

Plan de formación Java

....

ÍNDICE

Bloque 1 - Conociendo mejor Java	3
Java moderno	3
Expresiones Lambda	3
Ejercicios	4
Bloque 2 - Patrones de diseño y buenas prácticas	7
Arquitectura hexagonal	7
Principios SOLID	7
Bloque 3 - Herramientas y dependencias	8
Git	8
Conventional commits	9
Postman	11
Maven	11
IA	12
Bloque 4 - Librerías	13
Lombok	13
MapStruct	15
Bloque 5 - SpringBoot	16
JPA Básico	16
SpringBoot en DynamoDB	17
Inyección de dependencias	17
Ciclo de vida de Bean	17
Bloque 6 - SpringWeb básico	17
RestController	17
RequestHeader	17
RequestParam	17
PathVariable	17
ResponseEntity	17
Manejo de excepciones	17
Feign	17
Postman	17
Swagger	18
Bloque 7 - SpringData básico	18
Validación con @Valid	18
Códigos de respuesta	18
Arquitectura por capas	18
Bloque 8 - Testing	18
JUnit	18

Mockito	18
SonarQube	18
Bloque 9 - Docker	18
Docker Desktop	18
Docker Hub	19
Trabajando con contenedores	19
Trabajando con imágenes	19
Docker Compose	19
Docker Cheat Sheet	19
Bloque 10 - Spring Security	19
Bloque 11 - Mensajería aplicada a AWS	19
SQS	19
SNS	19
Bloque 12 - Spring Cloud	19
Balanceador de carga	20
Api Gateway	20
Eureka	20
Config server	20
Bloque 13 - Spring Batch	20
Bloque 14 - Microservicios con Spring cloud	20
Bloque 15 - Proyecto final	21

Bloque 1 - Conociendo mejor Java

Java moderno

Expresiones Lambda

Hasta la versión 8 de Java, no teníamos a nuestra disposición ciertas expresiones que ya eran comunes en lenguajes de programación como JavaScript o Python. Sin embargo, a partir de la versión 8, Java introdujo estas expresiones, marcando un intento de adaptarse al paradigma de programación funcional. Este paradigma difiere de la programación imperativa en su enfoque y en la forma en que aborda los problemas.

El paradigma funcional se basa en un lenguaje matemático formal, y su principal ventaja es su mayor expresividad. Esto significa que podemos lograr los mismos resultados con menos líneas de código, lo que hace que nuestro código sea más elegante y eficiente.

Para entender las expresiones lambda en Java, es importante comprender el concepto de interfaz funcional. Estas son interfaces que tienen un solo método abstracto, aunque pueden contener muchos métodos estáticos o métodos abstractos heredados de la clase Object. Si una interfaz tiene solo un método abstracto, se convierte en una interfaz funcional y puede utilizarse con una expresión lambda.

Una expresión lambda consta de tres partes:

- Parámetros o argumentos: Pueden ir dentro de paréntesis si hay más de uno, y son opcionales si es solo uno.
- Una flecha (->).
- El cuerpo de la expresión, que puede ser una sola expresión o un bloque de código encerrado entre llaves si contiene múltiples sentencias.

Por ejemplo:

Java

```
//Primer ejemplo donde calculamos la longitud del String que pasamos a la
función por parámetro
Function<String,Integer> sizeOf = (String s) -> {
    return s.length();
};
```

```
//Esta sería la versión compacta:
Function<String,Integer> sizeOf = s -> s.length();

//Podemos aplicar estas expresiones a las listas:
List<String> data = Arrays.asList("a","b","c");

data.forEach(letra -> System.out.println(letra)); // Output: a,b,c

//Versión referenciando el método
data.forEach(System.out::print); // Output: a,b,c
```

Ejercicios

Investiga sobre las expresiones que se pueden usar en las listas y realiza los siguientes ejercicios:

Ejemplo

```
Java
//Dada una lista de números, crea una nueva lista que contenga solo
los números pares.
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

//Resultado
// .stream() abre un flujo de trabajo
List<Integer> evenNumbers = numbers.stream()
// Filtra por los números pares y devuelve una lista
    .filter(n -> n % 2 == 0)
// El flujo se convierte en una lista
    .collect(Collectors.toList());
```

1

```
Java
//Dada una lista de cadenas, crea una nueva lista que contenga las
cadenas en mayúsculas.
List<String> strings = Arrays.asList("uno", "dos", "tres", "cuatro");
```

2

Java

```
//Dada una lista de números, calcula la suma de todos los elementos.  
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
```

3

Java

```
//Dada una lista de números, encuentra el número máximo y mínimo.  
List<Integer> numbers = Arrays.asList(5, 3, 9, 2, 8);
```

4

Java

```
//Dada una lista de cadenas, elimina las cadenas duplicadas y obtén  
una lista sin duplicados.  
List<String> strings = Arrays.asList("a", "b", "a", "c", "b");
```

5

Java

```
//Dada una lista de cadenas, cuenta cuántas de ellas contienen la  
letra "a".  
List<String> strings = Arrays.asList("manzana", "banana", "pera",  
"uva", "sandía");
```

6

Java

```
//Dada una lista de cadenas, crea una cadena que contenga todas las  
cadenas concatenadas.  
List<String> strings = Arrays.asList("Hola", "a", "todos", "en",  
"Java");
```

7

Java

```
//Dada una lista de cadenas, ordénalas en orden alfabético.  
List<String> strings = Arrays.asList("zorro", "perro", "gato",  
"elefante", "ratón");
```

8

Java

```
//Dada una lista de números, crea dos listas separadas: una para los  
números pares y otra para los números impares.  
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
```

9

Java

```
//Dada una lista de objetos (por ejemplo, personas con nombres y  
edades), filtra los objetos que sean mayores de 18 años.
```

```
public class Person {  
    private String name;  
    private int age;  
}
```

```
List<Person> people = Arrays.asList(new Person("Alice", 25), new  
Person("Bob", 17), new Person("Charlie", 30));
```

Bloque 2 - Patrones de diseño y buenas prácticas

Arquitectura hexagonal

Arquitectura Hexagonal

Principios SOLID

Los principios SOLID son un conjunto de cinco principios de diseño de software que se utilizan para crear código más mantenible, escalable y fácil de entender. Estos principios fueron propuestos por Robert C. Martin y son ampliamente aceptados en el mundo del desarrollo de software. A continuación, te explicaré cada uno de los principios SOLID de manera simplificada para principiantes:

- **Principio de Responsabilidad Única (SRP):** Una clase debe tener una única razón para cambiar, es decir, debe tener una única responsabilidad en el sistema, lo que facilita la comprensión y el mantenimiento del código.
- **Principio de Abierto/Cerrado (OCP):** Las clases deben estar abiertas para extender su funcionalidad sin necesidad de modificar su código existente, lo que promueve la extensibilidad del software sin alterar su comportamiento.
- **Principio de Sustitución de Liskov (LSP):** Las subclasses deben poder sustituir a sus clases base sin cambiar el comportamiento del programa, asegurando que las subclasses sean compatibles con las superclases.
- **Principio de Segregación de Interfaces (ISP):** Las interfaces deben ser pequeñas y específicas, de modo que las clases no se vean obligadas a implementar métodos que no utilizan, evitando la dependencia de funcionalidades innecesarias.
- **Principio de Inversión de Dependencia (DIP):** Las clases de alto nivel no deben depender de las clases de bajo nivel, sino de abstracciones, lo que promueve la flexibilidad al permitir cambiar implementaciones concretas sin afectar el código de alto nivel.

Conclusión: tenemos que hacer nuestro código, limpio, mantenible y escalable

Bloque 3 - Herramientas y dependencias

Git

Git es un sistema de control de versiones, lo que significa que es una herramienta que te permite llevar un registro de los cambios en tus archivos a lo largo del tiempo. Es especialmente útil cuando trabajas en proyectos de programación o cualquier tarea que requiera mantener un historial de ediciones.

Dejo por aquí una lista de reproducción donde se explica prácticamente todo lo que se puede hacer en GIT y cómo hacerlo:

<https://www.youtube.com/playlist?list=PLTd5ehlj0goMCnj6V5NdzSIHBgrIXckGU>

Es importante conocer que además de gestionar GIT por comandos, tenemos la posibilidad de hacerlo desde algún GUI, como por ejemplo:

GitHub Desktop, Fork, Sourcetree, SmartGit

Conventional commits

Los "Conventional Commits" (Commits Convencionales) son una convención específica para escribir mensajes de confirmación (commits) en Git. Esta convención se utiliza para estandarizar y estructurar los mensajes de confirmación de manera que sean más informativos y fáciles de seguir.

Esta práctica establece un formato específico para los mensajes de los commits, lo que proporciona varias ventajas significativas en el proceso de desarrollo colaborativo.

- **Semántica clara y consistente:** imponen un formato estructurado que sigue reglas semánticas específicas. Esto permite una comunicación más clara y consistente entre los miembros del equipo sobre los cambios realizados en el código.
- **Generación automática de versiones:** facilita la automatización del proceso de generación de versiones. Herramientas como Semantic Versioning (SemVer) pueden analizar los mensajes de los commits y determinar automáticamente la versión del software en función de los cambios realizados.

- **Historial de cambios más comprensible:** Los mensajes de los commits consistentes y semánticamente estructurados facilitan la revisión del historial de cambios. Esto resulta especialmente útil al buscar información específica sobre una característica, corrección de errores o mejora en el código.
- **Integración con herramientas y flujos de trabajo:** Muchas herramientas de desarrollo, sistemas de integración continua (CI), y servicios de gestión de versiones como GitHub y GitLab, están diseñadas para trabajar de manera más efectiva con mensajes de commit que siguen la convención de Conventional Commits.
- **Facilita la colaboración:** La adopción de una convención común en los mensajes de los commits hace que sea más fácil para los miembros del equipo entender y colaborar en el código de manera más efectiva. Esto es especialmente importante en equipos grandes y distribuidos.

Documentación oficial:

[Link pagina oficial Conventional Commits](#)

Los mensajes de confirmación convencionales siguen un formato predefinido que incluye tres partes clave:

Tipo (Type): El tipo describe la naturaleza del cambio realizado en el commit. Algunos ejemplos comunes de tipos incluyen "feat" (característica nueva), "fix" (arreglo de errores), "docs" (documentación), "chore" (tareas generales), entre otros.

Área (Scope) (Opcional): El ámbito opcional especifica la parte específica del proyecto afectada por el cambio. Esto puede ser un módulo, un archivo o una función, por ejemplo.

Mensaje (Message): El mensaje es una descripción concisa y clara del cambio que se realizó. Debe ser escrito en tiempo presente y en imperativo. Por ejemplo, "Agrega funcionalidad de inicio de sesión" o "Corrige error en la validación de correo electrónico".

Pero como en todo en Wipay, nos basamos en unos estándares y les añadimos valor propio. Por esto mismo, hemos decidido "personalizar" un poco el formato de Conventional Commits y agregarle algo de sabor para dejarlo a nuestro gusto y que herramientas de generación de changelog nos faciliten el trabajo.

Aunque los tipos siguen siendo los mismos (feat/chore/fix/styles...) simplemente hemos decidido cambiar el formato en el que añadimos el alcance que pasaría a estar detrás de los que siguen al tipo para asegurarnos de que todos seguimos el mismo criterio.

cambios

Ejemplos a la hora de hacer nuestros commits:

- Añadir una nueva característica (feat)
feat: [LOGIN]: Agregar autenticación de inicio de sesión
- Corrección de errores (fix)
fix: [VALIDATION]: Corregir validación de contraseñas en el formulario de registro
- Documentación (docs)
docs: [README]: Actualizar el archivo README con instrucciones de instalación
- Tareas generales (chore)
chore: [DEPLOY]: Preparar el proyecto para la implementación en producción
- Refactorización de código (refactor)
refactor: [API]: Reestructurar la capa de servicios para mejorar la eficiencia
- Añadir pruebas (test)
test: [LOGIN]: Agregar pruebas unitarias para el proceso de inicio de sesión
- Estilo y formato (style)
style: [CSS]: Alinear la sangría en los archivos CSS
- Eliminación de código (revert)
revert: [API]: Revertir el commit anterior que introdujo un error en la API
- Otro (misc)
misc: [LICENSE]: Actualizar la información de licencia

[Video de Carlos Azaustre que comenta esto mismo](#)

Actualmente la mayoría de repositorios en los que trabajamos están configurados con herramientas que se encargan de validar el mensaje del commit automáticamente, impidiendo al desarrollador introducir un commit que no cumpla con el estándar.

PONER EJEMPLO DE CÓMO GENERAR CHANGELOG

Postman

Postman es una herramienta esencial para trabajar con APIs y servicios web, ya sea para pruebas manuales, automatización de pruebas, colaboración en equipos de desarrollo o

monitorización de APIs. Es ampliamente utilizado en la comunidad de desarrollo de software y puede ayudarte a simplificar y mejorar tus procesos de desarrollo y prueba relacionados con APIs.

[Descárgalo aquí](#)

**En el Bloque 5 explicaremos como usar esta herramienta con peticiones HTTP*

Maven

Maven es una herramienta de construcción y gestión de proyectos que utiliza un archivo de configuración llamado "POM" (Project Object Model) para definir cómo se compila, prueba y empaqueta un proyecto. Puedes pensar en él como un sistema de automatización de tareas para proyectos Java.

Para qué se usa Maven:

- **Gestión de dependencias:** Maven facilita la gestión de las bibliotecas y dependencias de un proyecto. Puedes especificar las dependencias en el archivo POM y Maven se encargará de descargarlas automáticamente desde repositorios remotos.
- **Compilación y construcción:** Maven automatiza el proceso de compilación de tu proyecto. Ejecuta un simple comando Maven compilar tu código fuente y generará los artefactos deseados (como JAR o WAR).
- **Ejecución de pruebas:** Maven proporciona comandos para ejecutar pruebas unitarias y generar informes de cobertura de código.
- **Empaquetado de artefactos:** Puedes usar Maven para empaquetar tu proyecto en un formato específico, como un archivo JAR (Java Archive) o un archivo WAR (Web Application Archive), listo para desplegar en un servidor de aplicaciones.
- **Ciclo de vida del proyecto:** Maven define un ciclo de vida común para proyectos Java que incluye fases como "clean" (limpieza), "compile" (compilación), "test" (pruebas) y "package" (empaquetado). Puedes ejecutar estas fases según tus necesidades.

Ejemplo de uso

Supongamos que tienes un proyecto Java simple con las siguientes características:

- Nombre del proyecto: "MiProyecto"
- Dependencia: Necesitas la biblioteca Apache Commons Lang en tu proyecto.
- El archivo POM de Maven para este proyecto se vería así:

Java

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.ejemplo</groupId>
  <artifactId>MiProyecto</artifactId>
  <version>1.0.0</version>

  <dependencies>
    <dependency>
      <groupId>org.apache.commons</groupId>
      <artifactId>commons-lang3</artifactId>
      <version>3.12.0</version>
    </dependency>
  </dependencies>
</project>
```

Con este archivo POM, Maven descargará automáticamente la biblioteca Apache Commons Lang y te permitirá compilar, ejecutar pruebas y empaquetar tu proyecto Java con facilidad.

IA

Hoy en día, todos sabemos la importancia y relevancia de la Inteligencia Artificial, por lo tanto, evitarla es un claro error ya que nos proporciona una asistencia personalizada que no se ha visto nunca antes.

Es importante tener en cuenta que mientras las herramientas de IA como Chat GPT ofrecen muchas ventajas, también **es esencial que los estudiantes desarrollen habilidades críticas de resolución de problemas y programación de manera independiente**. Estas herramientas deben **complementar, no reemplazar**, la enseñanza y la tutoría humana en un entorno educativo efectivo.

Bloque 4 - Librerías

Lombok

Lombok es una librería de Java que se utiliza comúnmente en proyectos de Spring Boot para reducir la verbosidad del código fuente al eliminar la necesidad de escribir código repetitivo para la generación de getters, setters, constructores, y otros métodos comunes en las clases Java, sin Lombok, una clase sería algo como esto:

Java

```
public class Persona {

    //Atributos
    private String nombre;
    private List<Propiedad> propiedades;

    //Constructor
    public Persona(String nombre, List<Propiedad> propiedades) {
        this.nombre = nombre;
        this.propiedades = propiedades;
    }

    //Getters y Setters
    public String getNombre() {
        return nombre;
    }

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

    public List<Propiedad> getPropiedades() {
        return propiedades;
    }

    public void setPropiedades(List<Propiedad> propiedades) {
        this.propiedades = propiedades;
    }

    //To String
    @Override
    public String toString() {
        return "Persona{" +
            "nombre='" + nombre + '\'' +
            ", propiedades=" + propiedades +
            '}';
    }
}
```

Con Lombok sería esto:

Java

```
@AllArgsConstructor
@Getters
@Setters
@ToString
@NoArgsConstructor
public class Persona {
    private String nombre;
    private List<Propiedad> propiedades;
}
```

Ambos códigos hacen lo mismo, sin embargo, es obvio que nuestras clases se reducen en muchas líneas de código, lo que conlleva mucha limpieza y rapidez.

A continuación te listo las anotaciones principales que se usan en Lombok:

- **@Data**: Esta anotación agrega automáticamente los métodos equals, hashCode, toString, getters y setters para todos los campos de una clase.
- **@Getter y @Setter**: Estas anotaciones generan automáticamente los métodos getters y setters para los campos de una clase, respectivamente.
- **@NoArgsConstructor, @RequiredArgsConstructor, y @AllArgsConstructor**: Estas anotaciones generan constructores sin argumentos, constructores con argumentos requeridos y constructores con todos los argumentos para una clase, respectivamente.
- **@Builder**: Esta anotación genera un patrón de diseño de constructor fluente que facilita la creación de instancias de objetos con una sintaxis más clara.

Builder es interesante tenerlo en cuenta a la hora de instanciar una clase:

Java

//Clase

```
@Getter
@Setter
@AllArgsConstructor
@Builder
@ToString
@Entity
public class Persona {
    private String id;
    private String nombre;
    private String dni;
}
```

```
private List<Propiedad> propiedades;
}

public class PersonaModel {
    private String nombre;
    private String dni;
    private List<Propiedad> propiedades;
}

//Así se instanciaría:
Persona propietario = Persona.builder()
    .nombre("David")
    .propiedades(new ArrayList<>())
    .build();
```

- **@ToString**: Genera automáticamente el método toString para la clase.

En nuestro pom se vería reflejado como una dependencia más:

```
Java
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
</dependency>
```

MapStruct

MapStruct es una librería que se utiliza en proyectos de Spring Boot (y en otros proyectos Java) para simplificar el mapeo de objetos entre diferentes clases o estructuras de datos, como DTOs (Data Transfer Objects) y entidades. Esta librería genera automáticamente código de mapeo eficiente en tiempo de compilación, lo que mejora el rendimiento y la seguridad de tu aplicación.

Definición de mapper: primero, debes definir una interfaz llamada "Mapper" que describe cómo se debe mapear un objeto de una clase origen a una clase destino. Esta interfaz utiliza anotaciones de MapStruct para indicar cómo realizar el mapeo.

Generación de código: luego, MapStruct genera automáticamente la implementación de esa interfaz durante el proceso de compilación. El código generado se encarga de realizar el mapeo de manera eficiente, lo que evita la necesidad de escribir código de mapeo manualmente.

Uso en la aplicación: puedes utilizar el Mapper generado en tu aplicación Spring Boot para mapear objetos de origen a objetos de destino de manera sencilla. Esto es particularmente útil cuando trabajas con entidades JPA y DTOs, ya que simplifica la conversión entre ellos.

Algunas de las ventajas de MapStruct son la generación de código eficiente, la prevención de errores de mapeo en tiempo de compilación, y la reducción de la cantidad de código repetitivo que debes escribir para mapear objetos. Esto hace que tu código sea más limpio y fácil de mantener.

```
Java
<dependency>
  <groupId>org.mapstruct</groupId>
  <artifactId>mapstruct</artifactId>
  <version>1.5.3.Final</version>
</dependency>
```

**Este punto se verá mejor en el siguiente bloque aplicado a la práctica*

Bloque 5 - SpringBoot

SpringBoot en DynamoDB

Inyección de dependencias

Bloque 6 - SpringWeb básico

RestController

RequestHeader

RequestParam

PathVariable

ResponseEntity

Manejo de excepciones

Feign

Postman

Postman es una plataforma diseñada para desarrolladores que permite crear, probar y colaborar en APIs de manera sencilla. Con ella, puedes enviar peticiones HTTP, ver las respuestas, y realizar pruebas de tus APIs sin necesidad de escribir código.

Entre sus principales ventajas están:

Facilidad de uso: Interfaz gráfica para probar APIs rápidamente.

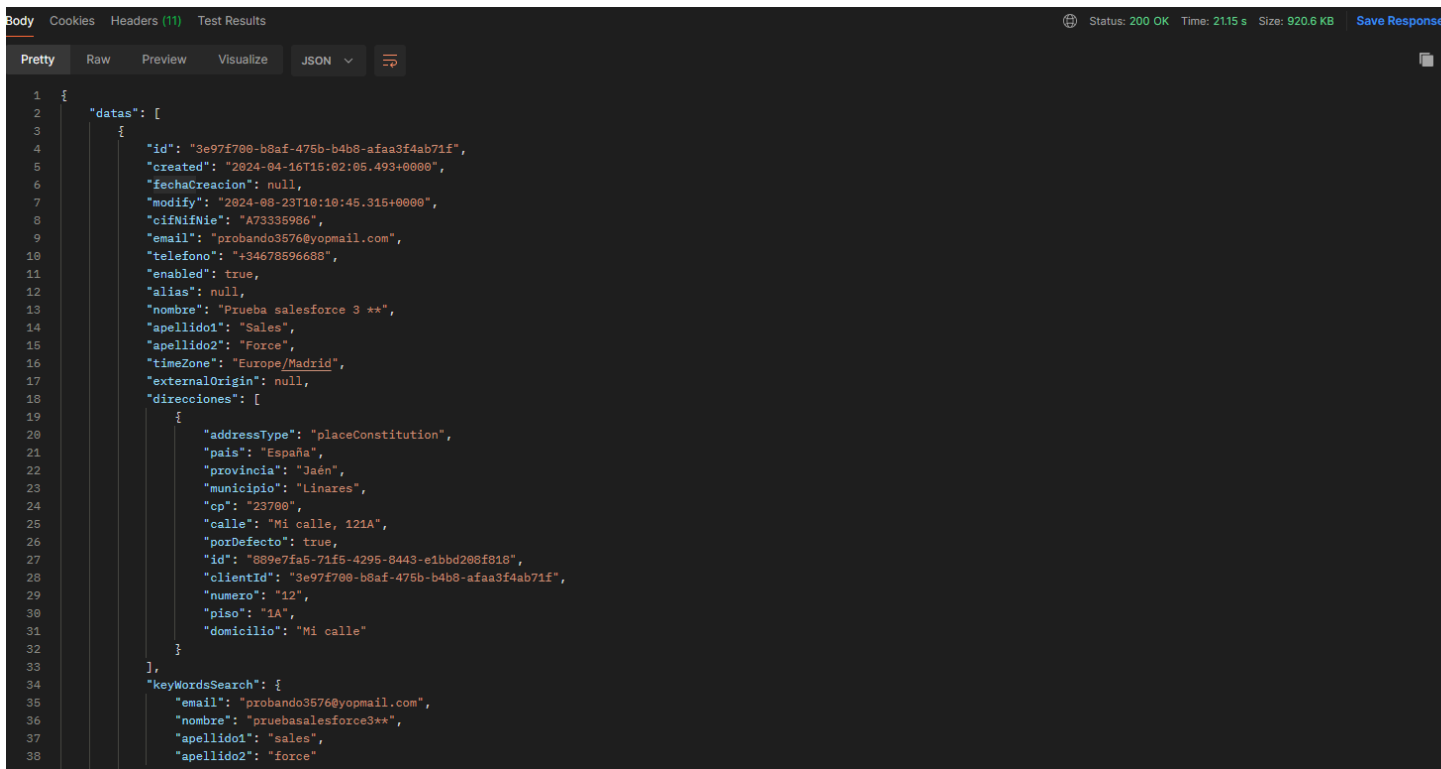
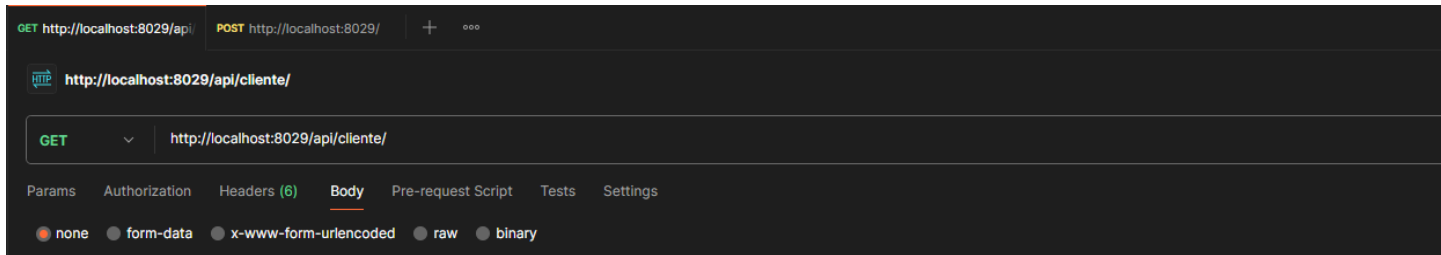
Automatización: Tests automáticos y soporte para integraciones de CI/CD.

Colaboración: Ideal para equipos que trabajan juntos en APIs.

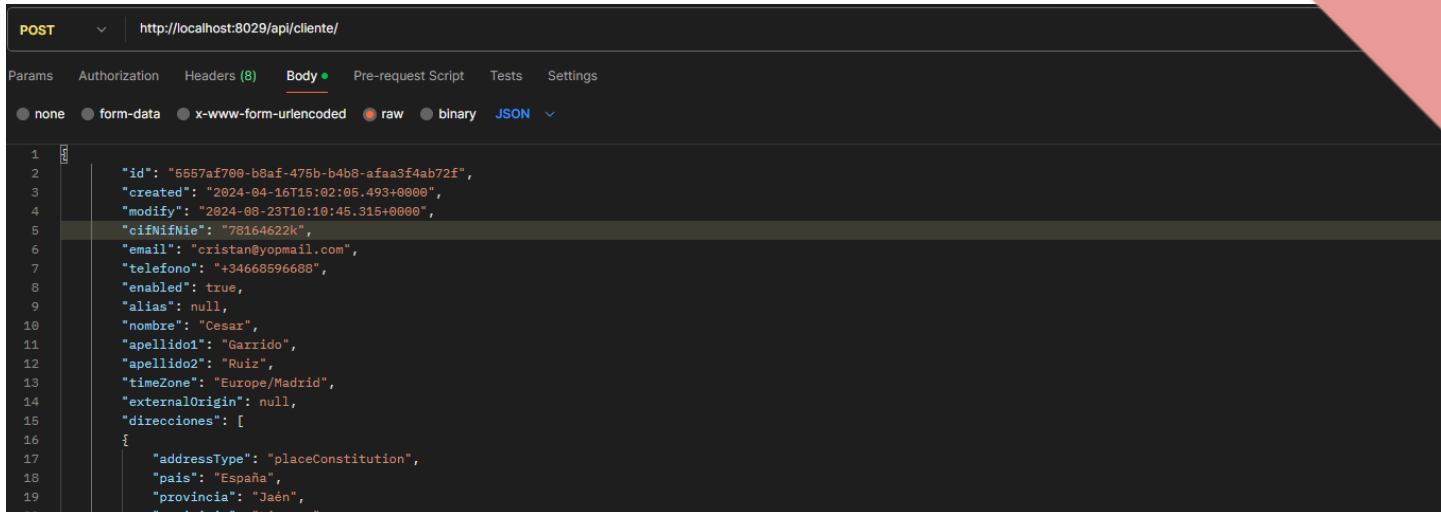
Es una herramienta clave para gestionar todo el ciclo de vida de una API.

EJEMPLOS:

- Petición GET: Hacemos una petición GET a <http://localhost:8029/api/cliente/> y nos devuelve la lista de todos los usuarios



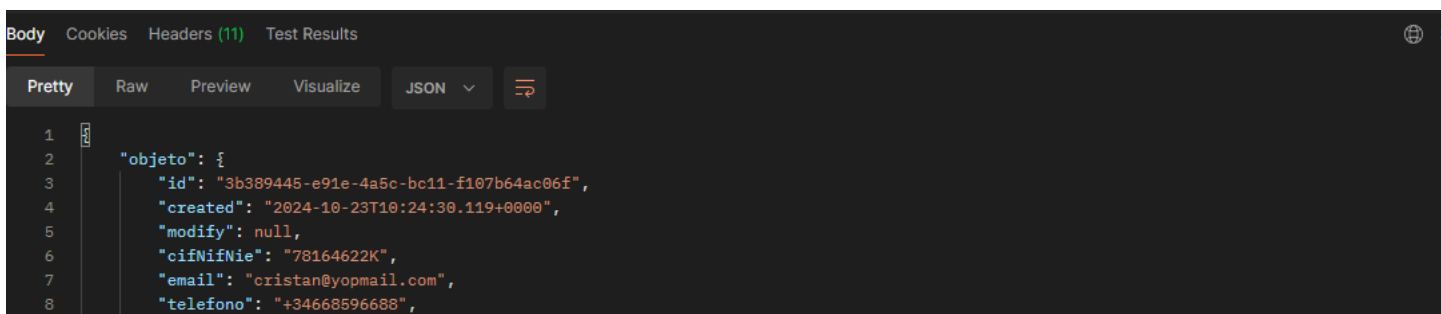
- Petición post: Hacemos una petición POST a <http://localhost:8029/api/cliente/> y nos devuelve la estado 201 que significa que el usuario fue creado con éxito



```
POST http://localhost:8029/api/cliente/

{
  "id": "5557af700-b8af-475b-b4b8-afaa3f4ab72f",
  "created": "2024-04-16T15:02:05.493+0000",
  "modify": "2024-08-23T10:10:45.315+0000",
  "cifNifNie": "78164622K",
  "email": "cristan@yopmail.com",
  "telefono": "+34668596688",
  "enabled": true,
  "alias": null,
  "nombre": "Cesar",
  "apellido1": "Garrido",
  "apellido2": "Ruiz",
  "timeZone": "Europe/Madrid",
  "externalOrigin": null,
  "direcciones": [
    {
      "addressType": "placeConstitution",
      "pais": "España",
      "provincia": "Jaén",
      "municipio": "Jaén"
    }
  ]
}
```

En el método post tendremos que poner un body para poder mandarle la petición post con lo que queramos “postear”



```
Body Cookies Headers (11) Test Results

Pretty Raw Preview Visualize JSON

{
  "objeto": {
    "id": "3b389445-e91e-4a5c-bc11-f107b64ac06f",
    "created": "2024-10-23T10:24:30.119+0000",
    "modify": null,
    "cifNifNie": "78164622K",
    "email": "cristan@yopmail.com",
    "telefono": "+34668596688",
  }
}
```

Swagger

Swagger es comparado con postman pero sus enfoques son diferentes, este esta mas centrado en la documentación y diseño de APIs con el estándar OpenAPI. En lo que postman destaca es la versatilidad para pruebas , mientras que Swagger destaca en la definición precisa y visualización de las APIs

Entre sus principales ventajas están:

Documentación automática: Swagger genera documentación en tiempo real basada en los esquemas de las APIs, permitiendo a los usuarios entender y probar los endpoints directamente desde un navegador.

Diseño de APIs: Con Swagger, puedes definir claramente los contratos de tu API desde el principio, usando el formato Open API, lo que asegura que tanto desarrolladores como clientes trabajen bajo las mismas especificaciones.

Interfaz interactiva: Swagger UI permite a los usuarios interactuar con la API directamente desde la documentación, probando cada endpoint sin necesidad de otra herramienta externa.

Bloque 7 - SpringData básico

Validación con @Valid

Códigos de respuesta

Arquitectura por capas

Anotaciones

[Ejemplo de caso real](#)

Bloque 8 - Testing

JUnit

Mockito

SonarQube

Bloque 9 - Docker

Docker Desktop

Docker Hub

Trabajando con contenedores

Trabajando con imágenes

Docker Compose

Docker Cheat Sheet

Bloque 10 - Spring Security

Bloque 11 - Mensajería aplicada a AWS

SQS

SNS

Bloque 12 - Spring Cloud

Balanceador de carga

Api Gateway

Eureka

Config server

Bloque 13 - Spring Batch

Bloque 14 - Microservicios con Spring cloud

<https://www.youtube.com/watch?v=ksmE3KoX9U&list=PL145AyWAbMDhwUbBL74s1D2ZV9EqBaQ1t>

<https://docs.localstack.cloud/user-guide/aws/sns/>

Bloque 15 - Proyecto final