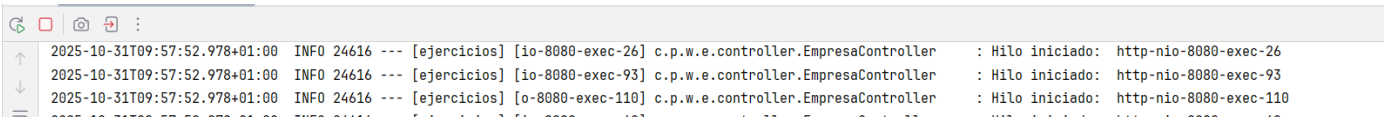
Logs

Spring Boot trae integrado **Logback** + **SLF4J**, por lo que **no hace falta añadir dependencias**.

Se usa creación de logger por clase:

```
// declaramos el log sobre la clase que queremos "logear"
private static final Logger log =
            LoggerFactory.getLogger(EmpresaController.class);
...
log.info("mensaje informativo"); // en vez de usar system.out.println..
log.error("mensajes de error"); // se suele usar en catch de Excepciones
log.debug("mensajes que solo apareceran en modo debug");
```

Ejemplo de mensaje log:



Niveles de log

- **TRACE** → máximo detalle
- **DEBUG** → depuración técnica
- **INFO** → información general del proceso (por defecto)
- **WARN** → advertencias
- **ERROR** → errores graves
- **OFF** → desactivar logs

Configurar nivel de log en **application.properties**

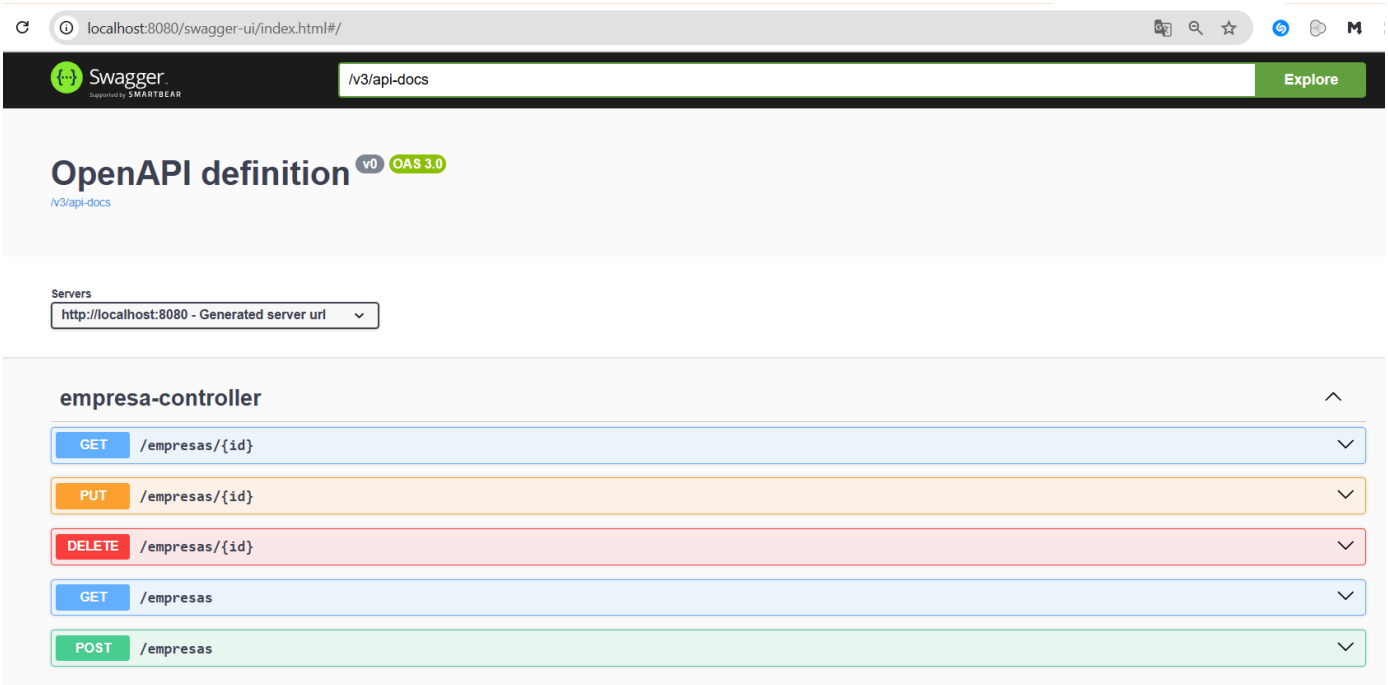
```
logging.level.root=INFO
logging.level.com.miapp=INFO

(⚠️ Cambia com.miapp por tu paquete, de esta manera
se pueden configurar distintos niveles de log en funcion del paquete)
```

Documentacion de API con Swagger/OpenAPI

Incorporamos la siguiente dependencia en el pom.xml

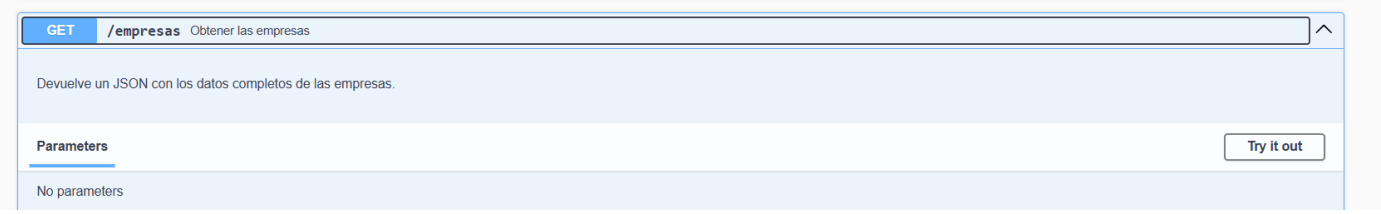
```
<dependency>
        <groupId>org.springdoc</groupId>
        <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
        <version>2.5.0</version>
</dependency>
```

Arrancamos el servicio en la url http://localhost:8080/swagger-ui



Podemos documentar usando anotaciones a nivel de Controller:

```
@Operation(
        summary = "Obtener las empresas",
        description = "Devuelve un JSON con los datos completos de las empresas."
)
@GetMapping
public List<Empresa> listar() {
```



Otras anotaciones básicas para documentar Swagger (OpenAPI)

- **@Operation**
  Documenta el endpoint (summary + description)
- **@Parameter**
  Describe parámetros (Path, Query, Body…)
- **@ApiResponses / @ApiResponse**
  Documenta códigos y mensajes de respuesta
- **@Schema**
  Describe campos y modelos (Entities)
- **@Tag** *(opcional)*
  Agrupa endpoints en categorías en Swagger UI