



Formación

Curso Spring con Microservicios



Formador

Ana Isabel Vegas



INGENIERA INFORMÁTICA con Master Máster Universitario en Gestión y Análisis de Grandes Volúmenes de Datos: Big Data, tiene la certificación PCEP en Lenguaje de Programación Python y la certificación JSE en JavaScript. Además de las certificaciones SCJP Sun Certified Programmer for the Java 2 Platform Standard Edition, SCWD Sun Certified Web Component Developer for J2EE 5, SCBCD Sun Certified Business Component Developer for J2EE 5, SCEA Sun Certified Enterprise Architect for J2EE 5.

Desarrolladora de Aplicaciones FULLSTACK, se dedica desde hace + de 20 años a la CONSULTORÍA y FORMACIÓN en tecnologías del área de DESARROLLO y PROGRAMACIÓN.



training@iconotc.com

Spring con Microservicios

□ **Duración:** 25 horas

□ **Modalidad:** On-line

□ **Fechas/Horario:**

- Días 2, 3, 4 y 5 de Diciembre de 2024
- Horario 9:30 – 14:00 y 15:00 – 17:30 hs. del 2 al 4 de Diciembre / 9:30 – 13:30 hs día 5 de Diciembre

□ **Contenidos:**

- Introducción a los Microservicios
- Spring Boot
- Acceso a Datos
- Consultas entre Servicios
- Monitorización de Servicios
- Spring Cloud

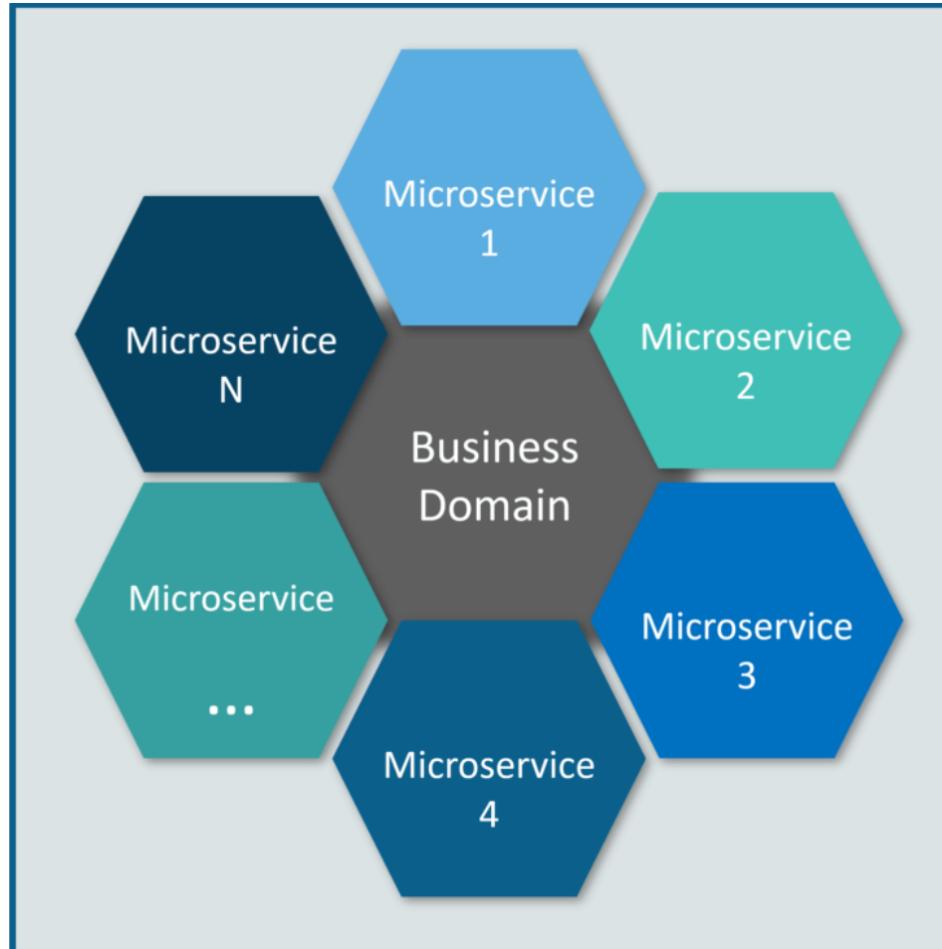
Introducción a los Microservicios

Tema 1

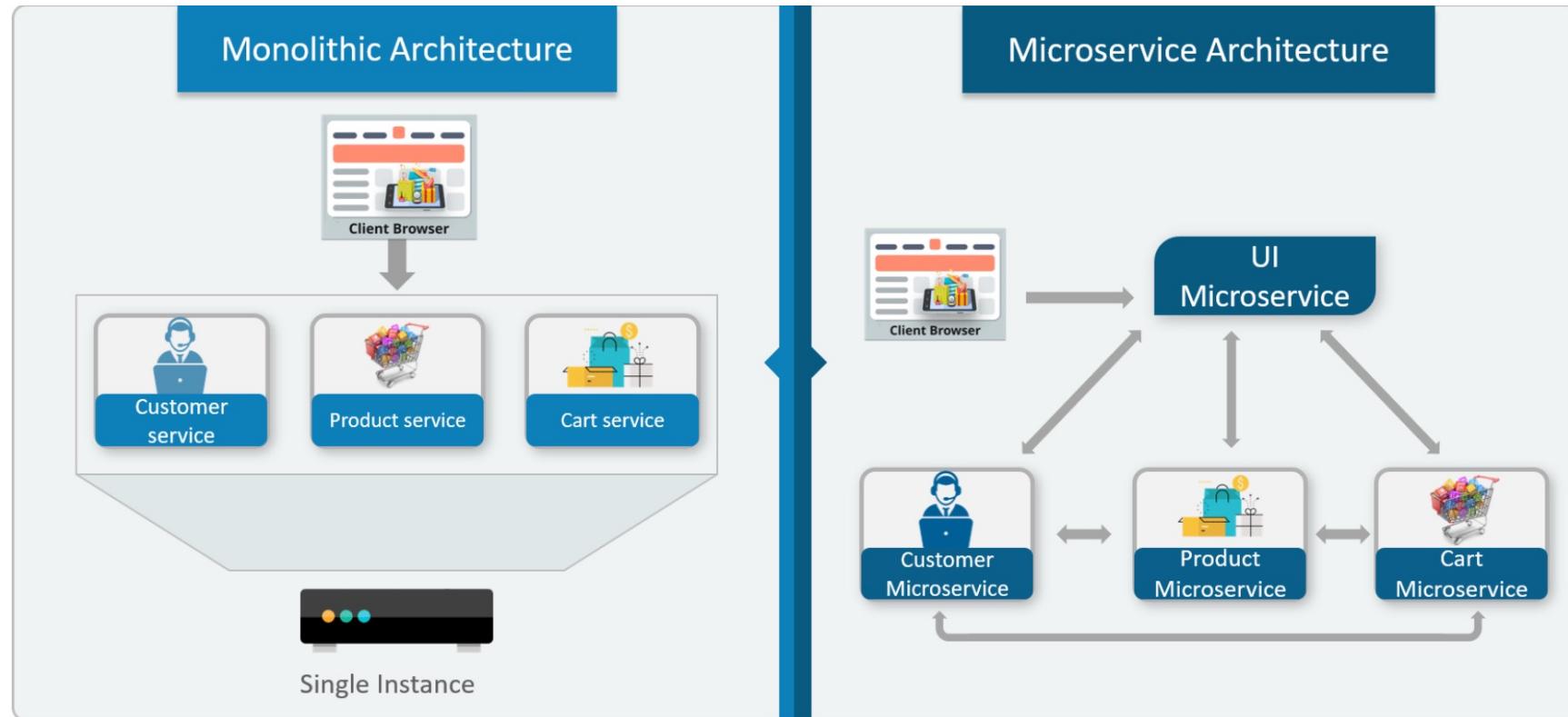
Qué es un microservicio?

- 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



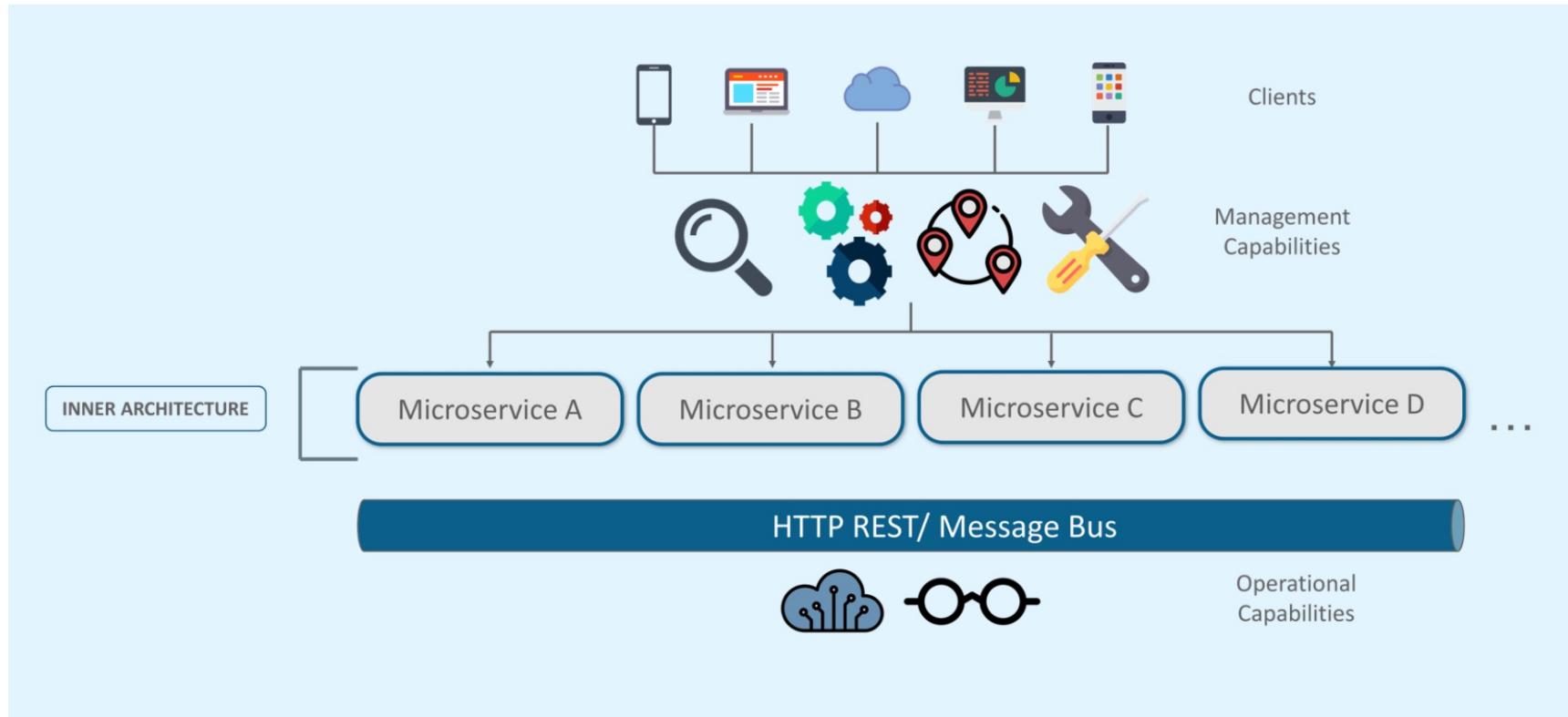
Tendencia en el desarrollo

- 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

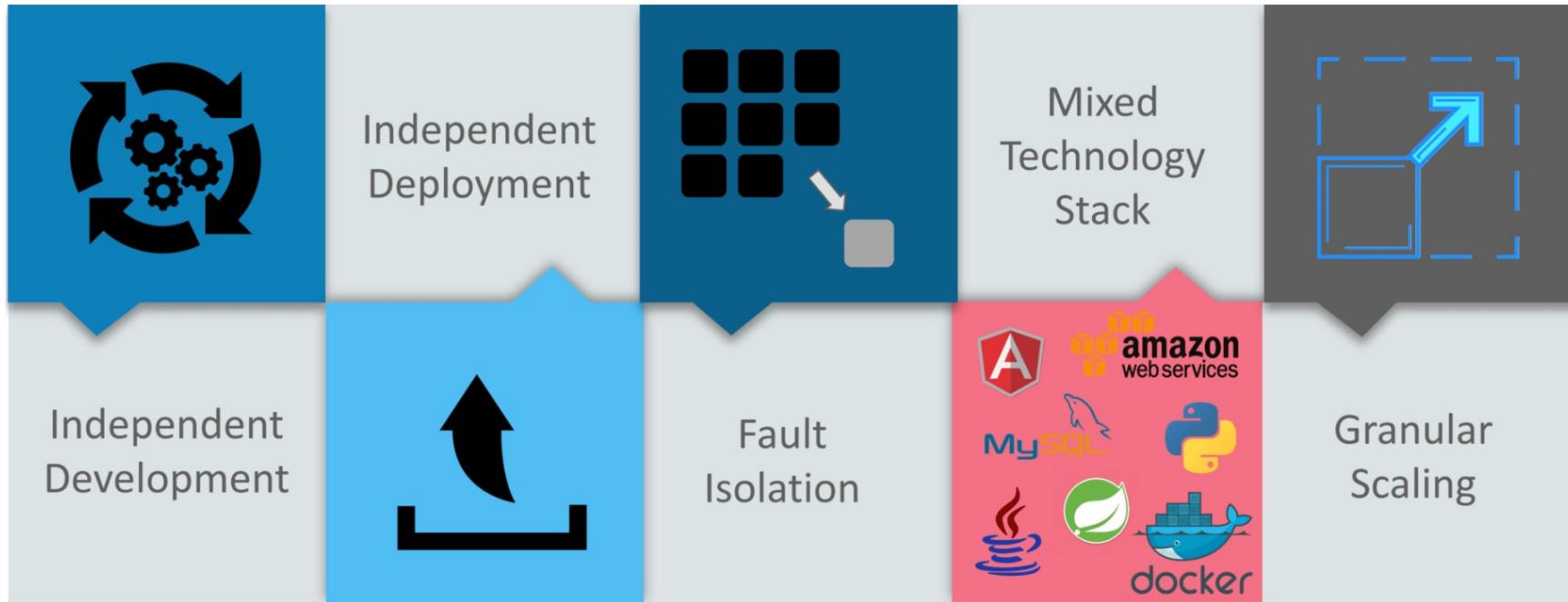
Características de los microservicios



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

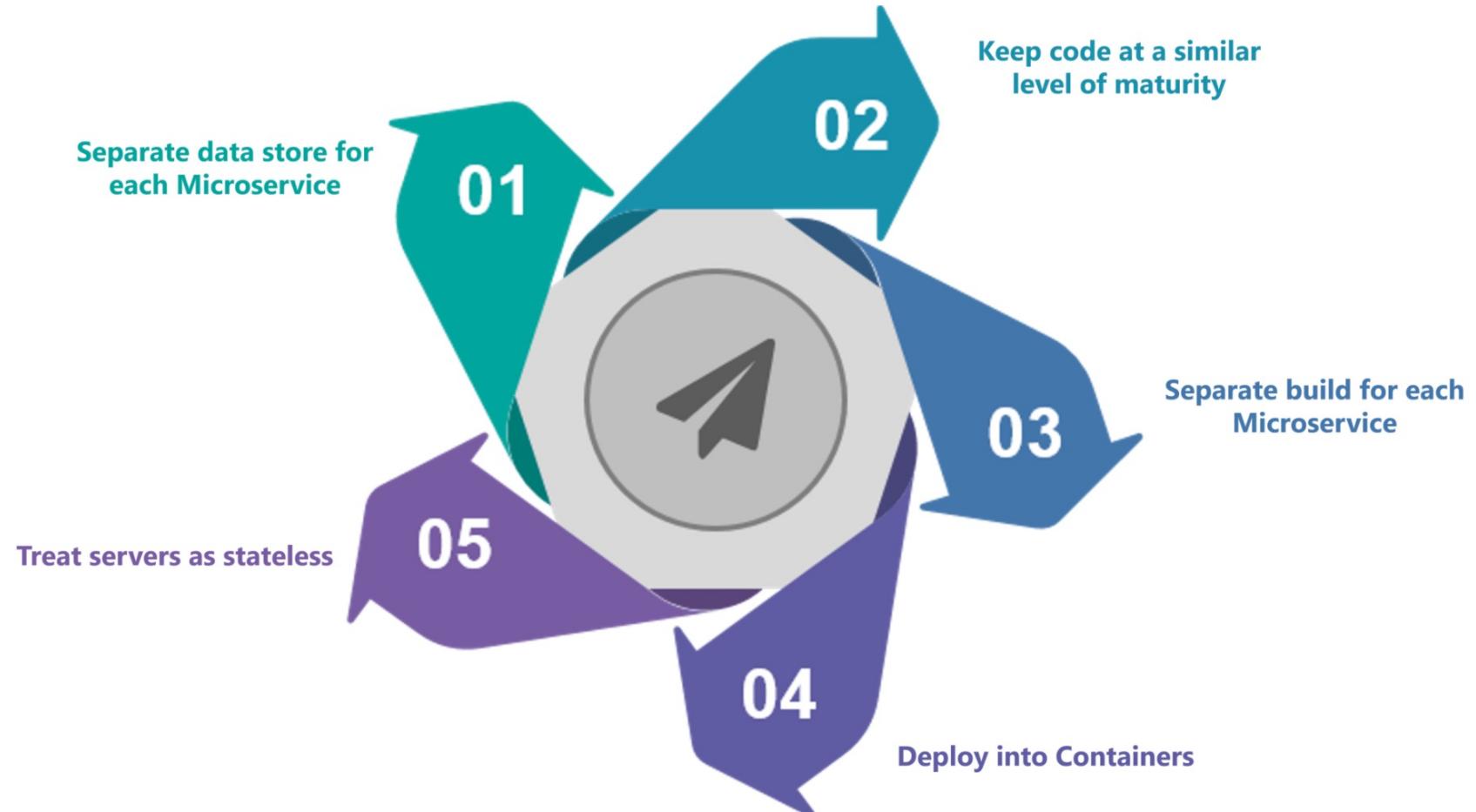
Ventajas de los microservicios



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?

amazon.com®



NETFLIX



GILT



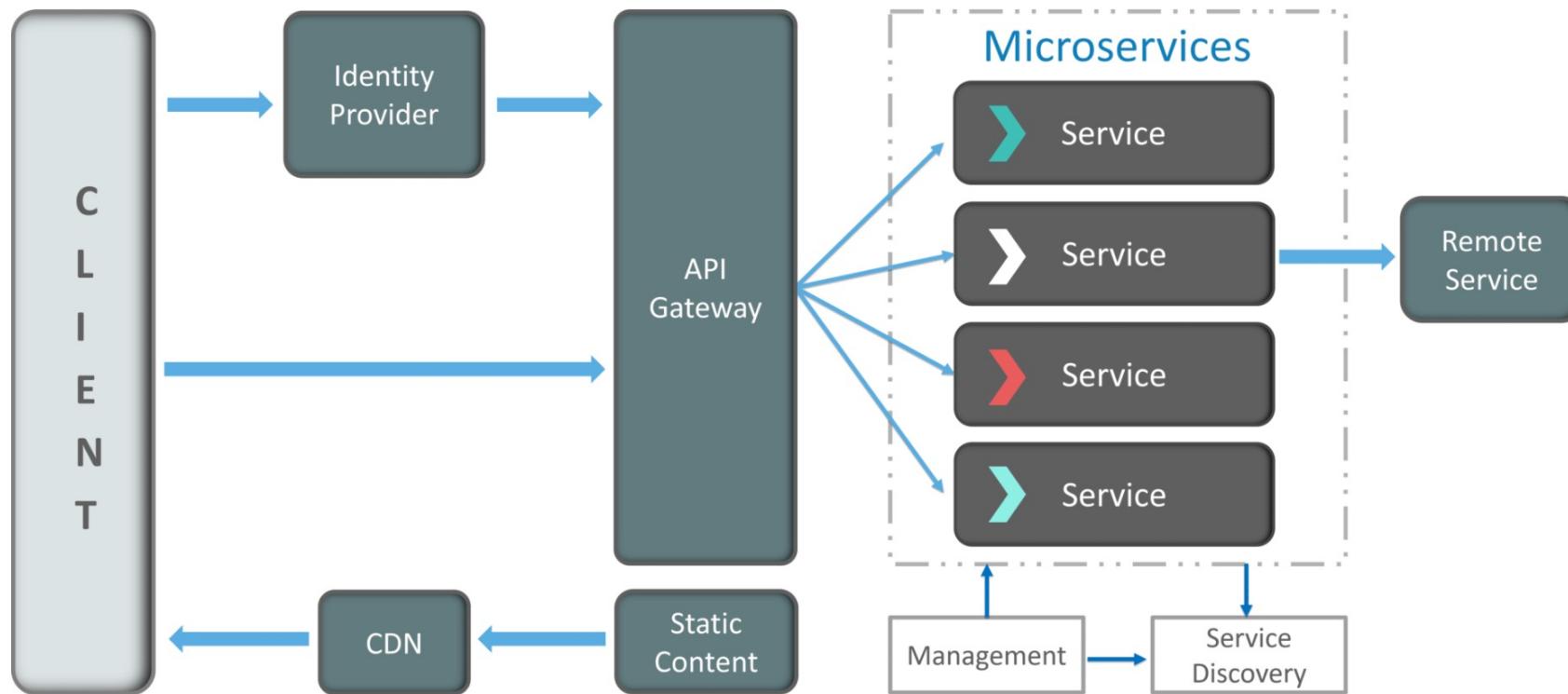
eBay



NORDSTROM

the guardian

Componentes de la arquitectura



Spring Boot

Tema 2

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 {

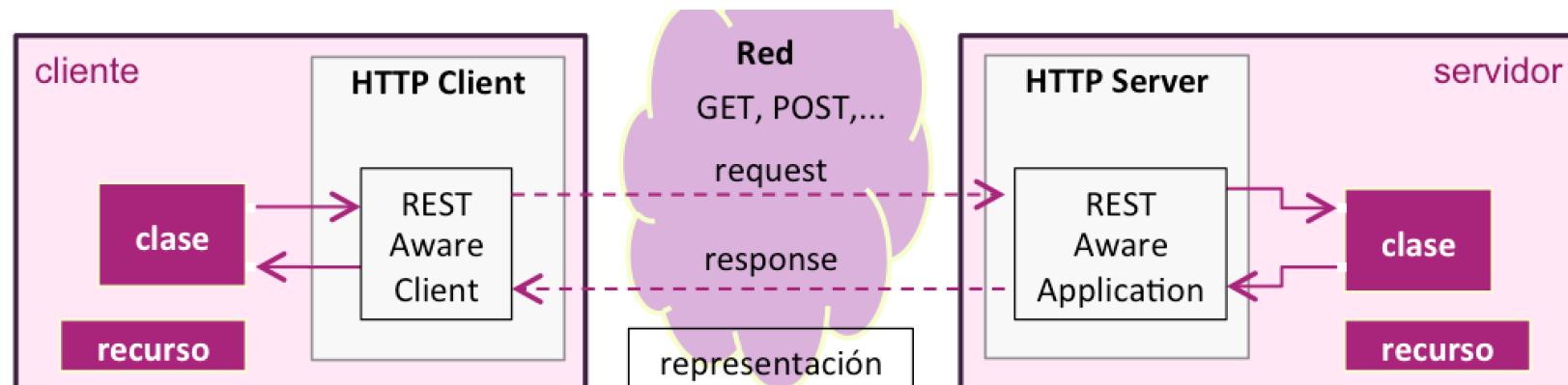
    public static void main(String[] args) {
        SpringApplication.run(HelloWorldApplication.class, args);
    }
}
```

Creando RESTfull Web Service

Tema 3

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

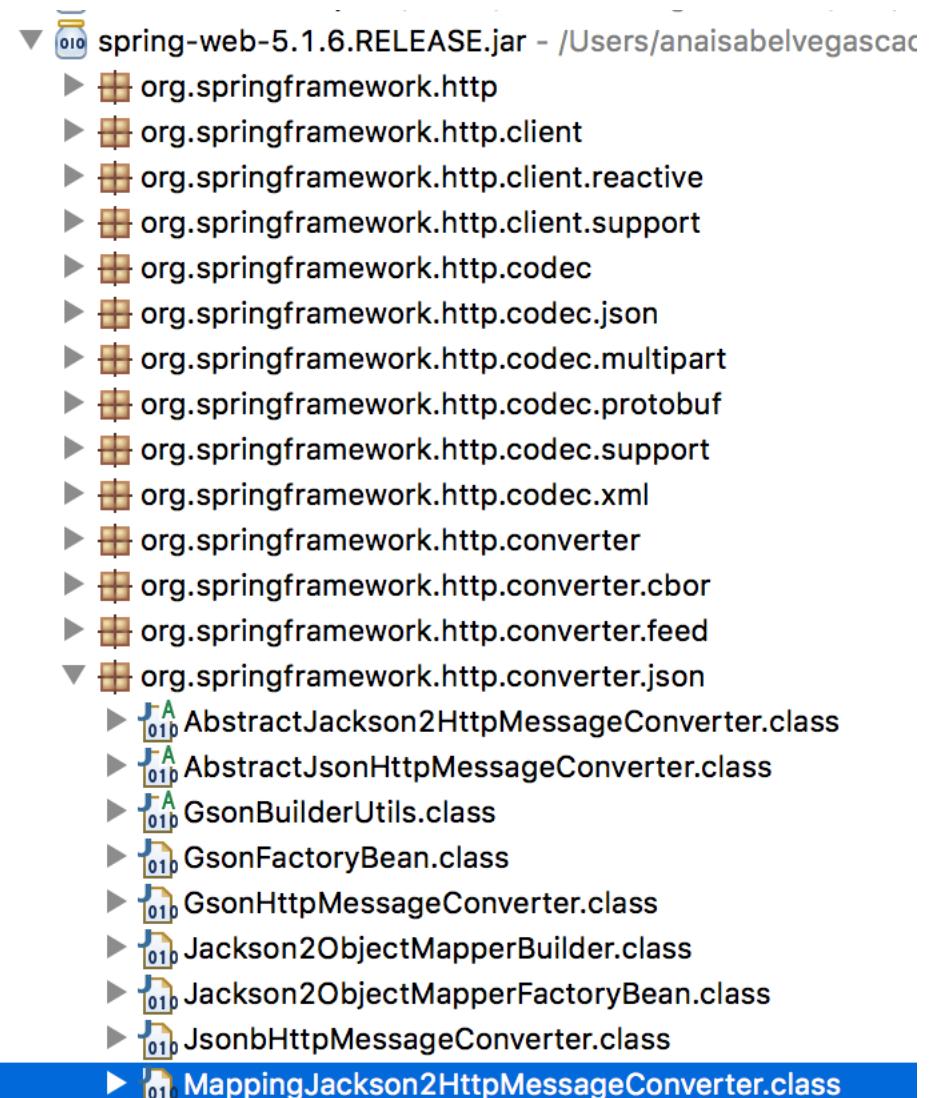
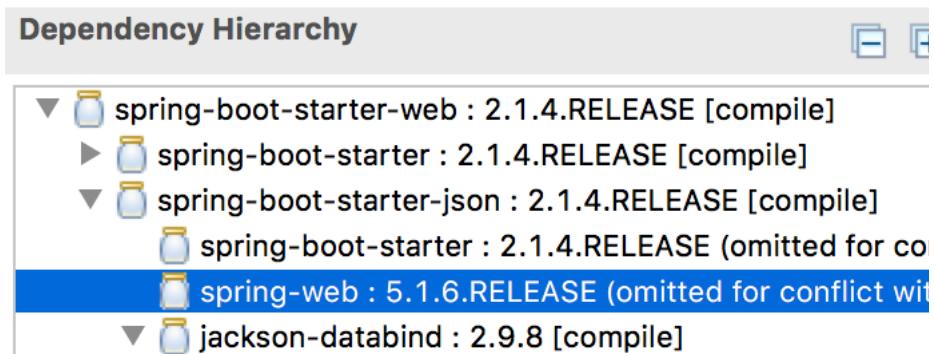
- Al agregar esta dependencia al pom.xml:

```
<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

Tema 4

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>
```

Acceso a datos con JPA

Tema 5

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

Tema 6

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

```
// Levantar el servidor de mongo
// comando mongod

// Desde la consola con permisos de administrador

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

//Alta de producto y consulta para su verificacion
// curl -i -X POST -H "Content-Type:application/json" -d '{"descripcion": "Enchufe", "precio":5.32 }' 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/5d13409d5161a02ec107c7a9
// curl http://localhost:8080/productos/search
// curl http://localhost:8080/productos/search/findByDescripcion?descripcion=Enchufe

//Modificar precio del producto y consulta para su verificacion
// curl -X PUT -H "Content-Type:application/json" -d '{"descripcion": "Enchufe", "precio":7.12 }' http://localhost:8080/productos/5d13409d5161a02ec107c7a9
// curl http://localhost:8080/productos/5d13409d5161a02ec107c7a9

//Borrar un producto y consulta para su verificacion
// curl -X DELETE http://localhost:8080/productos/5d13409d5161a02ec107c7a9
// curl http://localhost:8080/productos
```

Acceso a datos con MongoDB

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

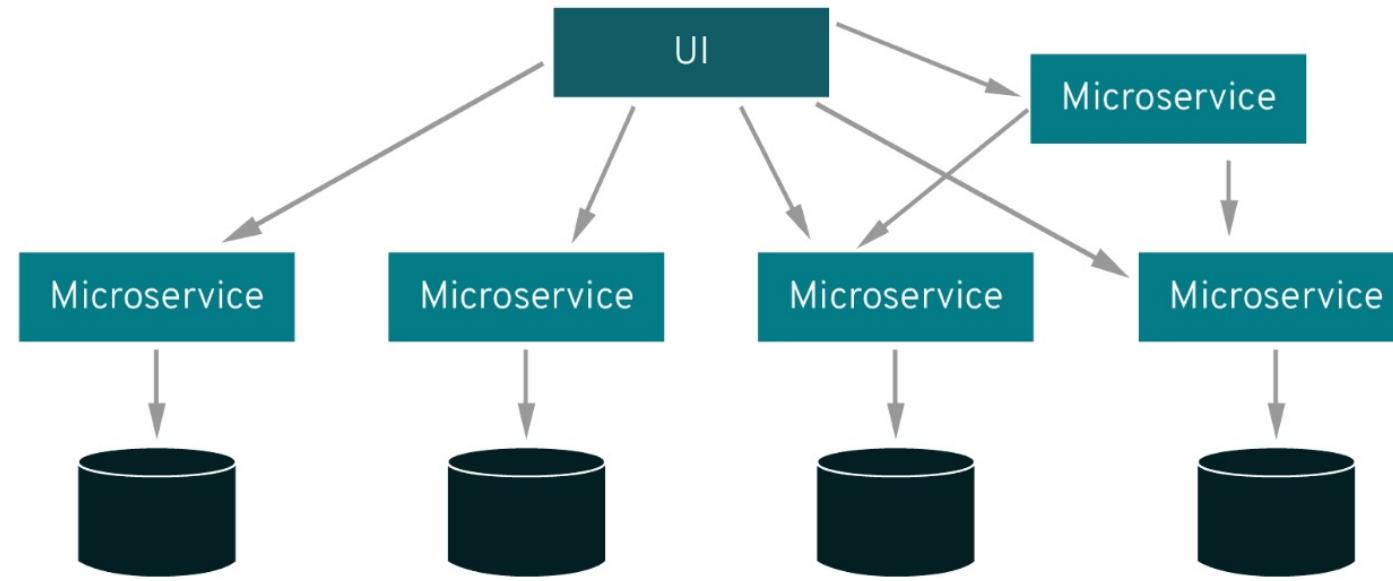
Acceso a datos con MongoDB

```
|  
@SpringBootApplication  
public class Application implements CommandLineRunner {  
  
    @Autowired  
    private ProductoRepository repository;  
  
    public static void main(String[] args) {  
        SpringApplication.run(Application.class, args);  
    }  
  
    @Override  
    public void run(String... args) throws Exception {  
  
        repository.deleteAll();  
  
        // alta de productos  
        repository.save(new Producto("Alargador", 18.34));  
        repository.save(new Producto("Bombilla Led", 5.23));  
  
        // listar todos los productos  
        System.out.println("Todos los productos encontrados");  
        System.out.println("-----");  
        for (Producto producto : repository.findAll()) {  
            System.out.println(producto);  
        }  
        System.out.println();  
  
        // Buscar un producto por su descripcion  
        System.out.println("Buscando Bombillas Led");  
        System.out.println("-----");  
        System.out.println(repository.findByDescripcion("Bombilla Led"));  
    }  
}
```

Consultas entre servicios

Tema 7

Consulta entre servicios



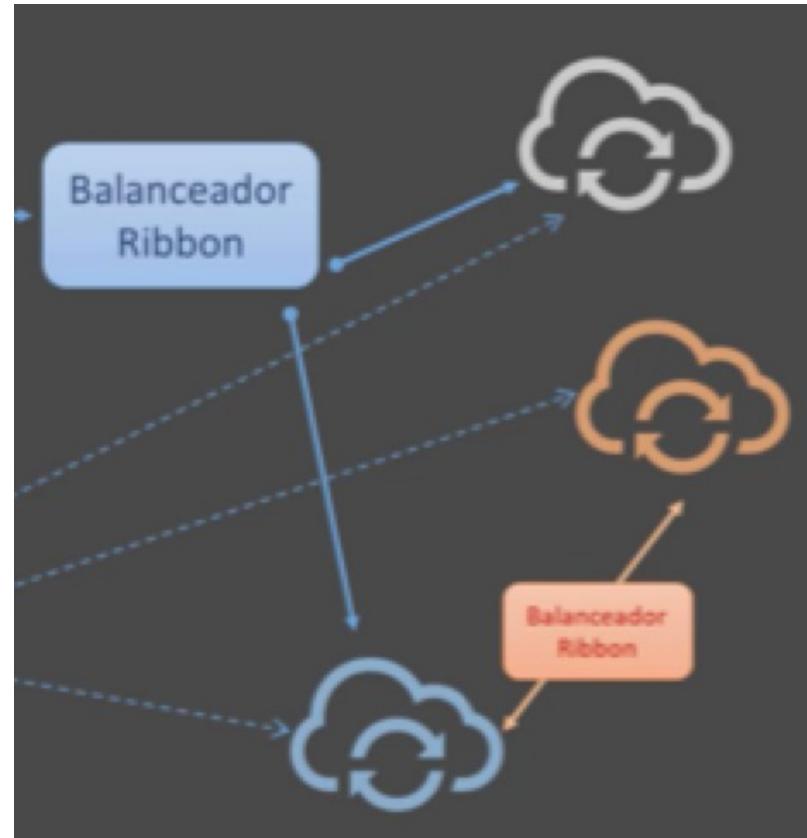
Consulta entre servicios

- 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

Tema 8

Ribbon



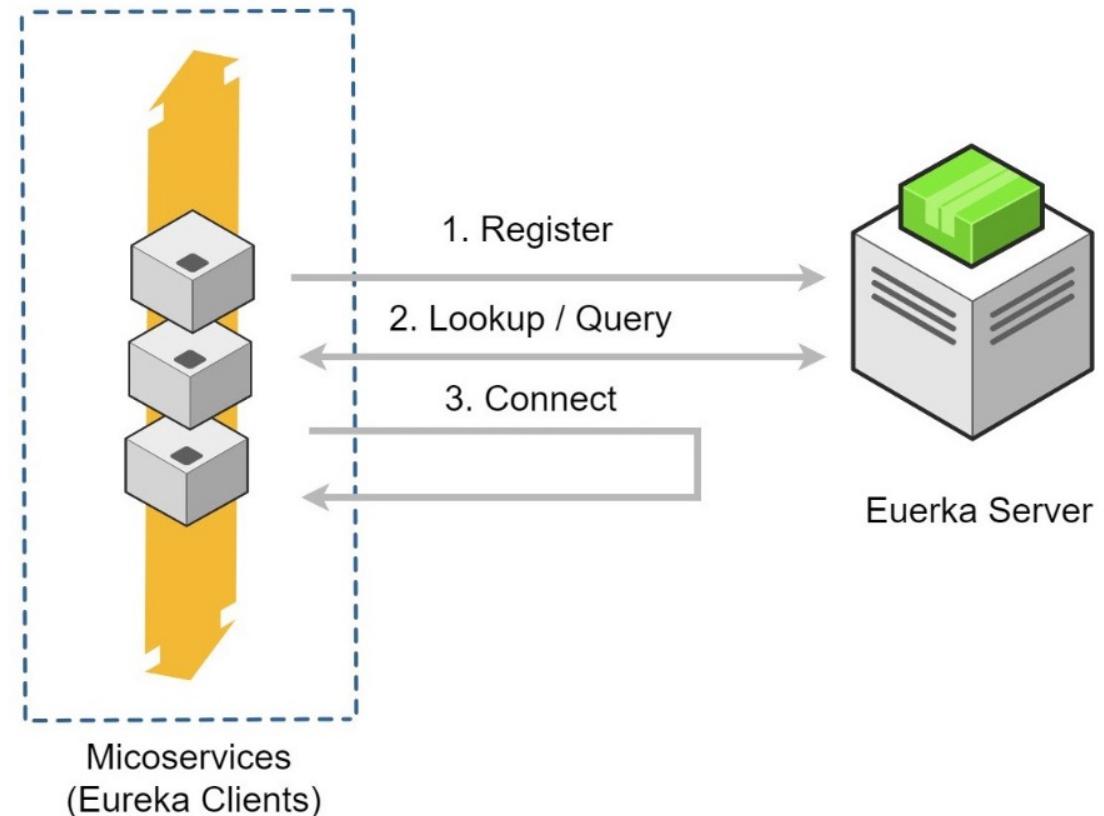
Ribbon

- 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

Tema 9

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

Circuit Breakers y Tolerancia a Fallos

Tema 10

Hystrix

- Es una librería de Spring Cloud para manejar errores y evitar la propagación a otros microservicios
- Dependencia necesaria

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
    <version>2.2.2.RELEASE</version>
</dependency>
```

- En caso de ocurrir un error se invoca al método manejarError

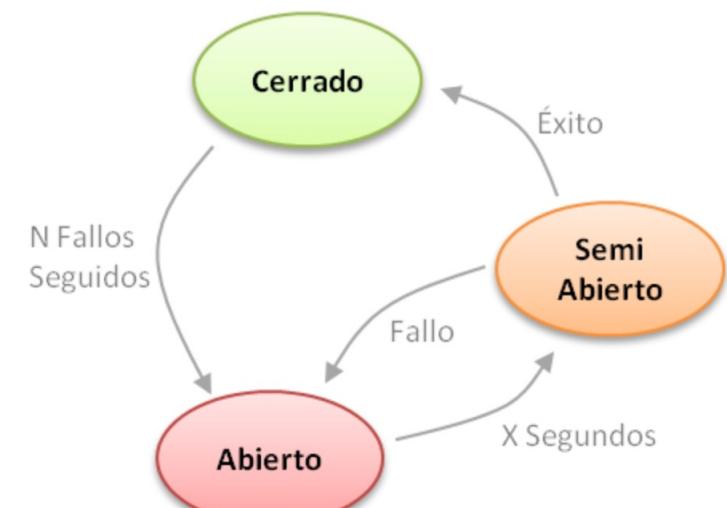
```
@HystrixCommand(fallbackMethod = "manejarError")
```

Resilience4J

- Resilience4J viene a sustituir a Hystrix.
- Dependencia necesaria

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-circuitbreaker-resilience4j</artifactId>
</dependency>
```

- Estados de Circuit Breaker:
 - **Cerrado**: Es el estado inicial. Cuando no hay errores.
 - **Abierto**. Cuando se ha superado la tasa de fallos se cambia a estado abierto. En este estado, las peticiones no se reciben. Transcurrido un cierto tiempo se cambiará a semiabierto.
 - **Semiabierto**. Se ejecutarán varias peticiones para ver si el microservicio funciona bien o no. Si todo funciona correctamente vuelve a estado cerrado, si sigue habiendo errores volverá a estado abierto.



- Parámetros de Circuit Breaker:
 - **slidingWindowSize(100)**; Es una muestra estadística, se lanzan 100 peticiones para saber si el estado debe ser abierto o cerrado
 - **failureRateThreshold(50%)**; Es el porcentaje de errores, si de 100 peticiones, fallan 50, el estado cambia a abierto
 - **waitDurationInOpenState(60000ms)**; Es el tiempo que esta esperando en modo abierto
 - **permittedNumberOfCallsInHalfOpenState(10)**; Numero de peticiones permitidas en estado semi abierto.
 - **slowCallDurationThreshold(60000ms)**; Si la petición tarda más de 60 segundas se considera conexión lenta.
 - **slowCallRateThreshold(50%)**; Es el umbral para las peticiones lentas. Si el 50% de las peticiones son lentas, el estado cambia a abierto.

API Gateway

Tema 11

Zuul

- Con Zuul creamos la puerta de enlace con las rutas dinámicas para acceder a los microservicios.
- Dependencia:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-zuul</artifactId>
</dependency>
```

- Rutas dinámicas:

```
zuul.routes.productos.service-id=servicio-productos
zuul.routes.productos.path=/api/productos/**
```

Spring Cloud Gateway

- Spring Cloud Gateway es una alternativa a Zuul Netflix.
- Spring Cloud Gateway trabaja con Spring WebFlux, mientras que Zuul trabaja con API Servlet.
- La recomendación es crear la puerta de enlace con Spring Cloud Gateway porque además de estar preparado para trabajar con programación reactiva, forma parte de Spring Cloud que ahora es lo mas recomendable ya que Zuul Netflix esta en modo mantenimiento y no habrá nuevas versiones.
- Dependencia necesaria:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
```

Configuración distribuida

Tema 12

Spring Cloud Config Server

- Cada microservicio puede mantener su propia configuración o bien centralizarlas en otro microservicio. Para esto utilizamos Spring Cloud Config Server.
- El funcionamiento es que cada microservicio consulta su configuración al servidor config server y tras obtenerla ya se puede registrar en Eureka Server y estará listo para funcionar.
- La dependencia necesaria:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

Spring Cloud Config Server

- Utilizando git podemos crear un repositorio local donde recoger todas las configuraciones
- También podemos utilizar repositorios remotos con GitHub.
- Podremos tener diferentes ambientes como por ejemplo: desarrollo y producción

DOCUMENTACION

- Herramientas utilizadas en el curso:
 - JDK 8: <https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>
 - Eclipse: <https://www.eclipse.org/downloads>
 - Tomcat 8: <https://tomcat.apache.org/download-80.cgi>
 - Maven: <https://maven.apache.org/download.cgi>
- Sitio oficial de maven: <https://maven.apache.org>
- Repositorio de maven: <https://mvnrepository.com>

Spring con Microservicios



GRACIAS

Indra

Completa nuestra encuesta
de satisfacción a través del QR

