

Autor:  
Ana Isabel Vegas

ibertech

**end2end dedication**

CURSO

# Microservicios con Java y Spring Boot

f X in

[www.ibertech.es](http://www.ibertech.es)



**end2end dedication**

**Tema 1** Introducción a Microservicios con Java y Spring Boot

**Tema 2** Creación de APIs RESTful con Spring Boot

**Tema 3** Comunicación entre Microservicios con Spring Cloud

**Tema 4** Persistencia y Gestión del Estado en Microservicios Java

**Tema 5** Seguridad y Autenticación en Microservicios con Spring Security

**Tema 6** Despliegue de Microservicios con Docker y Kubernetes

**Tema 7** Monitorización y Logging en Microservicios Java



ibertech

end2end dedication

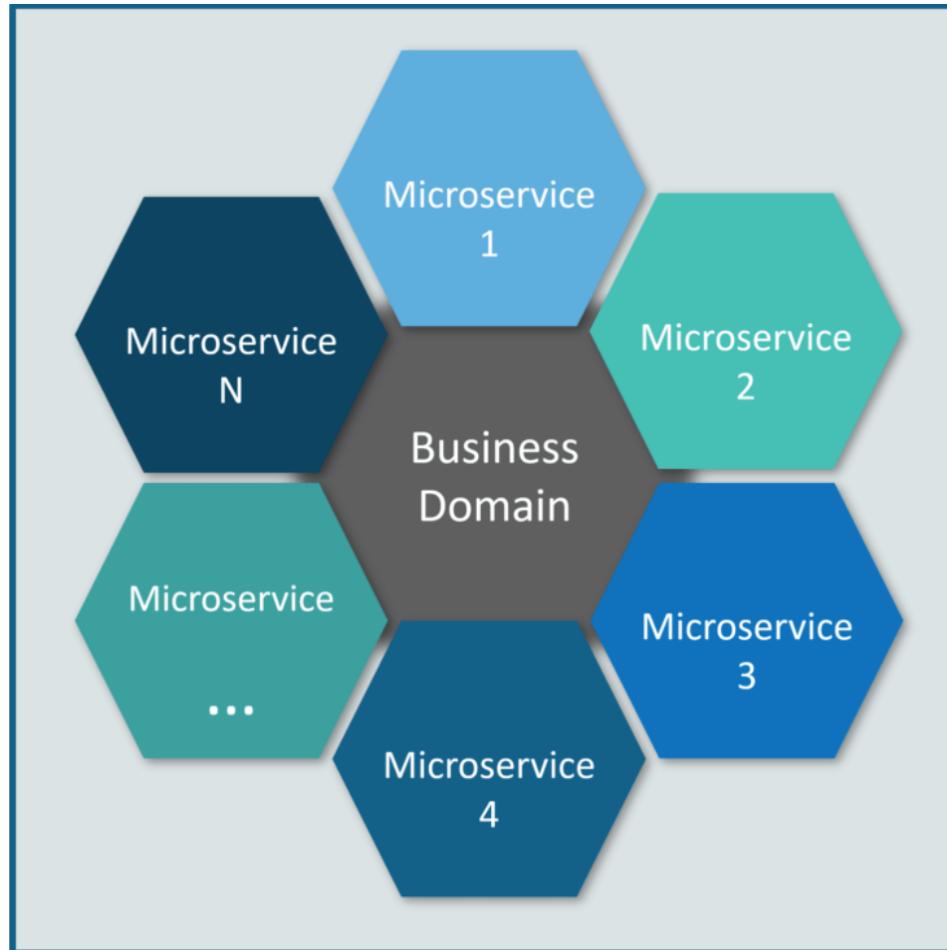
# Tema 1

Introducción a Microservicios con Java  
y Spring Boot

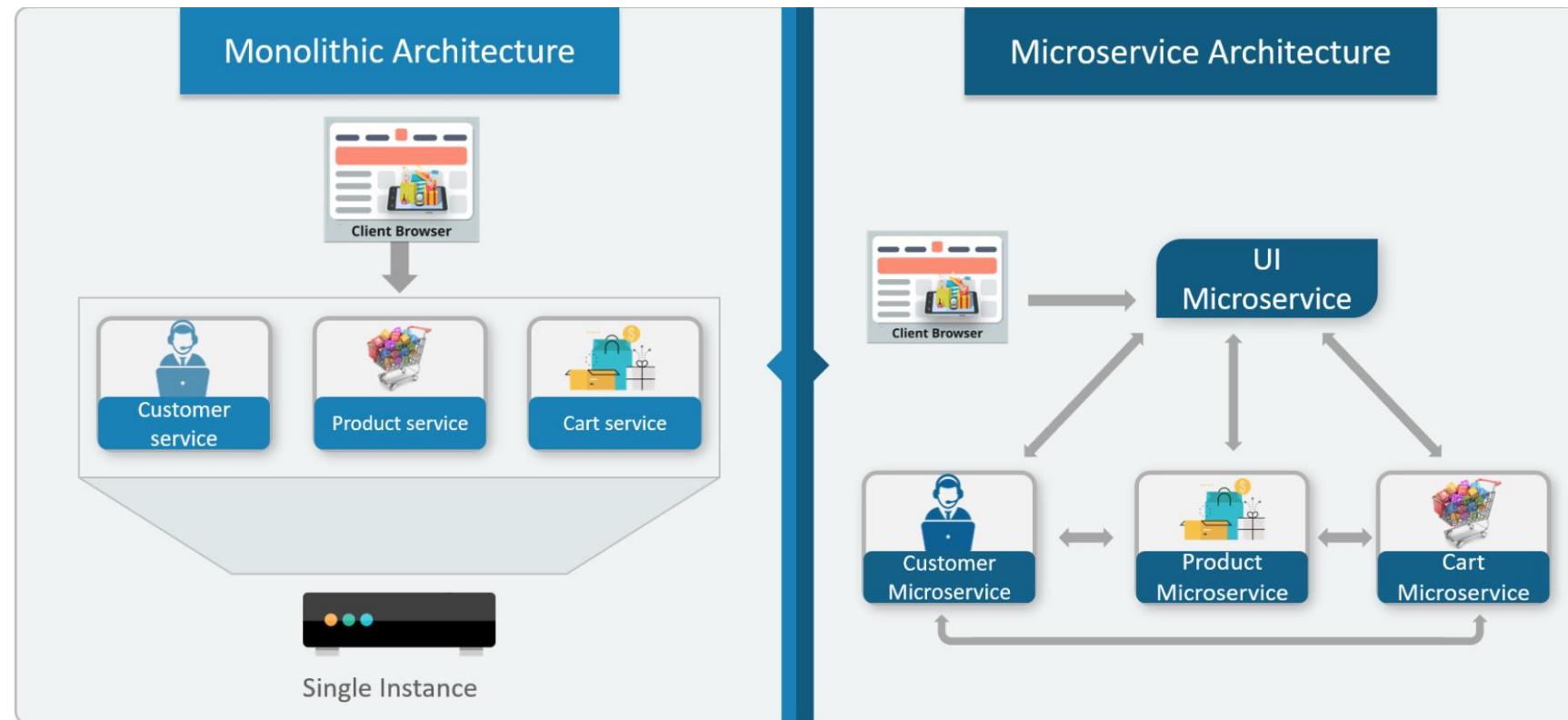
## Qué es un microservicio?

- Según [Martin Fowler](#) y [James Lewis](#) explican en su artículo [Microservices](#), los **microservicios** se definen como un estilo arquitectural, es decir, una forma de desarrollar una aplicación, basada en un conjunto de pequeños servicios, cada uno de ellos ejecutándose de forma autónoma y comunicándose entre si mediante mecanismos livianos, generalmente a través de peticiones **REST** sobre HTTP por medio de sus **APIs**.

## Qué es un microservicio?



## Arquitectura monolítica vs Arquitectura microservicios



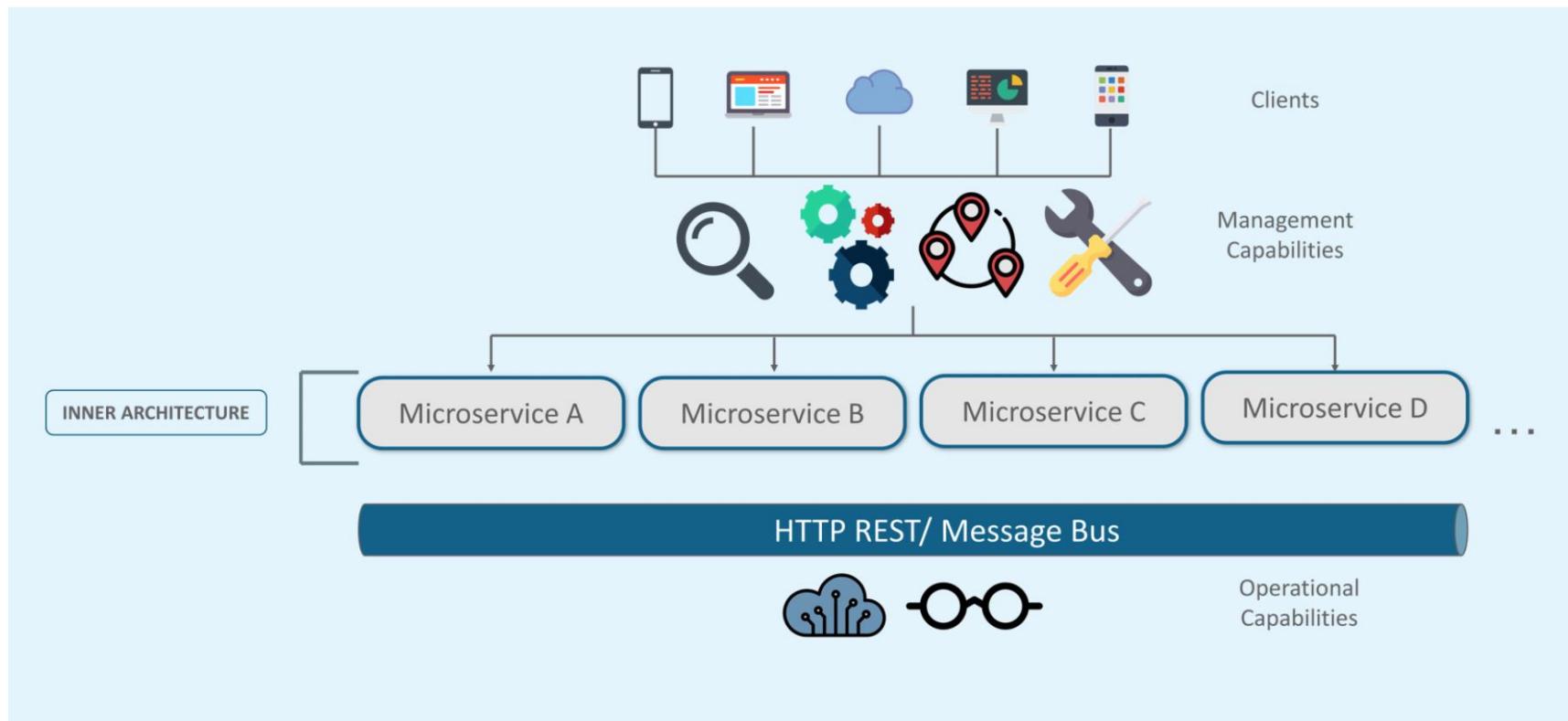
## Tendencia en el desarrollo

- La tendencia es que las aplicaciones sean diseñadas con un ***enfoque orientado a microservicios***, construyendo múltiples servicios que colaboran entre si, en lugar del ***enfoque monolítico***, donde se construye y despliega una única aplicación que contenga todas las funcionalidades.

## Características de los Microservicios

- Pueden ser auto-contenidos, de tal forma que incluyen todo lo necesario para prestar su servicio
- Servicios pequeños, lo que facilita el mantenimiento. Ej: Personas, Productos, Posición Global, etc
- Principio de responsabilidad única: cada microservicio hará una única cosa, pero la hará bien
- Políglotas: una arquitectura basada en microservicios facilita la integración entre diferentes tecnologías (lenguajes de programación, BBDD...etc)
- Despliegues unitarios: los microservicios pueden ser desplegados por separado, lo que garantiza que cada despliegue de un microservicio no implica un despliegue de toda la plataforma. Tienen la posibilidad de incorporar un servidor web embebido como Tomcat o Jetty
- Escalado eficiente: una arquitectura basada en microservicios permite un escalado elástico horizontal, pudiendo crear tantas instancias de un microservicio como sea necesario.

## Arquitectura microservicios



## Arquitectura microservicios

- Diferentes clientes de diferentes dispositivos intentan usar diferentes servicios como búsqueda, creación, configuración y otras capacidades de administración
- Todos los servicios se separan según sus dominios y funcionalidades y se asignan a microservicios individuales.
- Estos microservicios tienen su propio balanceador de carga y entorno de ejecución para ejecutar sus funcionalidades y al mismo tiempo captura datos en sus propias bases de datos.
- Todos los microservicios se comunican entre sí a través de un servidor sin estado que es REST o Message Bus.
- Los microservicios conocen su ruta de comunicación con la ayuda de Service Discovery y realizan capacidades operativas tales como automatización, monitoreo
- Luego, todas las funcionalidades realizadas por los microservicios se comunican a los clientes a través de la puerta de enlace API.
- Todos los puntos internos están conectados desde la puerta de enlace API. Por lo tanto, cualquiera que se conecte a la puerta de enlace API se conecta automáticamente al sistema completo

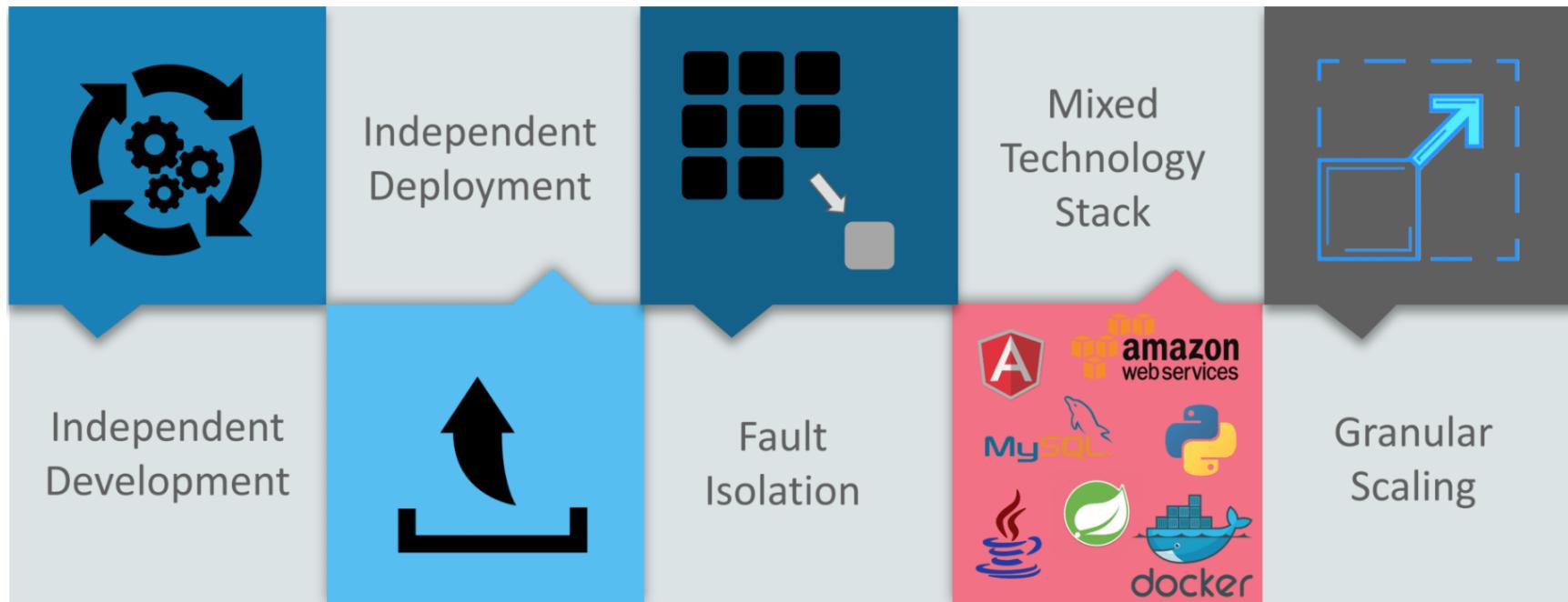
## Características de los microservicios



## Características de los microservicios

- Desacoplamiento: los servicios dentro de un sistema se desacoplan en gran medida. Por lo tanto, la aplicación en su conjunto se puede construir, modificar y escalar fácilmente.
- Componentes: los microservicios se tratan como componentes independientes que se pueden reemplazar y actualizar fácilmente.
- Capacidades empresariales: los microservicios son muy simples y se centran en una sola capacidad
- Autonomía: los desarrolladores y los equipos pueden trabajar de forma independiente, lo que aumenta la velocidad.
- Entrega continua: permite lanzamientos frecuentes de software, a través de la automatización sistemática de la creación, prueba y aprobación del software.
- Responsabilidad: Los microservicios no se centran en aplicaciones como proyectos. En cambio, tratan las aplicaciones como productos de los que son responsables.
- Gobernanza descentralizada: el enfoque está en usar la herramienta adecuada para el trabajo correcto. Eso significa que no hay un patrón estandarizado o ningún patrón tecnológico. Los desarrolladores tienen la libertad de elegir las mejores herramientas útiles para resolver sus problemas
- Agilidad - Los microservicios apoyan el desarrollo ágil. Cualquier nueva característica puede ser desarrollada rápidamente y descartada nuevamente

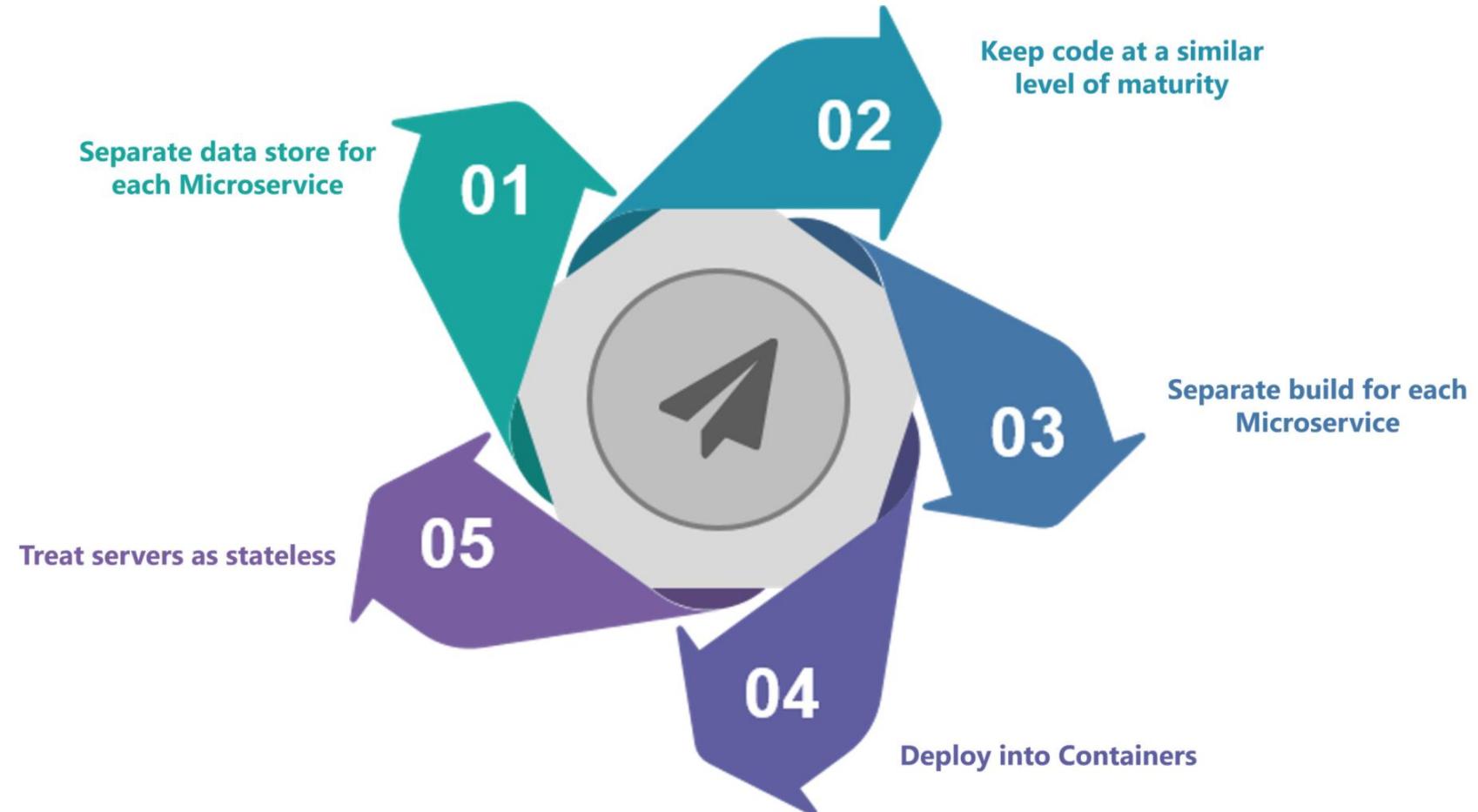
## Ventajas de los microservicios



## Ventajas de los microservicios

- Desarrollo independiente: todos los microservicios se pueden desarrollar fácilmente según su funcionalidad individual
- Implementación independiente: en función de sus servicios, se pueden implementar individualmente en cualquier aplicación
- Aislamiento de fallos: incluso si un servicio de la aplicación no funciona, el sistema continúa funcionando
- Pila de tecnología mixta: se pueden utilizar diferentes lenguajes y tecnologías para crear diferentes servicios de la misma aplicación
- Escalado granular: los componentes individuales pueden escalarse según la necesidad, no es necesario escalar todos los componentes

## Buenas prácticas de diseño



Quien los utiliza?



NETFLIX

GILT



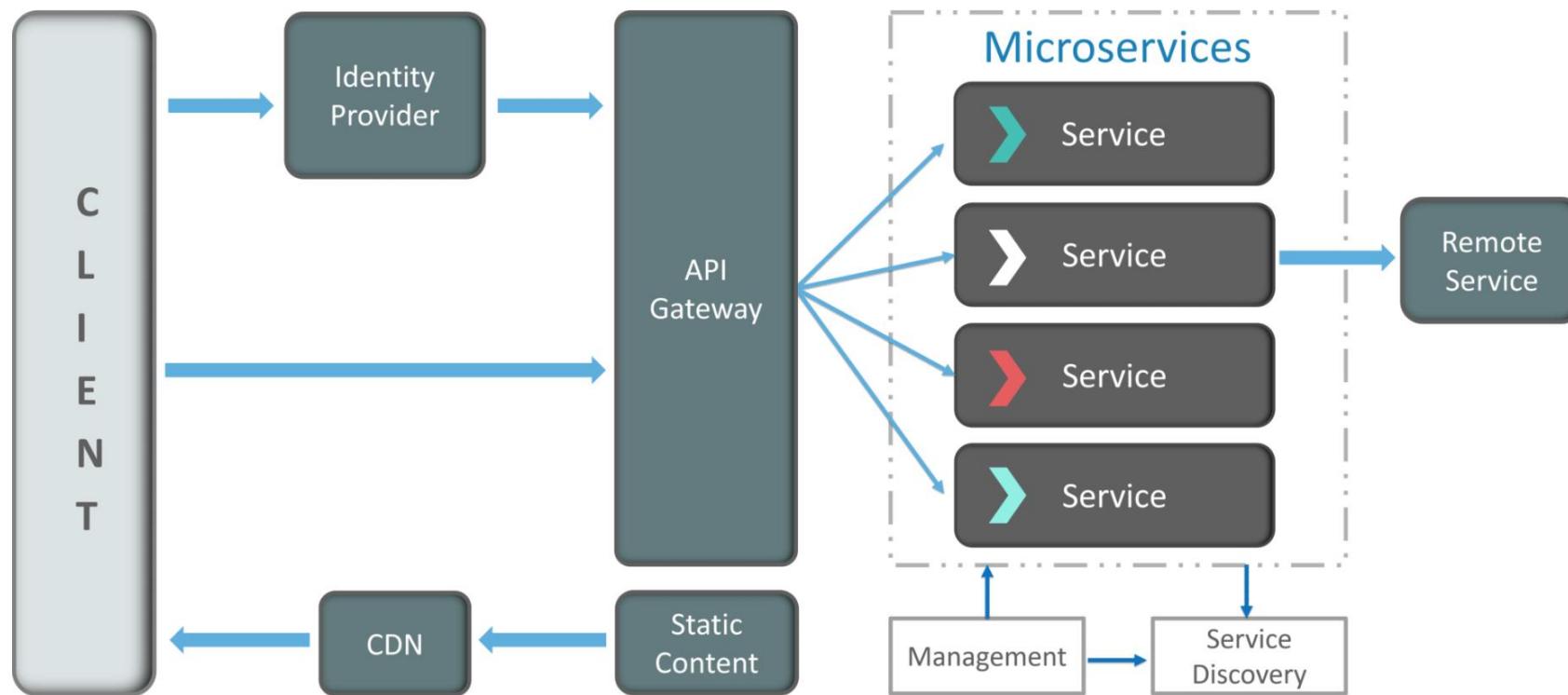
eBay



NORDSTROM

the guardian

## Componentes de la arquitectura



## Que es Spring Boot?

- Spring Boot es una parte de Spring que nos permite crear diferentes tipos de aplicaciones de una manera rápida y sencilla.
- Sus características principales son que provee out-of-the-box una serie de elementos que nos permiten desarrollar diferentes tipos de aplicaciones de forma casi inmediata. Algunas de estas características son:
  - Servidores de aplicaciones embebidos (Tomcat, Jetty, Undertow)
  - POMs con dependencias y plug-ins para Maven
  - Uso extensivo de anotaciones que realizan funciones de configuración

## Configuración del pom

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.4.RELEASE</version>
</parent>

<properties>
    <java.version>1.8</java.version>
</properties>

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
        <groupId>ch.qos.logback</groupId>
        <artifactId>logback-classic</artifactId>
        <version>1.1.11</version>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-logging</artifactId>
        <version>2.1.4.RELEASE</version>
    </dependency>
</dependencies>
```

## Principales anotaciones

- La etiqueta `@Configuration`, indica que la clase en la que se encuentra contiene la configuración principal del proyecto.
- La anotación `@EnableAutoConfiguration` indica que se aplicará la configuración automática del starter que hemos utilizado. Solo debe añadirse en un sitio, y es muy frecuente situarla en la clase main.
- En tercer lugar, la etiqueta `@ComponentScan`, ayuda a localizar elementos etiquetados con otras anotaciones cuando sean necesarios.
- Para no llenar nuestra clase de anotaciones, podemos sustituir las etiquetas `@Configuration`, `@EnableAutoConfiguration` y `@ComponentScan` por `@SpringBootApplication`

## Clase principal

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class HelloWorldApplication {

    public static void main(String[] args) {
        SpringApplication.run(HelloWorldApplication.class, args);
    }
}
```



ibertech

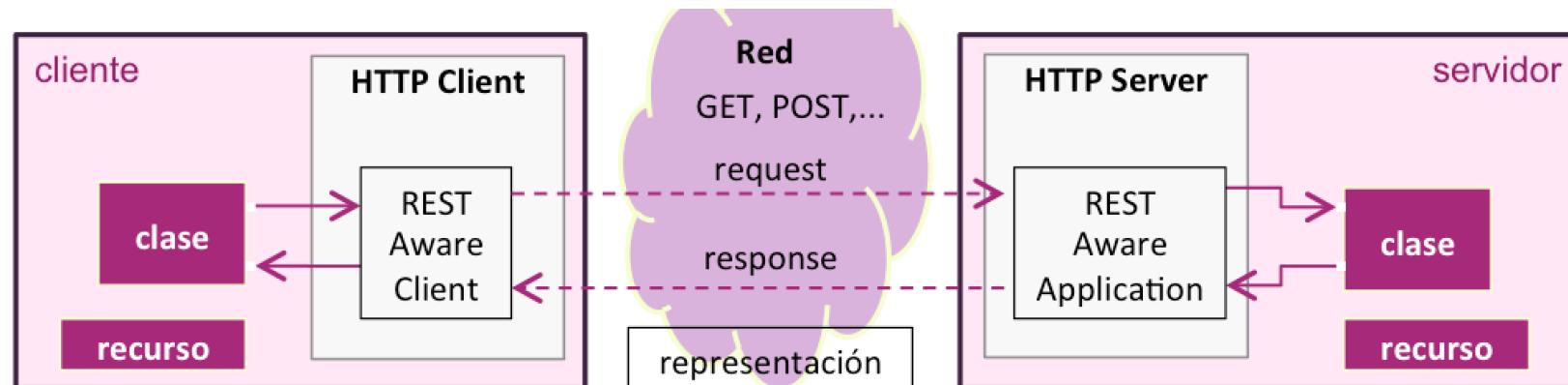
end2end dedication

# Tema 2

Creación de APIs RESTful con  
Spring Boot

## Creando RESTfull Web Service

- REST (Representational State Transfer) es un estilo de arquitectura para sistemas distribuidos, desarrollada por la W3C, junto con el protocolo HTTP.
- Las arquitecturas REST tienen clientes y servidores.
- El cliente realiza un envío (request) al servidor, el cual lo procesa y retorna una respuesta al cliente.
- Las peticiones y respuestas son construidas alrededor de representaciones de recursos. Recurso es una entidad, y representación es cómo se formatea.



## Creando RESTfull Web Service

- Una API del tipo RESTful, o RESTful Web Service, es una API web implementada con HTTP y los principios REST, con los siguientes aspectos:
  - Una URI base del servicio.
  - Un formato de mensajes, por ejemplo JSON o XML.
  - Un conjunto de operaciones, que utilizan los métodos HTTP (GET, PUT, POST o DELETE).
- La API debe manejar hipertextos.
- A diferencia de los Web Services basados en SOAP, no hay un estándar comúnmente aceptado para los RESTful. Esto es porque REST es una arquitectura, mientras que SOAP es un protocolo.
- Esta desventaja se compensa con la simplicidad de su utilización y el bajo consumo de recursos durante el binding. Esto es especialmente útil en aplicaciones para dispositivos móviles

## Creando RESTfull Web Service

- Con REST, los métodos HTTP se asocian a tipos de operaciones sobre recursos. El uso comúnmente aceptado es el siguiente:
  - GET: Para recuperar la representación de un recurso. Es idempotente, es decir, si se invoca múltiples veces, retorna el mismo resultado.
  - POST: Para crear un recurso, o para actualizarlo. También, por las características del método, se utiliza para envíos grandes, o para evitar limitaciones de los otros métodos.
  - PUT: Para actualizar un recurso, ya que POST no es idempotente.
  - DELETE: Para eliminar un recurso.
  - OPTIONS: Se puede utilizar para hacer un "ping" del servicio, es decir, verificar su disponibilidad.
  - HEAD: Para buscar un recurso o consultar estado. Similar a GET

## Creando RESTfull Web Service

```
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class SaludoRest {

    // http://localhost:8080/hola
    @RequestMapping("/hola")
    public String hola() {
        return "Bienvenidos al curso";
    }

    // http://localhost:8080/adios?usuario="Anabel"
    @RequestMapping("/adios")
    public String adios(@RequestParam(value="usuario", defaultValue="Admin") String user) {
        return "Nos vamos a desayunar " + user;
    }
}
```

## Creando RESTfull Web Service

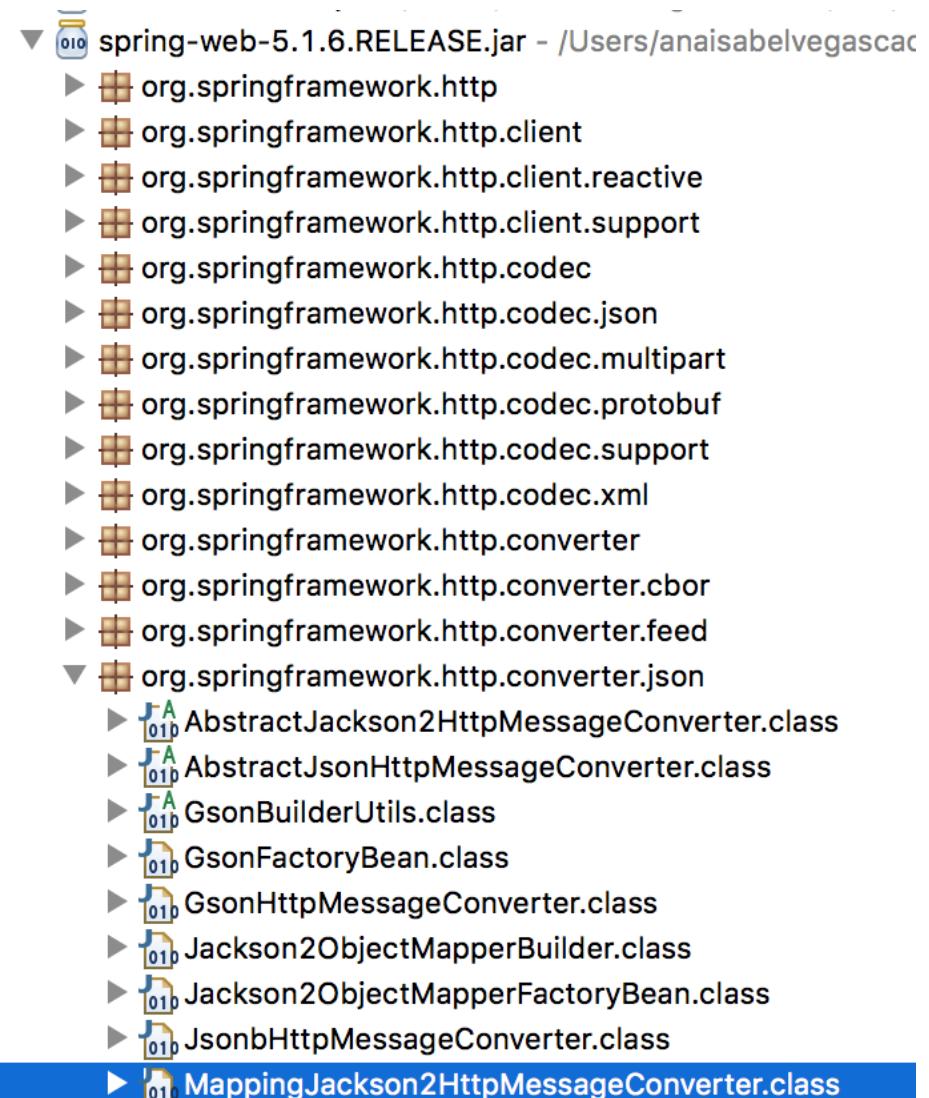
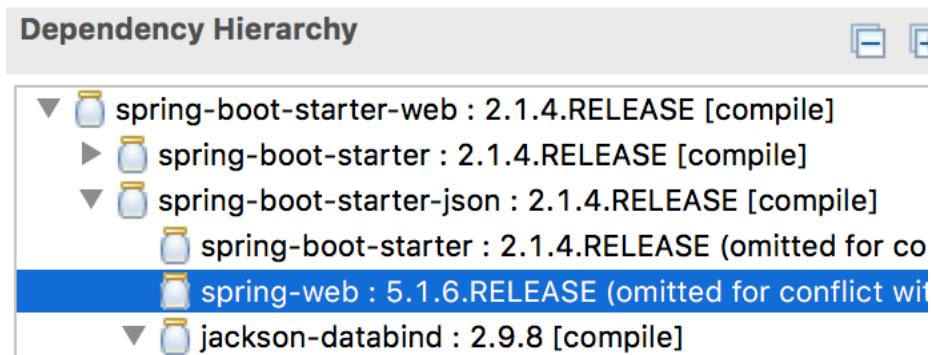
- Al agregar esta dependencia al pom.xml:

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

- La clase MappingJackson2HttpMessageConverter se encarga de convertir automáticamente la instancia a devolver en un formato JSON.

## Creando RESTfull Web Service

- Formateando la respuesta a formato JSON:



## Consumiendo RESTfull Web Service

- Para poder consumir un servicio Rest la dependencia Spring –Web nos proporciona un objeto que nos facilitara mucho la conectividad con el servicio. RestTemplate.

```
@Bean  
public RestTemplate restTemplate(RestTemplateBuilder builder) {  
    return builder.build();  
}
```

## Consumiendo RESTfull Web Service

- Una vez obtenido el objeto RestTemplate podemos lanzar la petición al servicio:

```
Producto producto = restTemplate.getForObject(  
    "http://localhost:8080/productos?codigo=2", Producto.class);
```

- Para mostrarlo en formato json debemos agregar la siguiente dependencia al pom.xml:

```
<dependency>  
    <groupId>com.fasterxml.jackson.core</groupId>  
    <artifactId>jackson-databind</artifactId>  
</dependency>
```

## Spring Validator

- El manejo de **parámetros** y la **validación de datos** en Spring se realiza principalmente mediante el uso de anotaciones y la integración con el estándar Bean Validation (JSR-303/JSR-380), lo que permite validar datos de forma declarativa y centralizada.
- **@Validated** en la clase es imprescindible para que las validaciones funcionen.
- Si la validación falla, Spring devuelve automáticamente un error HTTP 400 con un mensaje descriptivo

```
@RestController
@Validated // Necesario para activar la validación de parámetros en métodos GET
public class DemoController {

    // Validación de @RequestParam: debe ser entre 1 y 7
    @GetMapping("/name-for-day")
    public String getNameOfDayByNumber(
        @RequestParam @Min(1) @Max(7) Integer dayOfWeek) {
        // Lógica de negocio...
        return "Día número: " + dayOfWeek;
    }
}
```

## Reglas de validación

- @AssertFalse: El campo booleano tiene que ser false.
- @AssertTrue: El campo booleano tiene que ser true.
- @DecimalMax: El campo tiene que ser un numero cuyo valor sea menor o igual a un máximo especificado.
- @DecimalMin: El campo tiene que ser un numero cuyo valor sea mayor o igual a un mínimo especificado.
- @Digits: El campo tiene que ser un número cuyo valor se encuentre en un rango definido.
- @Future: El campo tiene que ser una fecha futura.
- @Max: El campo tiene que ser un numero cuyo valor sea menor o igual a un máximo especificado.
- @Min: El campo tiene que ser un numero cuyo valor sea mayor o igual a un mínimo especificado.
- @NotNull: El campo no puede tener valor null.
- @Null: El campo debe tener valor null.
- @Past: El campo debe tener una fecha ya pasada.
- @Pattern: La cadena tiene que coincidir con el patrón de expresión regular especificado.
- @Size: El tamaño del campo tiene que estar dentro de un límite especificado inclusive.

## Excepciones globales con `@ControllerAdvice`

- La implementación de **manejadores de excepciones globales** en Spring se realiza utilizando la anotación `@ControllerAdvice`, que permite capturar y procesar excepciones lanzadas desde cualquier controlador de la aplicación, centralizando así la gestión de errores y evitando duplicar código en cada controlador
- `@ControllerAdvice` indica que la clase manejará errores de forma global.
- `@ExceptionHandler` define métodos para gestionar tipos concretos de excepciones.

## Excepciones globales con @ControllerAdvice

```
// Manejador global de excepciones
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;

@ControllerAdvice
public class ManejadorGlobalExcepciones {

    @ExceptionHandler(RecursoNoEncontradoException.class)
    public ResponseEntity<String>
manejarRecursoNoEncontrado(RecursoNoEncontradoException ex) {
        return new ResponseEntity<>(ex.getMessage(), HttpStatus.NOT_FOUND);
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<String> manejarExcepcionGeneral(Exception ex) {
        return new ResponseEntity<>("Ocurrió un error inesperado",
HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

## Respuestas personalizadas con códigos de estado HTTP

- En **Spring Boot**, la gestión de respuestas personalizadas con códigos de estado HTTP se realiza principalmente mediante el uso de la clase **ResponseEntity**, la anotación **@ResponseStatus** en excepciones personalizadas y el manejo centralizado de errores con **@ControllerAdvice** y **@ExceptionHandler**.
- **ResponseEntity** permite construir respuestas HTTP completas, controlando el cuerpo, los encabezados y el código de estado

```
@GetMapping("/usuario/{id}")
public ResponseEntity<Usuario> obtenerUsuario(@PathVariable Long id) {
    Usuario usuario = servicioUsuario.obtenerPorId(id);
    if (usuario != null) {
        return ResponseEntity.ok(usuario); // 200 OK
    } else {
        return ResponseEntity.notFound().build(); // 404 Not Found
    }
}
```

## Respuestas personalizadas con códigos de estado HTTP

- En **Spring Boot**, la gestión de respuestas personalizadas con códigos de estado HTTP se realiza principalmente Puedes añadir encabezados personalizados y cualquier código de estado necesario

```
@GetMapping("/respuesta-personalizada")
public ResponseEntity<String> obtenerRespuestaPersonalizada() {
    HttpHeaders headers = new HttpHeaders();
    headers.add("Encabezado-Personalizado", "valor");
    return new ResponseEntity<>("Cuerpo de respuesta personalizado", headers,
HttpStatus.CREATED);
}
```

## Respuestas personalizadas con códigos de estado HTTP

- Puedes definir excepciones personalizadas y anotar la clase con `@ResponseStatus` para que Spring devuelva automáticamente el código de estado deseado

```
@ResponseStatus(HttpStatus.BAD_REQUEST)
public class DatosInvalidosException extends RuntimeException {
    public DatosInvalidosException(String mensaje) {
        super(mensaje);
    }
}
```

- Al lanzar esta excepción, Spring devolverá una respuesta **400 Bad Request** con el mensaje proporcionado



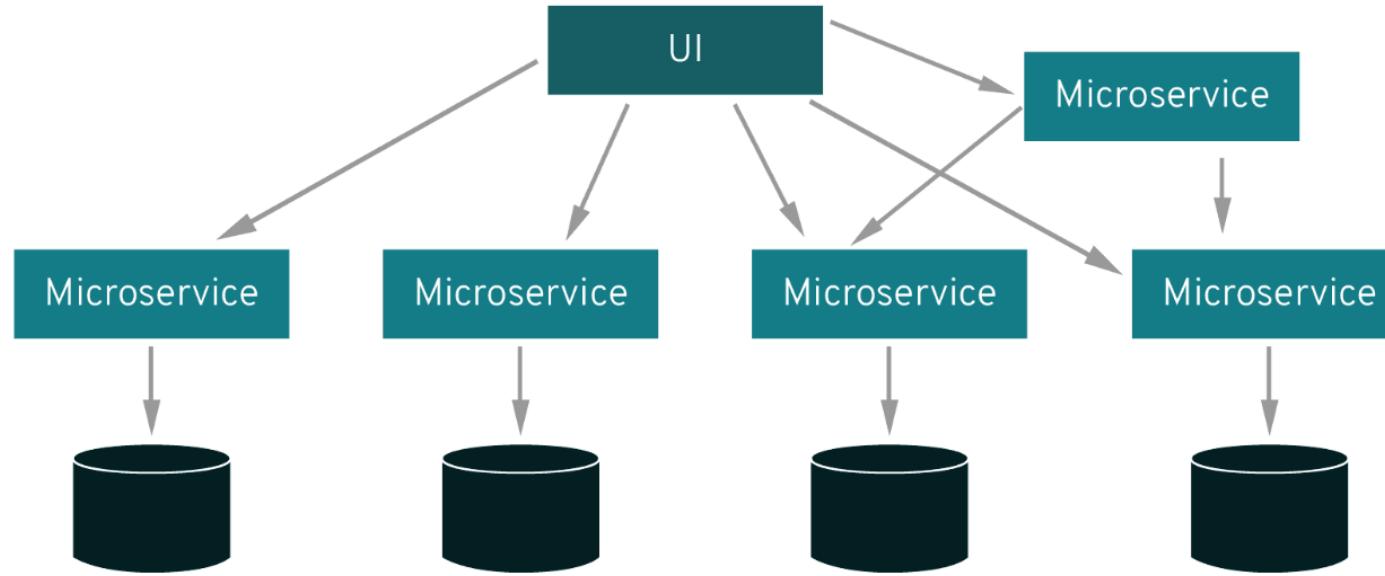
ibertech

end2end dedication

## Tema 3

### Comunicación entre Microservicios con Spring Cloud

## Comunicación entre microservicios



## Comunicación entre microservicios

- Los microservicios se crean de forma independiente y se comunican entre sí. Además, si se produce un error individual, este no provoca una interrupción de toda la aplicación.
- La comunicación se lleva a cabo a través de peticiones Rest.
- Dos formas de implementar el cliente de la petición:
  - RestTemplate
  - Feign

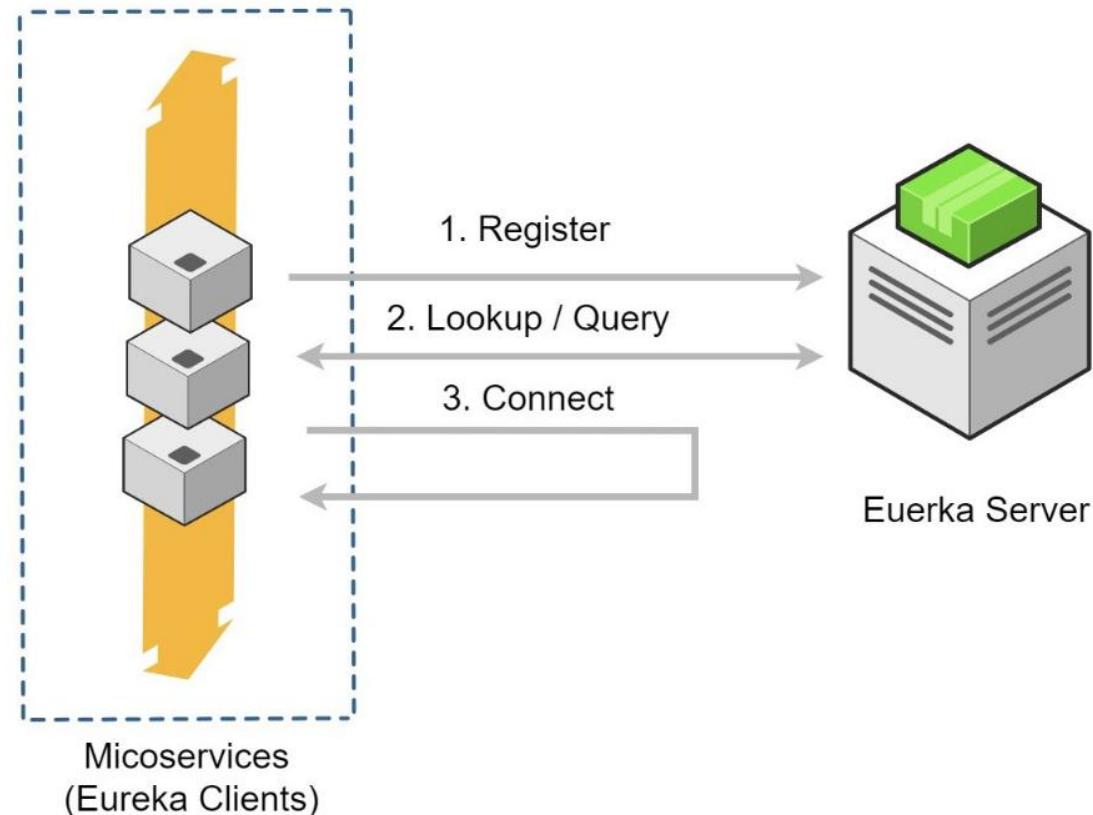
## Balanceo de carga con Ribbon



## Ribbon

- Ribbon es una librería usada para la intercomunicación de procesos, desarrollada por Netflix para su uso interno, y que se integra perfectamente con Apache Feign e Apache Eureka
- Ribbon nos da las siguientes capacidades:
  - Balanceo de carga, usando varios algoritmos que luego explicaremos detalladamente
  - Tolerancia a fallos. Ribbon determina dinámicamente qué servicios están corriendo y activos, al igual que cuales están caídos
  - Soporte de protocolo múltiple (HTTP, TCP, UDP) en un modelo asincrónico y reactivo
  - Almacenamiento en caché y procesamiento por lotes
  - Integración con los servicios de autodescubrimiento, como por ejemplo Eureka

## Eureka Netflix



## Eureka Netflix

- Eureka es un servicio rest que permite al resto de microservicios registrarse en su directorio.
- Esto es muy importante, puesto que no es Eureka quien registra los microservicios, sino los microservicios los que solicitan registrarse en el Eureka.

## Eureka Netflix

- Cuando un microservicio registrado en Eureka arranca, envía un mensaje a Eureka indicándole que está disponible.
- El servidor Eureka almacenará la información de todos los microservicios registrados así como su estado.
- La comunicación entre cada microservicio y el servidor Eureka se realiza mediante heartbeats cada X segundos.
- Si Eureka no recibe un heartbeat de un determinado microservicio, pasados 3 intervalos, el microservicio será eliminado del registro.
- Además de llevar el registro de los microservicios activos, Eureka también ofrece al resto de microservicios la posibilidad de "descubrir" y acceder al resto de microservicios registrados.
- Por ello Eureka es considerado un servicio de registro y descubrimiento de microservicios

## Comunicación asíncrona con Spring Cloud Stream

- La **gestión de la comunicación asíncrona con Spring Cloud Stream** permite que microservicios intercambien información mediante mensajes a través de brokers como RabbitMQ, Kafka, entre otros, siguiendo los patrones *message-driven* y *event-driven*. Esto desacopla los servicios y permite escalar, procesar eventos en tiempo real y mejorar la resiliencia del sistema
- ¿Cómo funciona Spring Cloud Stream?
  - Productores (publicadores) envían mensajes a un canal de salida.
  - Consumidores (suscriptores) reciben mensajes de un canal de entrada.
  - Los canales están conectados a un message broker (como RabbitMQ o Kafka) mediante binders

## Comunicación asíncrona con Spring Cloud Stream

- Ventajas principales:
  - Desacoplamiento: Los servicios no necesitan conocerse entre sí, solo el canal y el formato del mensaje.
  - Escalabilidad y resiliencia: Los mensajes pueden ser procesados por múltiples instancias y recuperarse ante fallos.
  - Asincronía real: El productor no espera la respuesta del consumidor, permitiendo procesar grandes volúmenes de eventos sin bloqueo

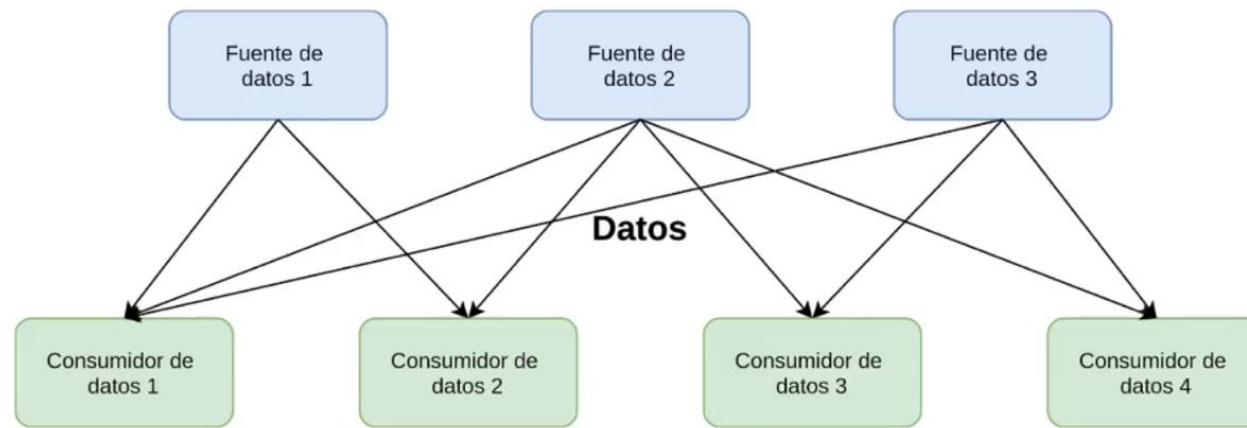
## Apache Kafka

- Apache Kakfa es una plataforma distribuida de transmisión de datos que permite publicar, almacenar y procesar flujos de datos en tiempo real
- Esta escrito en Java y Scala
- La idea inicial surge en Linkedin como solución a un problema interno de desarrollo
- En 2011 Apache se hace cargo del proyecto

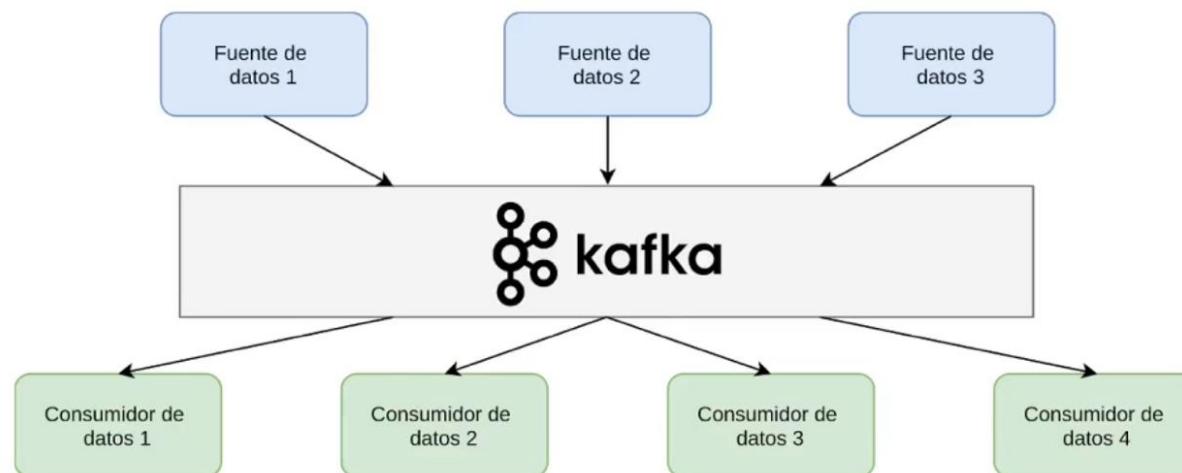


## Apache Kafka

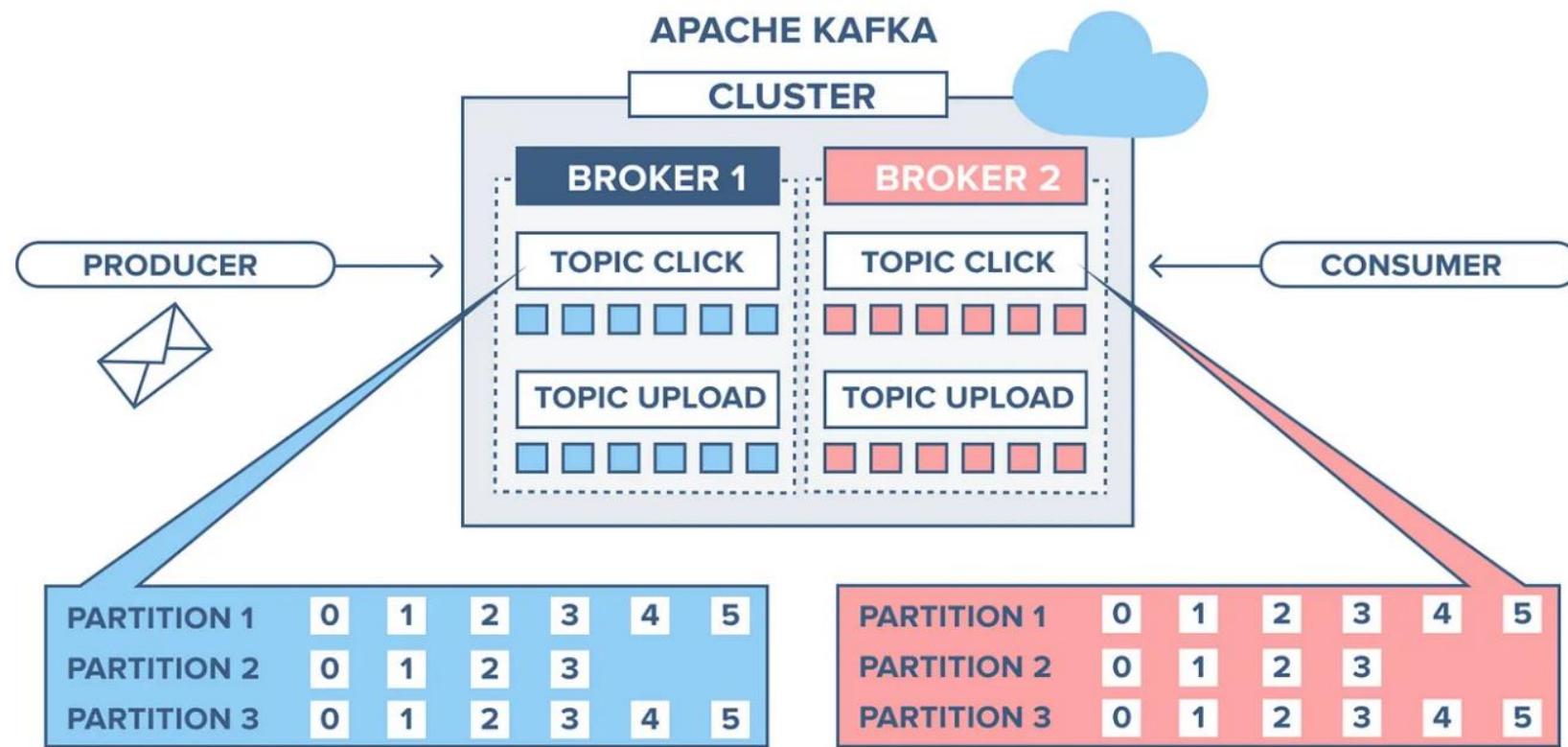
### El problema



### Apache Kafka



## Arquitectura de Apache Kafka





ibertech

end2end dedication

# Tema 4

Persistencia y Gestión del Estado  
en Microservicios Java

## Acceso a datos con JPA

- JPA es el acrónimo de Java Persistence API y se podría considerar como el estándar de los frameworks de persistencia.
- En JPA utilizamos anotaciones como medio de configuración.
- Consideramos una entidad al objeto que vamos a persistir o recuperar de una base de datos. Se puede ver una entidad como la representación de un registro de la tabla.
- Toda entidad ha de cumplir con los siguientes requisitos:
  - Debe implementar la interface Serializable
  - Ha de tener un constructor sin argumentos y este ha de ser público.
  - Todas las propiedades deben tener sus métodos de acceso get() y set().
- Para crear una entidad utilizamos la anotación @Entity, con ella marcamos un POJO como entidad.

## Acceso a datos con JPA

- Vamos a trabajar con una base de datos en memoria H2, para ello necesitamos agregar la siguiente dependencia al pom.xml

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
</dependency>
```

## Acceso a datos con JPA

- Spring nos facilita el trabajar con los datos incluyendo estas dependencias:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-rest</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

## Acceso a datos con JPA

- Debemos mapear la entidad a manejar en la base de datos:

```
@Entity  
public class Producto {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.AUTO)  
    private long id;  
  
    private String descripcion;  
    private double precio;
```

## Acceso a datos con JPA

- Y por ultimo tener el repositorio donde se generaran las queries de forma automatica;

```
@RepositoryRestResource(collectionResourceRel = "productos", path = "productos")
public interface ProductoRepository extends PagingAndSortingRepository<Producto, Long> {

    List<Producto> findByDescripcion(@Param("descripcion") String descripcion);

}
```

## Acceso a datos con JPA

- Una vez levantada la aplicación con Spring Boot ya podemos probarla con los siguientes comandos en consola:

```
// comandos a ejecutar en consola "con permiso de administrador"
// Acceso al servicio
// curl http://localhost:8080

// Consultar todos los productos
// curl http://localhost:8080/productos

// Alta de producto y consulta para su verificacion
// curl -i -X POST -H "Content-Type:application/json" -d '{"id": 1, "descripcion": "Macarrones", "precio":0.87}' http://localhost:8080/productos
// curl http://localhost:8080/productos

// Busqueda de un producto por su id y luego por su descripcion
// curl http://localhost:8080/productos/1
// curl http://localhost:8080/productos/search/findByDescripcion?descripcion=Macarrones

// Modificar precio del producto y consulta para su verificacion
// curl -X PUT -H "Content-Type:application/json" -d '{"id": 1, "descripcion": "Macarrones", "precio":0.98}' http://localhost:8080/productos/1
// curl http://localhost:8080/productos/1

// Borrar un producto y consulta para su verificacion
// curl -X DELETE http://localhost:8080/productos/1
// curl http://localhost:8080/productos
```

## Acceso a datos con MongoDB

- MongoDB, a pesar de ser una base de datos relativamente joven (su desarrollo empezó en octubre de 2007) se ha convertido en todo un referente a la hora de usar bases de datos NoSQL y está listo para entornos de producción ágiles, de alto rendimiento y con gran carga de trabajo.
- En lugar de guardar los datos en tablas como se hace en las bases de datos relacionales con estructuras fijas, las bases de datos NoSQL, como MongoDB, guarda estructuras de datos en documentos con formato JSON y con un esquema dinámico (MongoDB llama ese formato BSON).
- Ejemplo de documento almacenado en MongoDB:

```
{  
    "_id": ObjectId("4efa8d2b7d284dad101e4bc7"),  
    "Last Name": "PELLERIN",  
    "First Name": "Franck",  
    "Age": 29,  
    "Address": {  
        "Street": "1 chemin des Loges",  
        "City": "VERSAILLES"  
    }  
}
```

## Acceso a datos con MongoDB

- Para descargar MongoDB debemos irnos a su pagina de descargas: <https://www.mongodb.com/download-center/community> donde encontrareis la versión adecuada a vuestra plataforma.
- Una vez descargados los binarios de MongoDB para Windows, se extrae el contenido del fichero descargado (ubicado normalmente en el directorio de descargas) en C:\.
- Renombra la carpeta a mongodb: C:\mongodb
- MongoDB es autónomo y no tiene ninguna dependencia del sistema por lo que se puede usar cualquier carpeta que elijas. La ubicación predeterminada del directorio de datos para Windows es "C:\data\db". Crea esta carpeta.
- Para iniciar MongoDB, ejecutar desde la Línea de comandos

```
C:\mongodb\bin\mongod.exe
```

- Esto iniciará el proceso principal de MongoDB. El mensaje "waiting for connections" indica que el proceso mongod.exe se está ejecutando con éxito.

## Acceso a datos con MongoDB

- Dependencias necesarias:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-rest</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
```

## Acceso a datos con MongoDB

```
public class Producto {  
    @Id  
    private String id;  
  
    private String descripcion;  
    private double precio;  
  
    public String getDescripcion() {  
        return descripcion;  
    }  
  
    public void setDescripcion(String descripcion) {  
        this.descripcion = descripcion;  
    }  
  
    public double getPrecio() {  
        return precio;  
    }  
  
    public void setPrecio(double precio) {  
        this.precio = precio;  
    }  
}
```

## Acceso a datos con MongoDB

- Producto repository:

```
@RepositoryRestResource(collectionResourceRel = "productos", path = "productos")
public interface ProductoRepository extends MongoRepository<Producto, String> {

    List<Producto> findByLastName(@Param("descripcion") String descripcion);

}
```

## Acceso a datos con MongoDB

```
public interface ProductoRepository extends MongoRepository<Producto, String> {  
    public List<Producto> findByDescripcion(String descripcion);  
}
```

## Transacciones distribuidas

- La **gestión de transacciones distribuidas en microservicios** es uno de los retos más complejos en arquitecturas modernas. Los dos enfoques principales para abordar este desafío son el **Two-Phase Commit (2PC)** y el **patrón Saga**. Ambos buscan mantener la consistencia de los datos, pero difieren radicalmente en su funcionamiento, ventajas y limitaciones.

## Two-Phase Commit (2PC)

- 2PC es un protocolo clásico para garantizar la atomicidad y consistencia de transacciones que involucran múltiples sistemas o bases de datos. Se basa en un coordinador central que gestiona dos fases

Fase	Acción del Coordinador	Acción de los Participantes
Prepare	Solicita a todos "¿pueden preparar?"	Preparan la transacción y responden "Sí/No"
Commit/ Abort	Si todos dicen "Sí", ordena commit; si alguno dice "No", ordena rollback	Ejecutan commit o rollback según la orden

- Ventajas:
  - Garantiza consistencia fuerte y atomicidad.
  - Transparente para la lógica de negocio si el soporte está integrado en la plataforma.
- Desventajas:
  - Bloqueante: Los recursos quedan bloqueados hasta que todos los participantes responden, lo que puede afectar el rendimiento.
  - Punto único de fallo: El coordinador es crítico; si falla, puede dejar el sistema en estado incierto.
  - Escalabilidad limitada: No es adecuado para transacciones largas ni sistemas altamente distribuidos.

## Saga Pattern

- El patrón Saga descompone una transacción global en una **secuencia de transacciones locales** independientes, cada una gestionada por un microservicio. Si alguna etapa falla, se ejecutan **transacciones compensatorias** para deshacer los cambios previos y restaurar la consistencia.
- Modos de coordinación:
  - Choreography: Cada servicio publica eventos que disparan acciones en otros servicios. No hay un coordinador central.
  - Orchestration: Un orquestador central gestiona el flujo de la saga, invocando servicios y compensaciones según sea necesario.
- Ventajas:
  - Escalabilidad y resiliencia: No hay bloqueo global de recursos; cada servicio gestiona su propia transacción.
  - Adecuado para transacciones largas: Permite procesos de negocio que pueden durar minutos u horas.
  - Desacoplamiento: Los servicios están menos acoplados y pueden evolucionar de forma independiente.
- Desventajas:
  - Garantiza consistencia eventual, no fuerte.
  - Mayor complejidad en el diseño: se deben definir transacciones compensatorias idempotentes y manejar fallos parciales.

## Two-Phase Commit y Saga Pattern

Característica	Two-Phase Commit (2PC)	Saga Pattern
Consistencia	Fuerte (ACID)	Eventual
Escalabilidad	Limitada	Alta
Duración transacción	Corta	Larga
Bloqueo de recursos	Sí	No
Complejidad	Menor (si la plataforma lo soporta)	Mayor (requiere lógica compensatoria)
Tolerancia a fallos	Menor (punto único de fallo)	Mayor (sin bloqueo global)
Uso típico	Bancos, pagos, sistemas críticos	E-commerce, reservas, workflows

- Recomendaciones:
  - Usa 2PC si necesitas consistencia fuerte, las transacciones son cortas y puedes tolerar el bloqueo de recursos y la dependencia de un coordinador central.
  - Usa Saga si priorizas escalabilidad, resiliencia y disponibilidad, y puedes trabajar con consistencia eventual. Es el patrón recomendado en la mayoría de arquitecturas de microservicios modernos.

## Patrón CQRS (Command Query Responsibility Segregation)

- El **patrón CQRS (Command Query Responsibility Segregation)** es una arquitectura que **separa la responsabilidad de las operaciones de escritura (commands) y lectura (queries)** en la persistencia de datos, permitiendo optimizar y escalar cada una de forma independiente.
- **Modelo de comandos (Write Model):**
  - Gestiona las operaciones de crear, actualizar y eliminar datos.
  - Suele incluir lógica de negocio, validaciones y reglas complejas.
  - Persiste los cambios en una base de datos de escritura optimizada para operaciones transaccionales y consistencia
- **Modelo de consultas (Read Model):**
  - Gestiona las operaciones de lectura de datos.
  - Utiliza una base de datos de lectura optimizada para consultas rápidas y escalables, como vistas materializadas, réplicas o bases NoSQL
  - No contiene lógica de negocio ni validaciones; solo devuelve DTOs o vistas para el consumidor

## Spring Cache

- Spring Cache es un módulo del framework Spring que permite almacenar en memoria los resultados de métodos que se ejecutan con frecuencia, evitando así realizar operaciones repetitivas o consultas innecesarias a la base de datos u otros recursos externos. Esto optimiza el rendimiento de la aplicación, reduce el consumo de recursos y mejora los tiempos de respuesta.
- ¿Cómo funciona Spring Cache?
  - Cuando anotas un método con `@Cacheable`, Spring guarda automáticamente el resultado de la primera ejecución en una caché identificada por un nombre (por ejemplo, "miCache").
  - Si el método se vuelve a invocar con los mismos parámetros, Spring recupera el resultado directamente de la caché en vez de ejecutar el método de nuevo.
  - Esto es especialmente útil para métodos cuyos resultados no cambian frecuentemente, como consultas a tablas de configuración, catálogos o listas de parámetros.

## Spring Cache

- En este ejemplo, la primera vez que se consulta una categoría, el resultado se almacena en la caché. Las siguientes veces, si se consulta la misma categoría, el resultado se obtiene de la memoria, sin acceder a la base de datos

```
@Service
public class ProductoServicio {
    @Cacheable("productosPorCategoria")
    public List<Producto> obtenerProductosPorCategoria(String categoria) {
        // Consulta a base de datos
        return productoRepository.findByCategoria(categoria);
    }
}
```

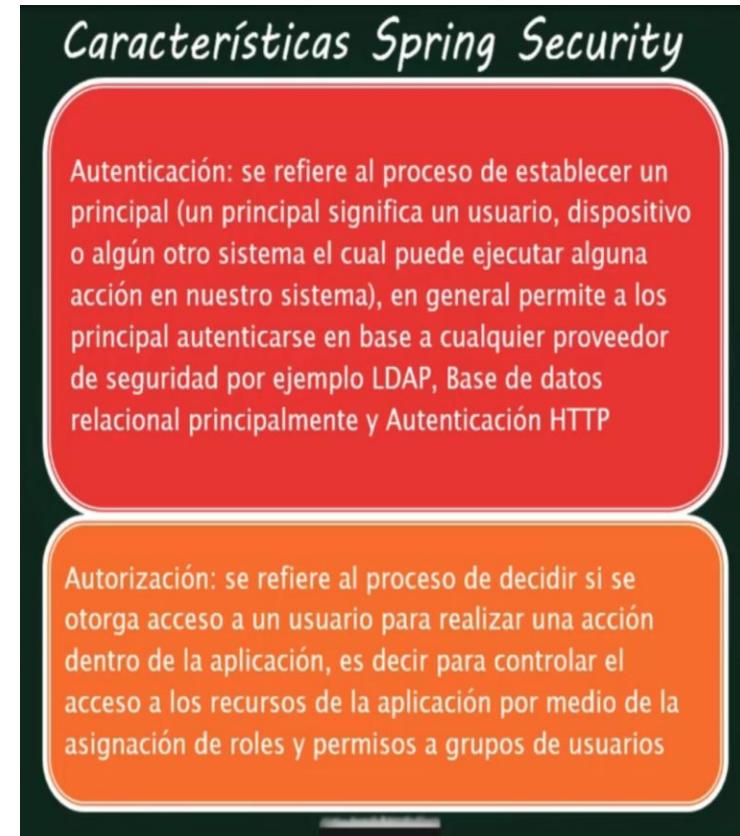
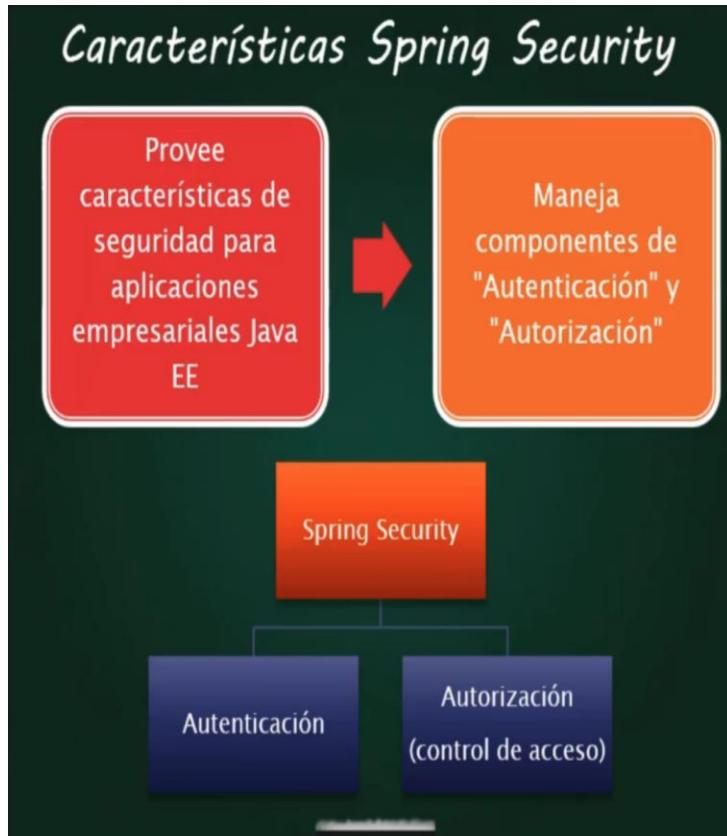
## Redis

- El uso de **Redis** es una de las estrategias más efectivas para **mejorar el rendimiento de acceso a datos** en aplicaciones modernas. Redis es una base de datos en memoria, clave-valor, diseñada para ofrecer acceso ultrarrápido y baja latencia, lo que la convierte en una solución ideal para escenarios donde la velocidad de consulta es crítica
- Casos de uso comunes:
  - Caché de resultados de consultas frecuentes (por ejemplo, productos más vendidos, perfiles de usuario, resultados de búsqueda)
  - Gestión de sesiones de usuario y almacenamiento temporal de información de autenticación
  - Almacenamiento de rankings, contadores y estadísticas en tiempo real
  - Soporte para mensajería en tiempo real y pub/sub

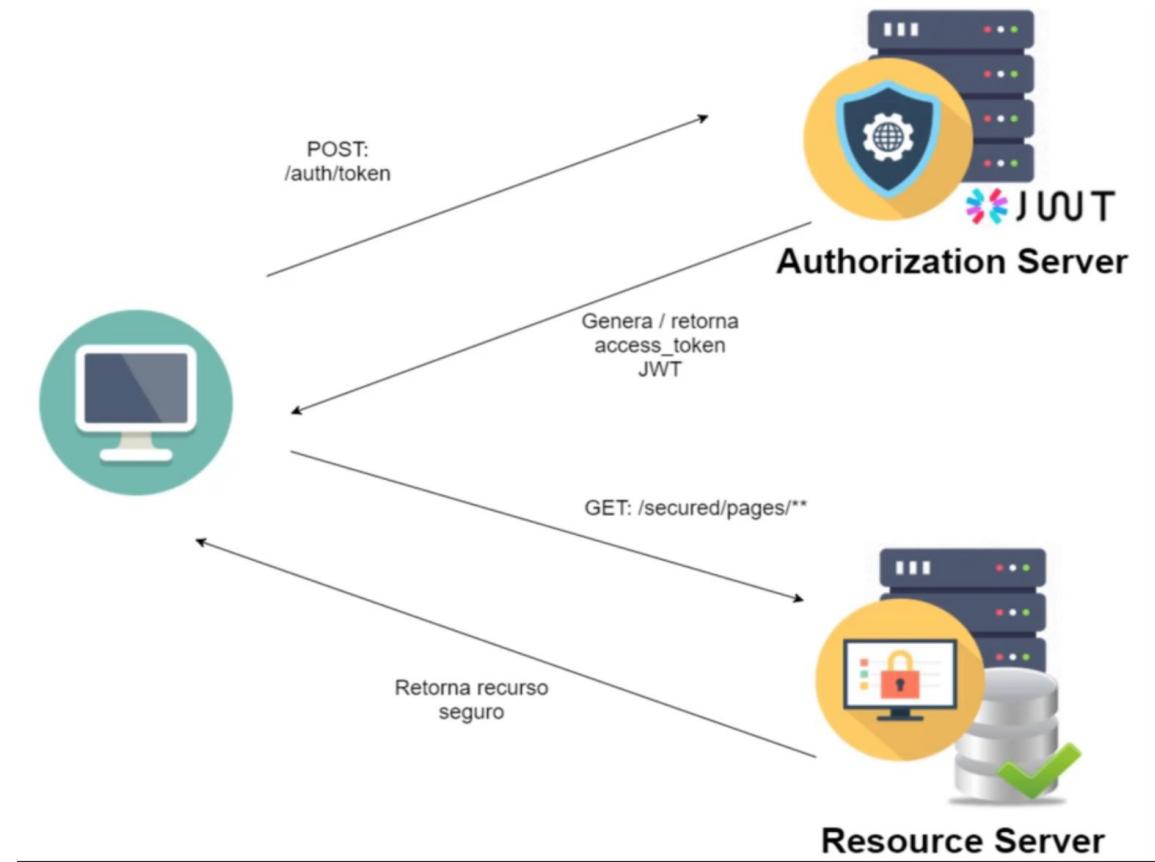
# Tema 5

Seguridad y Autenticación en  
Microservicios con Spring Security

## Spring Security



## Oauth2



## Authorization Server

url: POST /auth/token

header:

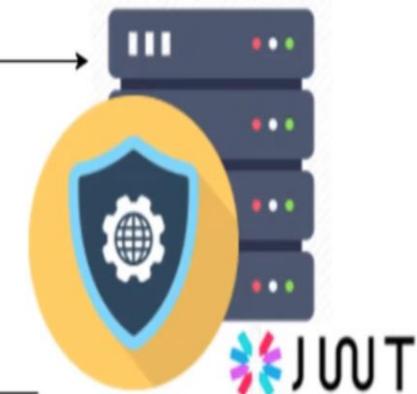
- Authorization: Basic Base64(client\_id:client\_secret)
- Content-Type: application/x-www-form-urlencoded

body:

- ```
grant_type = password
username = andres
password = 12345
```



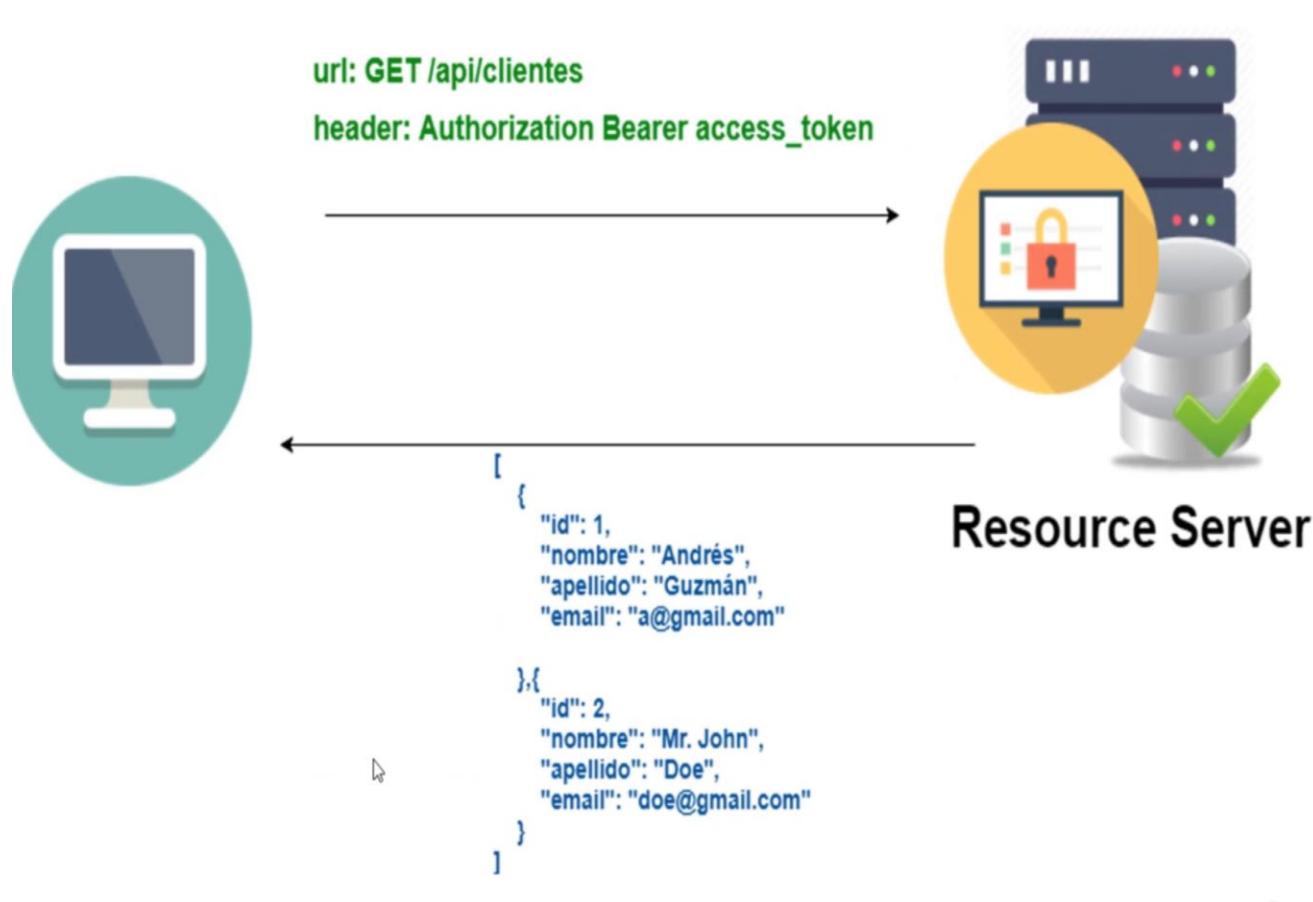
```
{
  "access_token": "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9...",
  "token_type": "bearer",
  "refresh_token": "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9...",
  "expires_in": 3599,
  "scope": "read write",
  "jti": "58efb674-46e6-4f6b-bbf0-e92e21e4b34a"
}
```



Authorization Server

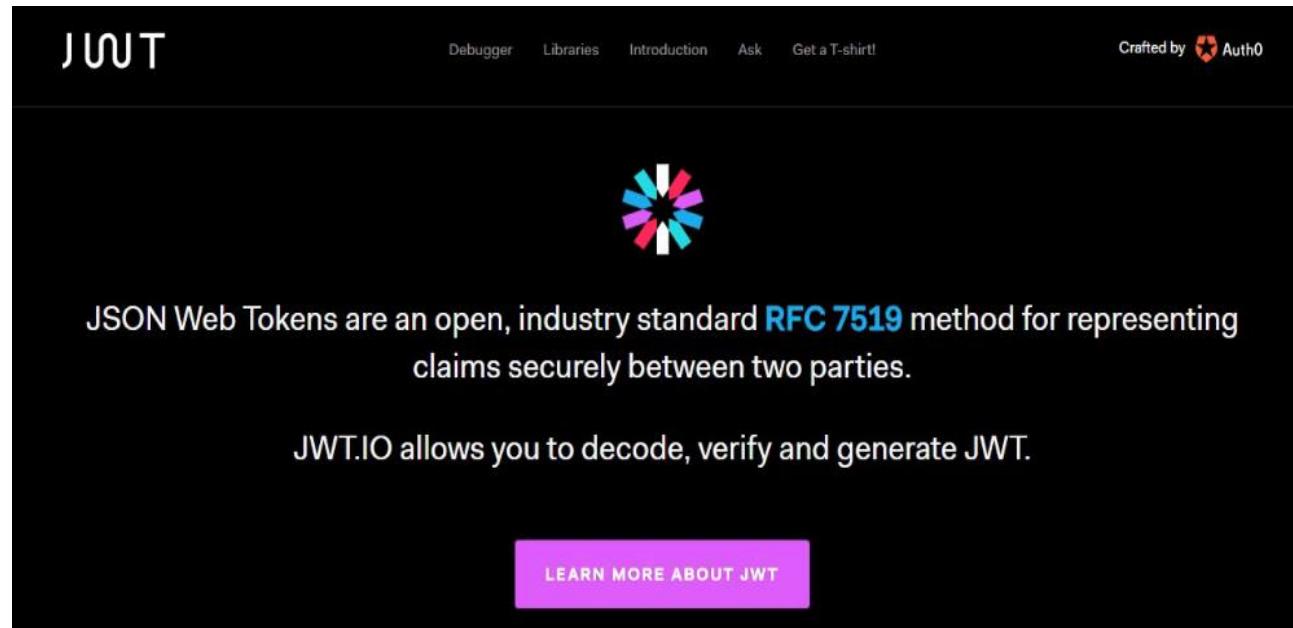
Diagram

## Resource Server

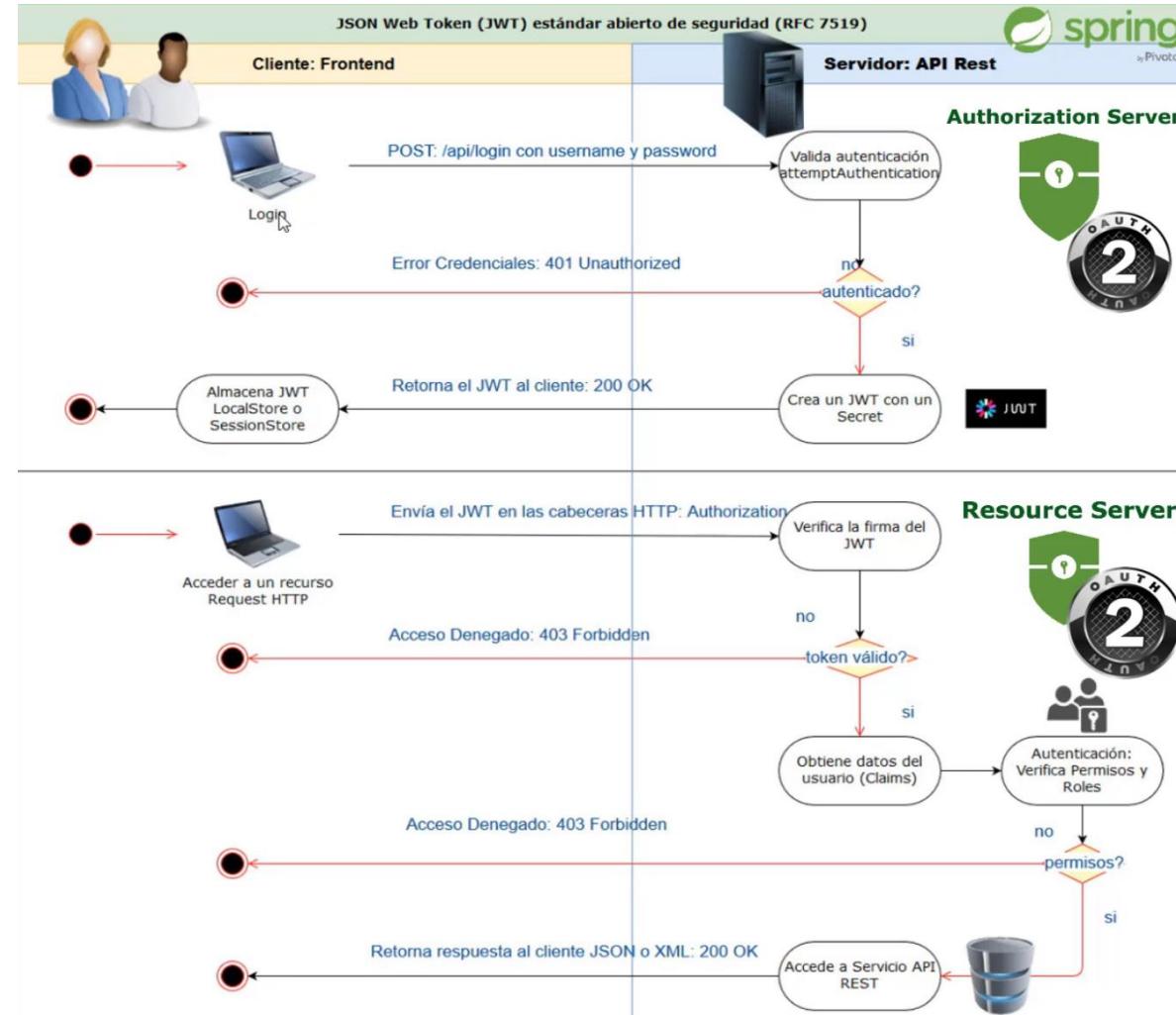


## JWT

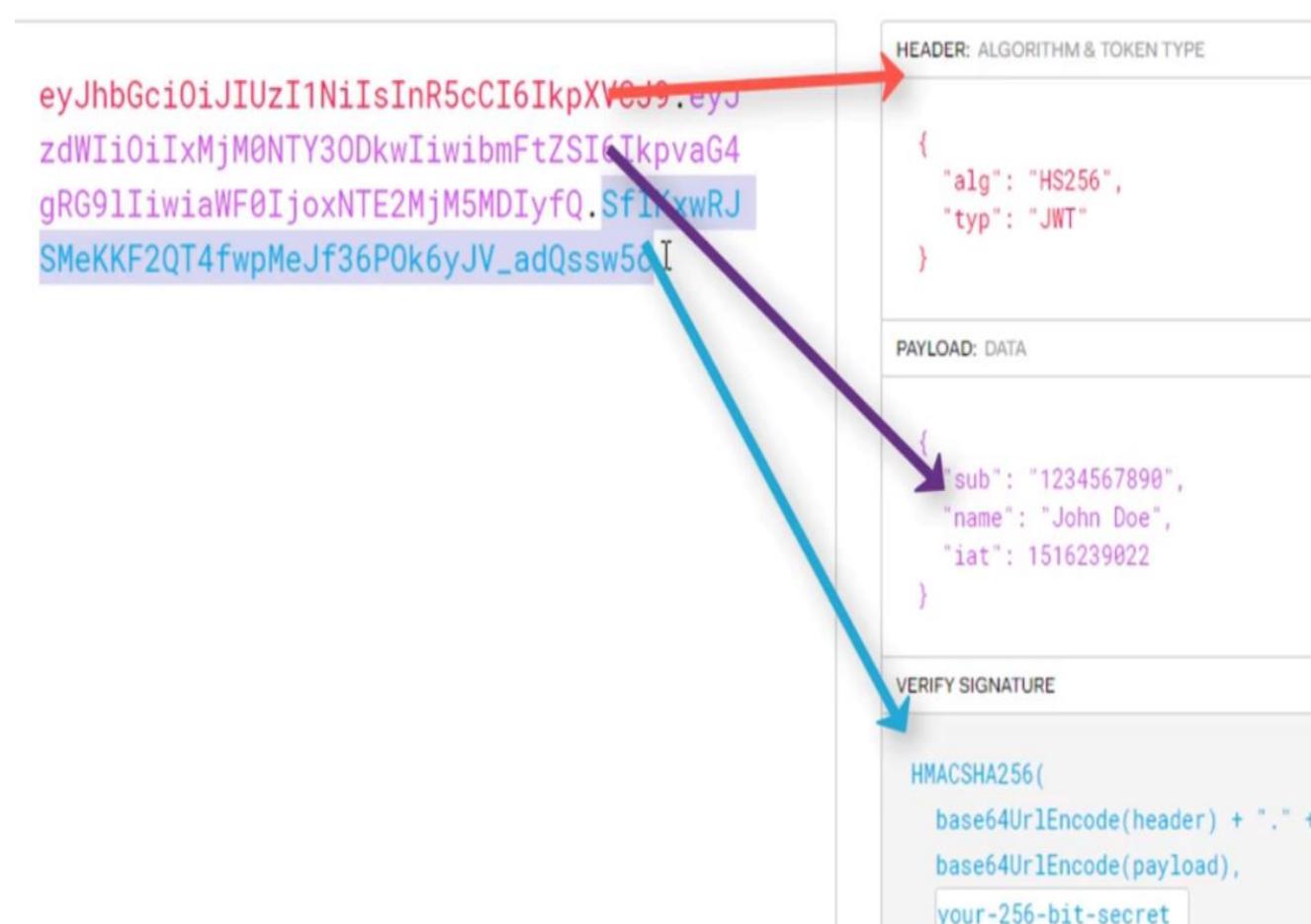
- JWT es un estándar abierto para implementar seguridad en nuestras aplicaciones API Rest.



# JWT



## Token



## Autenticación con Google o Facebook

- Para **integrar proveedores externos de autenticación como Google o Facebook en una aplicación Spring Boot**, se utiliza el soporte nativo de **Spring Security OAuth2 Client**. Esto permite que los usuarios inicien sesión usando sus cuentas de Google o Facebook de forma segura y sencilla, sin que la aplicación gestione directamente contraseñas.
- Spring Boot permite integrar autenticación social con Google y Facebook fácilmente configurando los datos de cliente y usando las rutas /oauth2/authorization/google y /oauth2/authorization/facebook.
- Spring Security gestiona automáticamente el flujo OAuth2, la obtención de los datos del usuario y la sesión autenticada, facilitando la implementación de login social seguro y moderno

## Implementación de SSL/TLS

- La **implementación de SSL/TLS en microservicios con Spring Boot** es fundamental para proteger la comunicación entre servicios y con clientes externos, asegurando la confidencialidad e integridad de los datos en tránsito. Spring Boot facilita esta configuración tanto para servidores como para clientes.
- Los pasos a seguir son:
  1. Generar o adquirir un certificado SSL/TLS
  2. Configuración del servidor Spring Boot para HTTPS
  3. Proteger endpoints REST
  4. Consumo seguro de APIs entre microservicios

## Protección contra ataques comunes

- Spring Boot, junto con Spring Security y buenas prácticas de desarrollo, proporciona mecanismos robustos para **proTEGER APLICACIONES CONTRA ATAQUES COMUNES COMO CSRF, XSS Y SQL INJECTION.**
  - CSRF o Cross-Site Request Forgery (Falsificación de Peticiones Cross-Site), es un tipo de ataque cibernético que ocurre cuando un usuario, ya autenticado en un sitio web, es engañado para realizar acciones no deseadas en ese sitio, sin su conocimiento o consentimiento.
  - El cross-site scripting (XSS) es un tipo de vulnerabilidad de seguridad web que permite a los atacantes injectar código malicioso (generalmente JavaScript) en sitios web legítimos. Este código malicioso se ejecuta en el navegador del usuario, permitiendo al atacante acceder a información sensible, realizar acciones en nombre del usuario o redirigirlo a sitios web maliciosos
  - Las inyecciones de SQL (o SQLI) se producen cuando el hacker introduce o inyecta en el sitio web código SQL malicioso, un tipo de malware que se conoce como la carga útil, y consigue subrepticiamente que envíe ese código a su base de datos como si de una consulta legítima se tratara.

## Protección contra CSRF (Cross-Site Request Forgery)

- Spring Security habilita la protección CSRF por defecto en aplicaciones web, generando y validando automáticamente un token CSRF en formularios y peticiones mutables (POST, PUT, DELETE).
- Solo solicitudes legítimas que incluyan el token correcto serán aceptadas por el servidor, bloqueando intentos de ejecución no autorizada desde sitios externos.
- Si tu aplicación expone solo APIs REST (stateless), puedes desactivar CSRF, pero para aplicaciones web tradicionales debe mantenerse activo

## Protección contra XSS (Cross-Site Scripting)

- Validación y sanitización de entradas: Siempre valida y limpia los datos recibidos del usuario antes de almacenarlos o mostrarlos.
- Escapado de salidas: Usa frameworks y motores de plantillas (Thymeleaf, JSP, etc.) que escapan automáticamente el contenido mostrado en vistas, evitando la ejecución de scripts maliciosos
- Cabeceras de seguridad: Spring Security añade por defecto cabeceras como X-XSS-Protection y permite configurar políticas CSP (Content Security Policy) para restringir la ejecución de scripts no autorizados

## Protección contra SQL Injection

- Uso de consultas preparadas y parámetros: Utiliza siempre consultas preparadas o frameworks ORM como JPA/Hibernate, que separan la lógica SQL de los datos del usuario, evitando la inyección de código malicioso.
- Validación de entradas: Asegúrate de validar y sanear todos los datos que provienen del usuario antes de utilizarlos en consultas a la base de datos.
- Principio de privilegios mínimos: Limita los permisos de la cuenta de base de datos utilizada por la aplicación, restringiendo las acciones que un posible atacante podría realizar en caso de vulnerabilidad.
- Evita concatenar cadenas con datos del usuario en sentencias SQL.

# Tema 6

Despliegue de Microservicios con  
Docker y Kubernetes

## Docker

- Docker es una plataforma de código abierto que permite desarrollar, enviar y ejecutar aplicaciones dentro de contenedores. Estos contenedores son unidades ligeras y portables que incluyen todo lo necesario para ejecutar una aplicación: código, bibliotecas, dependencias y configuración, garantizando que la aplicación funcione de manera idéntica en cualquier entorno que soporte Docker
- Ventajas de Docker:
  - Portabilidad entre diferentes entornos (desarrollo, pruebas, producción)
  - Aislamiento de aplicaciones, evitando conflictos de dependencias
  - Facilidad para escalar y actualizar aplicaciones
  - Eficiencia en el uso de recursos del sistema

## Arquitectura de Docker

- Docker sigue una arquitectura cliente-servidor:
  - Docker Daemon (dockerd): Es el proceso principal que gestiona los contenedores y escucha las peticiones de la API de Docker.
  - Cliente de Docker (docker CLI): Es la interfaz de línea de comandos que permite a los usuarios interactuar con el daemon para construir, ejecutar y administrar contenedores.
  - API de Docker: Permite la comunicación entre el cliente y el daemon, facilitando la automatización y la integración con otras herramientas.

## Componentes clave de Docker

- **Dockerfile:** Archivo de texto que define las instrucciones para construir una imagen de Docker (por ejemplo, qué sistema base usar, qué dependencias instalar, qué comandos ejecutar).
- **Imagen:** Plantilla de solo lectura creada a partir de un Dockerfile. Contiene el sistema de archivos y las dependencias necesarias para ejecutar una aplicación.
- **Contenedor:** Instancia en ejecución de una imagen. Es el entorno aislado donde se ejecuta la aplicación.
- **Docker Hub:** Repositorio público (o privado) donde se almacenan y comparten imágenes de Docker.

## Diferencias de Docker con las Máquinas Virtuales

| Docker (Contenedores)                   | Máquinas Virtuales (VM)                   |
|-----------------------------------------|-------------------------------------------|
| Virtualiza a nivel de sistema operativo | Virtualiza hardware completo              |
| Comparte el kernel del host             | Cada VM tiene su propio sistema operativo |
| Más ligeros y rápidos                   | Más pesados y lentos                      |
| Menor consumo de recursos               | Mayor consumo de recursos                 |

## Despliegue de microservicios Java en Kubernetes

- El **despliegue de microservicios Java en Kubernetes** es un proceso que implica la creación de imágenes Docker, la definición de recursos de Kubernetes mediante archivos YAML y la gestión de la comunicación, escalabilidad y resiliencia de los servicios. A continuación se describen los pasos y mejores prácticas clave para desplegar aplicaciones Java (por ejemplo, Spring Boot) en un clúster Kubernetes:
  - Empaquetado del microservicio Java
  - Definición de recursos Kubernetes: Deployment, Service, Ingress
  - Despliegue en el clúster

## Helm para gestionar el despliegue automatizado

- Helm actúa como gestor de paquetes para Kubernetes, permitiendo empaquetar, versionar, parametrizar y reutilizar despliegues mediante “charts”, facilitando tanto la automatización como la gestión de configuraciones por entorno.
- Ventajas de usar Helm con Spring Boot
  - **Despliegue automatizado y reproducible:** Helm permite instalar, actualizar y revertir aplicaciones de forma sencilla y controlada, asegurando consistencia en todos los entornos
  - **Gestión centralizada de configuración:** Los valores específicos de cada entorno (dev, test, prod) se gestionan en archivos values.yaml, evitando cambios en el código fuente y facilitando la personalización
  - **Versionado y rollback:** Cada despliegue se versiona, permitiendo revertir a versiones anteriores si es necesario, lo que mejora la resiliencia y la capacidad de respuesta ante errores
  - **Reutilización y estandarización:** Los charts pueden compartirse y reutilizarse entre proyectos y equipos, acelerando la adopción de mejores prácticas y la estandarización de despliegues
  - **Soporte de hooks y gestión de ciclo de vida:** Helm permite ejecutar scripts o tareas personalizadas antes o después de cada fase del despliegue (pre-install, post-install, etc.)

## Autoescalado de microservicios en Kubernetes con HPA

- El **Horizontal Pod Autoscaler (HPA)** en Kubernetes permite que los microservicios (como los desarrollados con Spring Boot) escalen automáticamente el número de réplicas de sus pods en función de métricas como el uso de CPU, memoria o métricas personalizadas. Esto garantiza que tu aplicación pueda responder eficientemente a picos de demanda y reducir recursos cuando la carga disminuye, optimizando costes y disponibilidad
- Como funciona HPA?
  - **Monitoriza métricas** (por ejemplo, CPU o memoria) de los pods de un Deployment, ReplicaSet o StatefulSet.
  - **Ajusta automáticamente el número de pods** entre un mínimo y un máximo configurado, según los umbrales definidos.
  - **Escalado horizontal:** Si la carga aumenta y supera el umbral, Kubernetes crea más pods; si baja, reduce el número de pods

## Balanceo de carga en Kubernetes

- Kubernetes gestiona el balanceo de carga entre servicios mediante la abstracción de Services, distribuyendo el tráfico entre los pods asociados y garantizando disponibilidad, escalabilidad y eficiencia.
- El tipo de Service y el uso de Ingress o herramientas avanzadas permiten adaptar el balanceo a las necesidades específicas de cada aplicación
  - **ClusterIP**: balancea el tráfico entre pods dentro del clúster.
  - **NodePort**: expone el servicio en un puerto específico de cada nodo, permitiendo el acceso externo y el balanceo entre nodos.
  - **LoadBalancer**: solicita un balanceador de carga externo (de un proveedor cloud) que dirige el tráfico a los servicios dentro del clúster.
  - **Ingress**: permite reglas avanzadas de ruteo HTTP/HTTPS, balanceando el tráfico a diferentes servicios según el dominio o la ruta.



ibertech

end2end dedication

# Tema 7

## Monitorización y Logging en Microservicios Java

## Monitorización del rendimiento de microservicios con Prometheus y Grafana

- La **monitorización del rendimiento de microservicios con Prometheus y Grafana** es una práctica esencial para garantizar la salud, disponibilidad y eficiencia de sistemas distribuidos. Estas herramientas, ampliamente adoptadas en entornos cloud-native y Kubernetes, permiten recopilar, analizar y visualizar métricas en tiempo real, facilitando la detección proactiva de problemas y la optimización continua de los servicios.
- **Prometheus**: es una herramienta de monitorización y alertas que recopila métricas de microservicios mediante peticiones HTTP (modelo pull) y las almacena como series temporales en su base de datos interna. Permite definir alertas y realizar consultas avanzadas usando PromQL.
- **Grafana**: es una plataforma de visualización que se conecta a Prometheus para mostrar las métricas en dashboards interactivos y personalizables. Permite crear paneles para observar el rendimiento, analizar tendencias y recibir alertas visuales

## Configuración de métricas para aplicaciones Spring Boot con Micrometer

- Para **configurar métricas en aplicaciones Spring Boot con Micrometer**, debes aprovechar la integración nativa de Spring Boot Actuator y la flexibilidad de Micrometer para recolectar y exponer métricas tanto estándar como personalizadas, compatibles con sistemas como Prometheus, Grafana, InfluxDB y otros

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

```
management.endpoints.web.exposure.include=prometheus
```

## Configuración de métricas para aplicaciones Spring Boot con Micrometer

```
import io.micrometer.core.instrument.MeterRegistry;
import org.springframework.stereotype.Service;

@Service
public class MyService {
    private final MeterRegistry meterRegistry;

    public MyService(MeterRegistry meterRegistry) {
        this.meterRegistry = meterRegistry;
    }

    public void doSomething() {
        meterRegistry.counter("myservice.invocations").increment();
        // lógica del servicio...
    }
}
```

- Esto generará una métrica llamada myservice\_invocations que puedes consultar en /actuator/metrics/myservice.invocations y en /actuator/prometheus

## Visualización de métricas en Grafana y configuración de alertas

- Visualización de métricas en Grafana:
  - Grafana es una plataforma líder para la visualización de métricas provenientes de sistemas como Prometheus, InfluxDB y otros.
  - Permite crear dashboards personalizados donde puedes mostrar métricas clave (CPU, memoria, latencia, etc.) en gráficos de líneas, barras, medidores, tablas y más
  - Los paneles se pueden organizar, personalizar (colores, etiquetas, títulos) y adaptar para mostrar información relevante de un vistazo, facilitando el monitoreo en tiempo real y el análisis histórico
  - Puedes ajustar las consultas según la fuente de datos y las necesidades del sistema, y modificar visualmente los parámetros de cada gráfico para mejorar la comprensión y el impacto visual

## Visualización de métricas en Grafana y configuración de alertas

- Configuración de alertas en Grafana:
  1. Abre el panel donde visualizas la métrica de interés.
  2. Define una consulta (por ejemplo, uso de CPU, latencia, etc.).
  3. Haz clic en la opción para añadir alerta y establece:
    - Nombre de la alerta.
    - Frecuencia de evaluación (por ejemplo, cada 1 minuto).
    - Condición (por ejemplo, si el valor medio de CPU supera el 90% durante 5 minutos)
  4. Configura el canal de notificación: Grafana soporta correo electrónico, Slack, Teams, Telegram y otros
  5. Guarda la alerta y, si lo deseas, realiza un test para verificar su funcionamiento

## Implementación de tracing distribuido con Spring Cloud Sleuth y Jaeger

- El tracing distribuido es fundamental para observar, depurar y analizar el flujo de peticiones en arquitecturas de microservicios. Spring Cloud Sleuth y Jaeger permiten instrumentar aplicaciones Spring Boot para obtener visibilidad completa de las transacciones distribuidas.
- **Spring Cloud Sleuth** es una librería que añade identificadores de traza (**traceId**) y de operación (**spanId**) a las peticiones que atraviesan los microservicios. Estos identificadores se propagan automáticamente entre servicios, permitiendo correlacionar logs y reconstruir el recorrido completo de una petición
- **Jaeger** es una plataforma open source para tracing distribuido desarrollada por Uber. Permite recolectar, almacenar y visualizar trazas generadas por los microservicios, facilitando el análisis de latencias, dependencias y cuellos de botella en sistemas distribuidos
- Arquitectura típica:
  - Microservicios instrumentados con Sleuth: Cada servicio añade automáticamente los identificadores de traza y propaga los contextos.
  - Jaeger Agent/Collector: Recibe las trazas y las almacena.
  - Jaeger UI: Permite consultar y analizar visualmente las trazas completas.

## Gestión de logs centralizados con Elasticsearch, Fluentd y Kibana

- La gestión centralizada de logs es fundamental en arquitecturas modernas, especialmente en entornos de microservicios y Kubernetes, donde múltiples aplicaciones y servicios generan grandes volúmenes de registros. El stack EFK (Elasticsearch, Fluentd y Kibana) es una de las soluciones más populares y robustas para recolectar, almacenar, analizar y visualizar logs de manera centralizada.
- Que es el stack EFK?
  - **Elasticsearch:** Motor de búsqueda y análisis distribuido, encargado de almacenar e indexar los logs para búsquedas y análisis rápidos.
  - **Fluentd:** Agente de recolección de logs que recopila, transforma y reenvía los registros desde diversas fuentes hacia Elasticsearch.
  - **Kibana:** Interfaz web para visualizar, explorar y crear dashboards a partir de los datos almacenados en Elasticsearch

## Gestión de logs centralizados con Elasticsearch, Fluentd y Kibana

- Arquitectura y flujo de datos:
  - **Recolección:** Fluentd se despliega en cada nodo (por ejemplo, como DaemonSet en Kubernetes) y recopila logs de aplicaciones, contenedores y del sistema operativo.
  - **Transformación y envío:** Fluentd puede filtrar, transformar y enriquecer los logs antes de enviarlos a Elasticsearch.
  - **Almacenamiento e indexación:** Elasticsearch almacena los logs en índices, permitiendo búsquedas complejas y agrupaciones.
  - **Visualización:** Kibana se conecta a Elasticsearch y permite consultar, visualizar y crear alertas o dashboards personalizados sobre los logs centralizados

## Gestión de logs centralizados con Elasticsearch, Fluentd y Kibana

- Ventajas del stack EFK:
  - **Centralización:** Todos los logs se almacenan en un solo lugar, facilitando la gestión, auditoría y análisis.
  - **Escalabilidad:** Elasticsearch permite almacenar y consultar grandes volúmenes de datos y escalar horizontalmente.
  - **Visualización avanzada:** Kibana proporciona potentes herramientas de visualización y análisis en tiempo real.
  - **Flexibilidad:** Fluentd puede adaptarse a múltiples fuentes y formatos de logs, integrándose fácilmente con diferentes aplicaciones y plataformas.
  - **Trazabilidad y troubleshooting:** Permite rastrear eventos y errores a través de múltiples servicios, facilitando la detección y resolución de problemas



ibertech

**end2end dedication**



Muchas gracias por vuestra asistencia



[www.ibertech.es](http://www.ibertech.es)