

Servicios Web

Profesora:
Ana Isabel Vegas

Detalles del curso

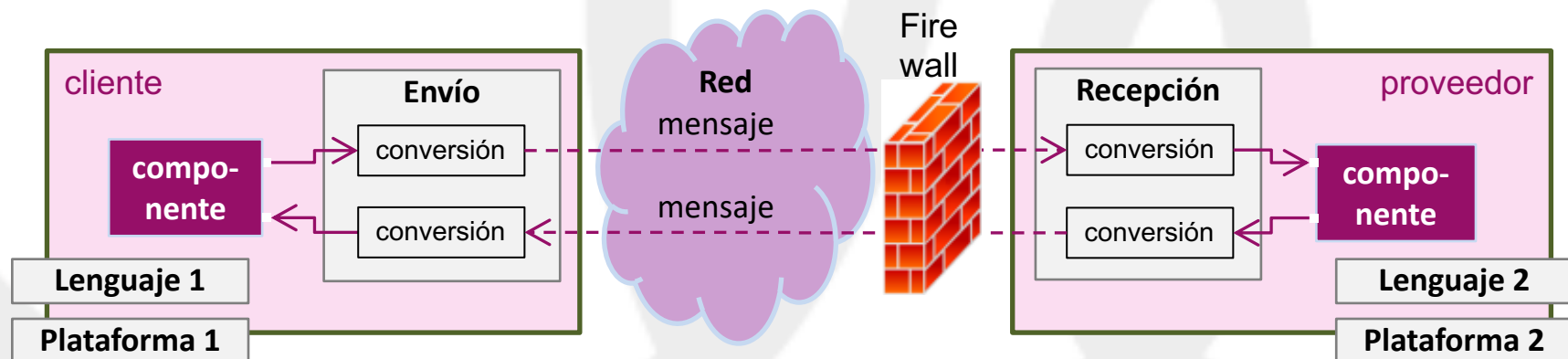
- Duración: 30 horas.
- Temario:
 - Arquitectura de servicios Web
 - Aproximaciones para creación de servicios Web
 - Introducción a los servicios Web XML. Características e inconvenientes
 - Servicios Web REST
 - Implementación de servicios REST con JAX-WS
 - Implementación de servicios REST con Spring MVC
 - Interacción entre servicios REST
 - Acceso a un servicio REST desde el front
 - Microservicios vs servicios
 - Implementación de microservicios con Spring Boot
 - Componentes cloud en una arquitectura de microservicios
 - Desarrollo de clientes.

Arquitectura de Servicios Web

Introducción

Operaciones entre plataformas a través de la red

- Problema a resolver: Integración que consiste en que un proveedor ofrece un servicio que es útil para un cliente, con las siguientes características:
 - Entornos físicamente separados.
 - Posiblemente son distintos lenguajes de programación (Java, C#, ...)
 - Posiblemente distintas plataformas (Windows, Linux, ...)
 - Invocación a través de la red, pasando eventualmente por firewalls.

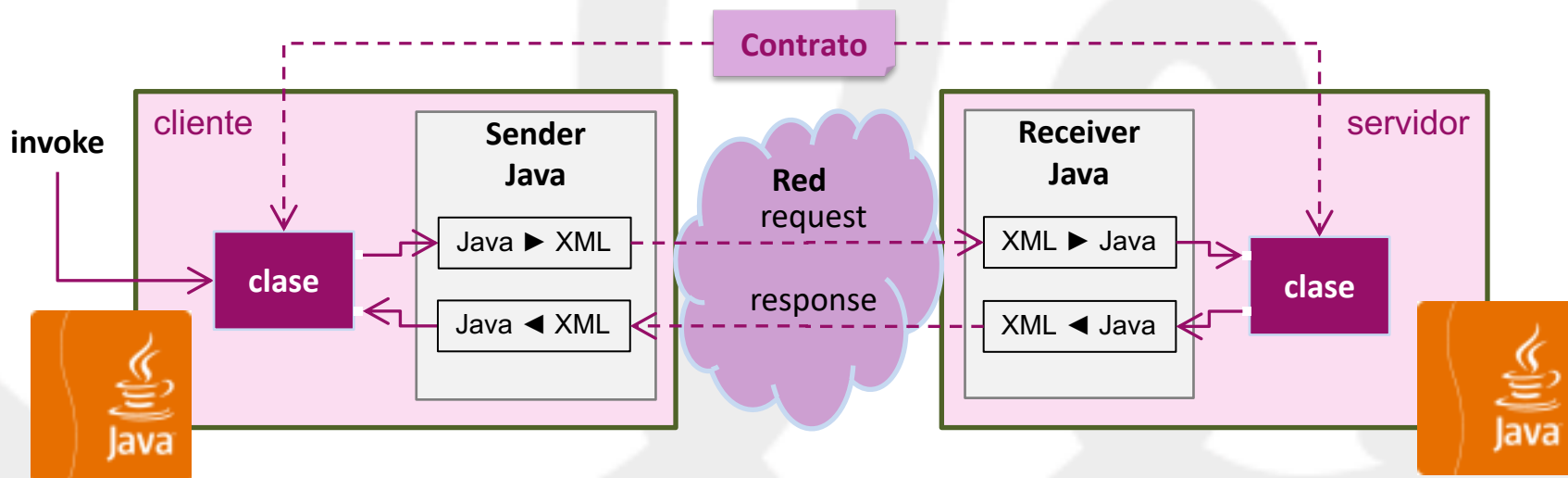


Funcionamiento básico de un Web Service

- Web Service como solución al problema:
- Servicio que se publica en un servidor, que es capaz de recibir un mensaje, interpretarlo, procesarlo y generar una respuesta, la cual se retorna en otro mensaje. Las funcionalidades que expone se llaman operaciones.
- Un cliente invoca a través de una URL:
 - Envía un mensaje de requerimiento (request) que contiene la operación y los parámetros.
 - Recibe un mensaje de respuesta (response) con el resultado.
- Para resolver el problema de las diferencias de plataformas, la invocación y la respuesta utilizan un formato estándar, normalmente XML.
- Tanto el lado que invoca (cliente) como el que provee (servidor) hacen la conversión desde y hacia XML (u otro utilizado), permitiendo que la interacción sea en una forma única.

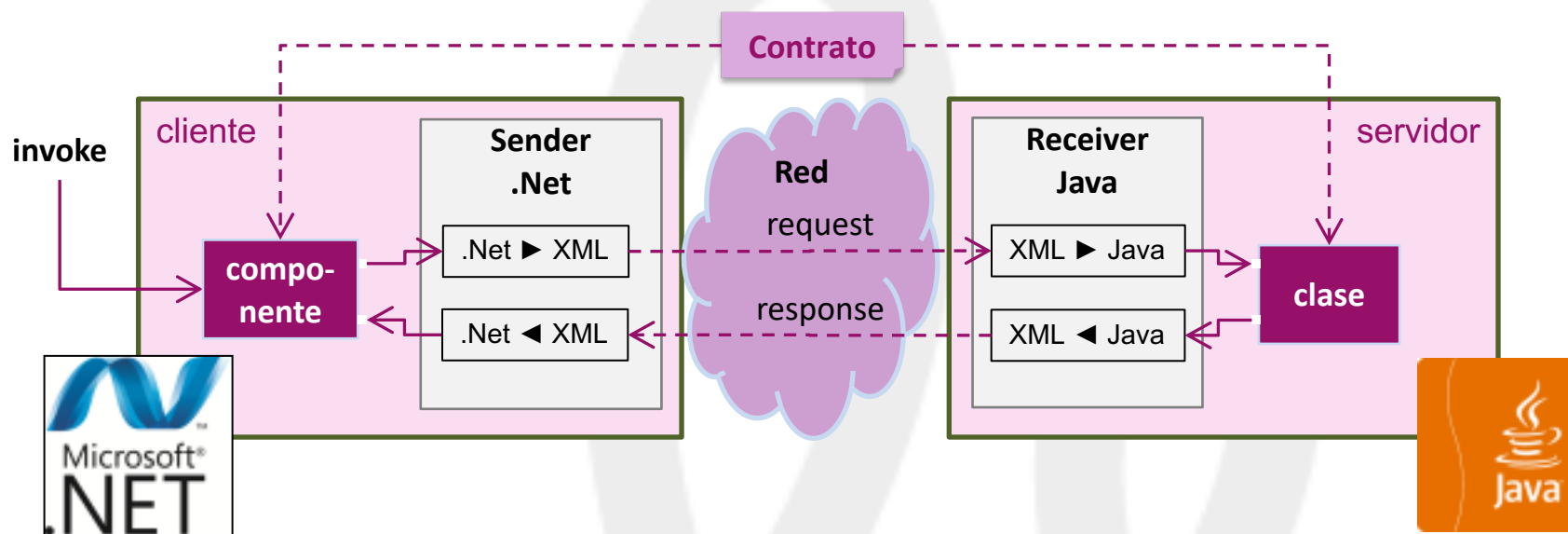
Funcionamiento básico de un Web Service

- Características de la solución:
 - Cliente y servidor están desacoplados completamente.
 - La especificación de la interacción entre ambos, se define a través de un "contrato", que sigue un estándar en formato XML, y se llama WSDL.
 - Caso con cliente Java y servidor Java:



Funcionamiento básico de un Web Service

- Caso con cliente en .Net y servidor en Java:



También se podría tener:

- Servidor en .NET, PHP, CGI, y cualquiera que pueda manejar peticiones HTTP.
- Cliente en JavaScript, PHP, y cualquiera que pueda generar peticiones HTTP.

Ventajas de los Web Service

- Proporcionan interoperabilidad entre aplicaciones de software independientes y/o entre las plataformas sobre las que se instalan, ya que son independientes del lenguaje de programación.
- Utilizan estándares y protocolos definidos.
- Al basarse en protocolos de transporte como HTTP, los Web Services pueden atravesar firewall sin necesidad de cambiar las reglas de filtrado.
- Permiten que servicios y software con diferentes orígenes puedan ser integrados.
- Las especificaciones son gestionadas por una organización independiente, la W3C.
- En conclusión:
 - Permiten la interoperabilidad entre plataformas de distintos fabricantes por medio de protocolos estándares y abiertos.

Servicios SOAP con JAX-WS

Parte 1

JAX-WS

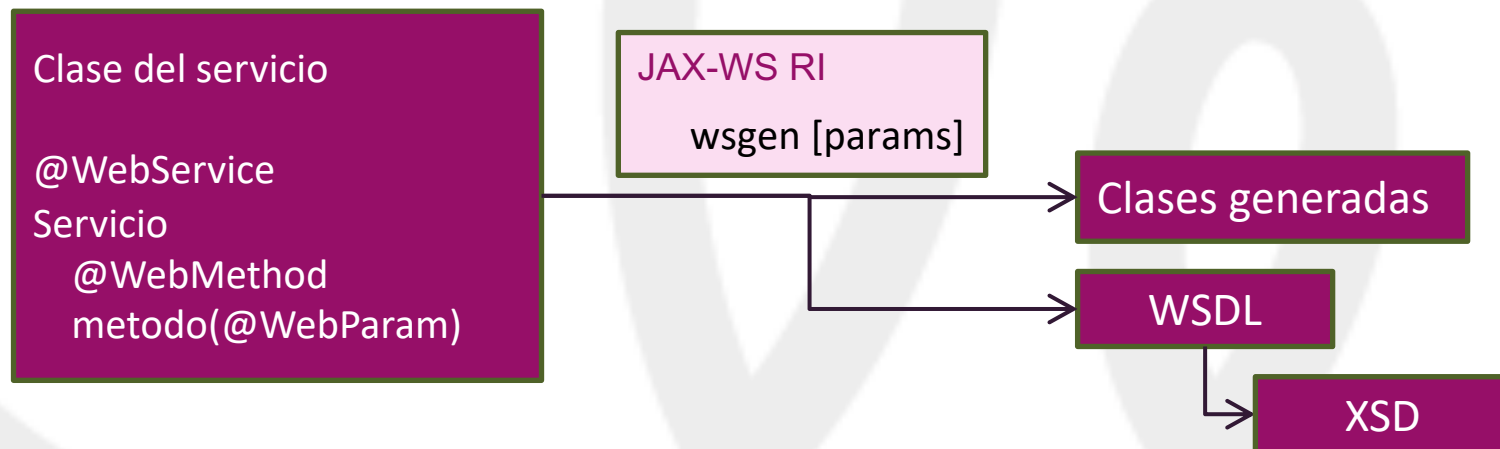
- JAX-WS (Java API for XML Web Services):
- Es un estándar Java EE, presente desde la versión 5.
- Es decir, JAX-WS no es un framework como tal. Cada servidor de aplicaciones lo implementa, y también existe una "Reference Implementation", llamada JAX-WS RI, como parte del proyecto "GlassFish Metro".
- Utiliza anotaciones de Java 5 para facilitar el desarrollo y despliegue, tanto del lado servidor como del lado cliente.
- JAX-WS RI contiene tanto las librerías como las herramientas de generación de código.

JAX-WS

- Para publicar un Web Service con JAX-WS RI, se requiere:
- Tener una aplicación Web.
- Agregar a la aplicación las librerías de JAX-WS RI, que incluyen la API, la implementación de referencia, la API e implementación de JAXB, y el compilador XJC, entre otras.
- Declarar en el web.xml de la aplicación el Listener de JAX-WS, que lo inicializa, y el Servlet genérico de JAX-WS, que maneja la configuración y los mensajes SOAP de request y response.

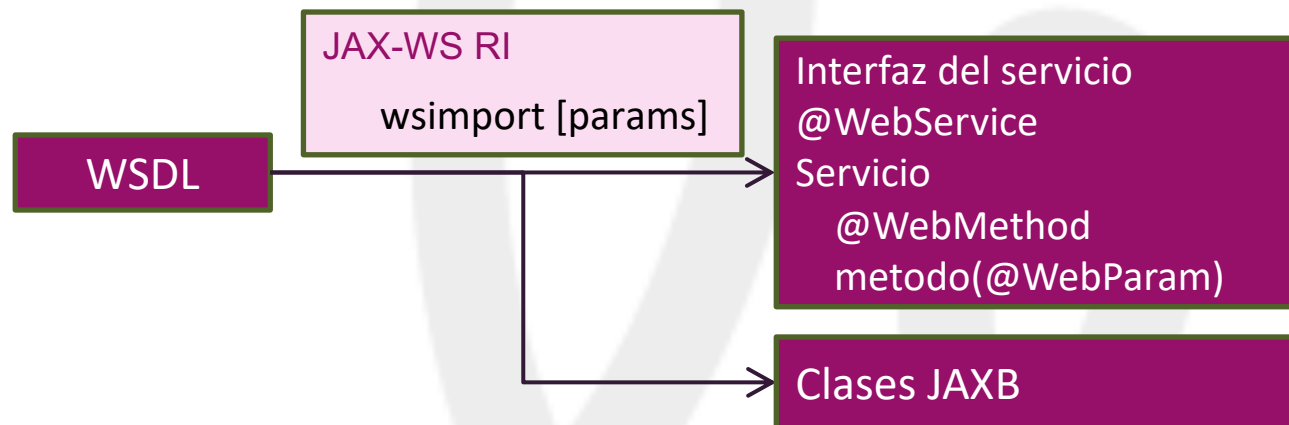
JAX-WS

- Si se utiliza Bottom Up:
- Implementar la clase que se expone como Web Service, y agregarle las anotaciones `@WebService`, `@WebMethod` y `@WebParam`.
- Utilizar el comando `wsgen` que viene con JAX-WS RI para generar las clases del servicio y el WSDL.



JAX-WS

- Si se utiliza Top Down:
- Implementar un archivo WSDL con la definición del servicio.
- Generar la interfaz java del servicio y las clases JAXB asociadas, utilizando el comando `wsimport` que viene con JAX-WS RI.



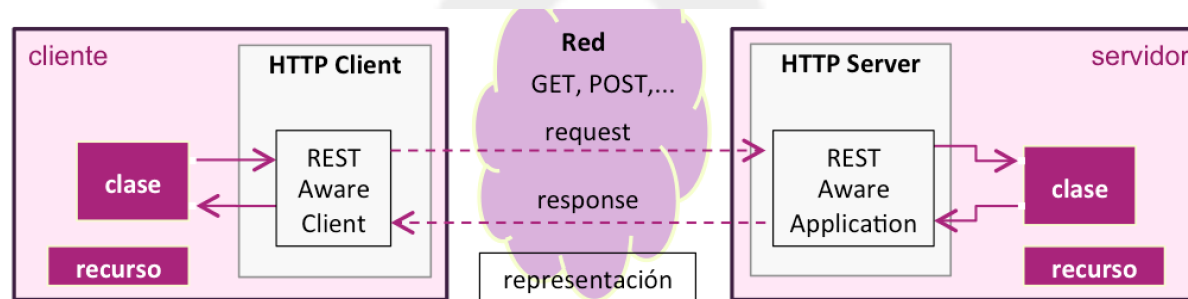
- Implementar la interfaz del servicio generada.

Servicios REST con JAX-RS

Parte 2

JAX-RS

- **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.

JAX-RS

- 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

JAX-RS

- 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, pero no contiene un body.

JAX-RS

- **WADL** (Web Application Description Language):
- Es el descriptor de servicio de aplicaciones basadas en HTTP, y en particular REST
- Equivalente al WSDL utilizado en SOAP.
- No es un estándar y por el momento no hay planes para hacerlo.
- Estructura:
 - application
 - grammars: incluye los XSD.
 - resources: define la URI
 - resource: define la operación
 - method: puede ser GET, POST, ...
 - request: parámetros en el envío.
 - response: incluye representaciones.

Comparación con SOAP

- Ventajas con respecto a SOAP:
 - El mensaje se construye directamente con HTTP, lo que da más simplicidad y rendimiento.
 - Los mensajes normalmente son más pequeños.
 - Permite definir interacciones con estado en forma nativa, a través del uso de la sesión HTTP.
 - Se puede invocar en forma sencilla desde JavaScript, lo que facilita su uso en páginas dinámicas con Ajax.
- Desventajas con respecto a SOAP:
 - SOAP es un protocolo estándar abierto.
 - SOAP facilita más la interoperabilidad entre sistemas, al ser estándar.
 - SOAP facilita más la depuración.

Spring MVC

Parte 3

Spring MVC

- Spring MVC es una alternativa de framework basado en el patrón modelo-vista-controlador. Es un modulo más dentro del framework Spring.
- A partir de la versión 3 permite trabajar con anotaciones que suplen al código xml.

Controladores basados en anotaciones

```
@Controller
//@RequestMapping("/")
@RequestMapping({"/", "/home"})
public class InicioController {

    @RequestMapping(method = RequestMethod.GET, value = "")
    public String inicio() {
        return "index";
    }
}
```

Configuración vía anotaciones

```
@Configuration
@EnableWebMvc
@ComponentScan("com.example")
public class JavaConfig {

    @Bean
    public ViewResolver internalResourceViewResolver() {
        InternalResourceViewResolver view = new InternalResourceViewResolver();
        view.setViewClass(JstlView.class);
        view.setPrefix("/WEB-INF/JSP/");
        view.setSuffix(".jsp");
        return view;
    }

    @Bean
    public BeanNameViewResolver beanNameViewResolver() {
        return new BeanNameViewResolver();
    }
}
```

Activar páginas JSP

```
<!-- Necesitamos habilitar JSP -->  
<dependency>  
  <groupId>org.apache.tomcat.embed</groupId>  
  <artifactId>tomcat-embed-jasper</artifactId>  
</dependency>  
  
<dependency>  
  <groupId>javax.servlet</groupId>  
  <artifactId>jstl</artifactId>  
</dependency>
```


Restful Web Services con Spring MVC

Parte 4

Building a RESTful 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;
    }
}
```

Building a RESTful 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.

Building a RESTful Web Service

- Formateando la respuesta

Dependency Hierarchy

- ▼  spring-boot-starter-web : 2.1.4.RELEASE [compile]
 - ▶  spring-boot-starter : 2.1.4.RELEASE [compile]
 - ▼  spring-boot-starter-json : 2.1.4.RELEASE [compile]
 -  spring-boot-starter : 2.1.4.RELEASE (omitted for conflict with spring-web)
 -  spring-web : 5.1.6.RELEASE (omitted for conflict with spring-web)
 - ▼  jackson-databind : 2.9.8 [compile]

- ▼  spring-web-5.1.6.RELEASE.jar - /Users/anaisabelvegascac
 - ▶  org.springframework.http
 - ▶  org.springframework.http.client
 - ▶  org.springframework.http.client.reactive
 - ▶  org.springframework.http.client.support
 - ▶  org.springframework.http.codec
 - ▶  org.springframework.http.codec.json
 - ▶  org.springframework.http.codec.multipart
 - ▶  org.springframework.http.codec.protobuf
 - ▶  org.springframework.http.codec.support
 - ▶  org.springframework.http.codec.xml
 - ▶  org.springframework.http.converter
 - ▶  org.springframework.http.converter.cbor
 - ▶  org.springframework.http.converter.feed
 - ▼  org.springframework.http.converter.json
 - ▶  AbstractJackson2HttpMessageConverter.class
 - ▶  AbstractJsonHttpMessageConverter.class
 - ▶  GsonBuilderUtils.class
 - ▶  GsonFactoryBean.class
 - ▶  GsonHttpMessageConverter.class
 - ▶  Jackson2ObjectMapperBuilder.class
 - ▶  Jackson2ObjectMapperFactoryBean.class
 - ▶  JsonbHttpMessageConverter.class
 - ▶  MappingJackson2HttpMessageConverter.class

Consuming a RESTful Web Service

Tema 5

Consuming a RESTful 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();  
}
```

Consuming a RESTful 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 Boot

Parte 6

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, inyección, etc.

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**, que engloba al resto.

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);
    }
}
```

Spring Boot con bases de datos

Parte 7

Accessing JPA Data with REST

- 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.

Accessing JPA Data with REST

- 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>
```

Accessing JPA Data with REST

- 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>
```


Accessing JPA Data with REST

- 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;
```

Accessing JPA Data with REST

- 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);

}
```

Accessing MongoDB Data with REST

- 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 base 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"
  }
}
```

Accessing MongoDB Data with REST

- 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.

Accessing MongoDB Data with REST

- 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>
```

Accessing MongoDB Data with REST

```
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;  
    }  
}
```

Accessing MongoDB Data with REST

- Producto repository:

```
@RepositoryRestResource(collectionResourceRel = "productos", path = "productos")  
public interface ProductoRepository extends MongoRepository<Producto, String> {  
  
    List<Producto> findByLastName(@Param("descripcion") String descripcion);  
  
}
```

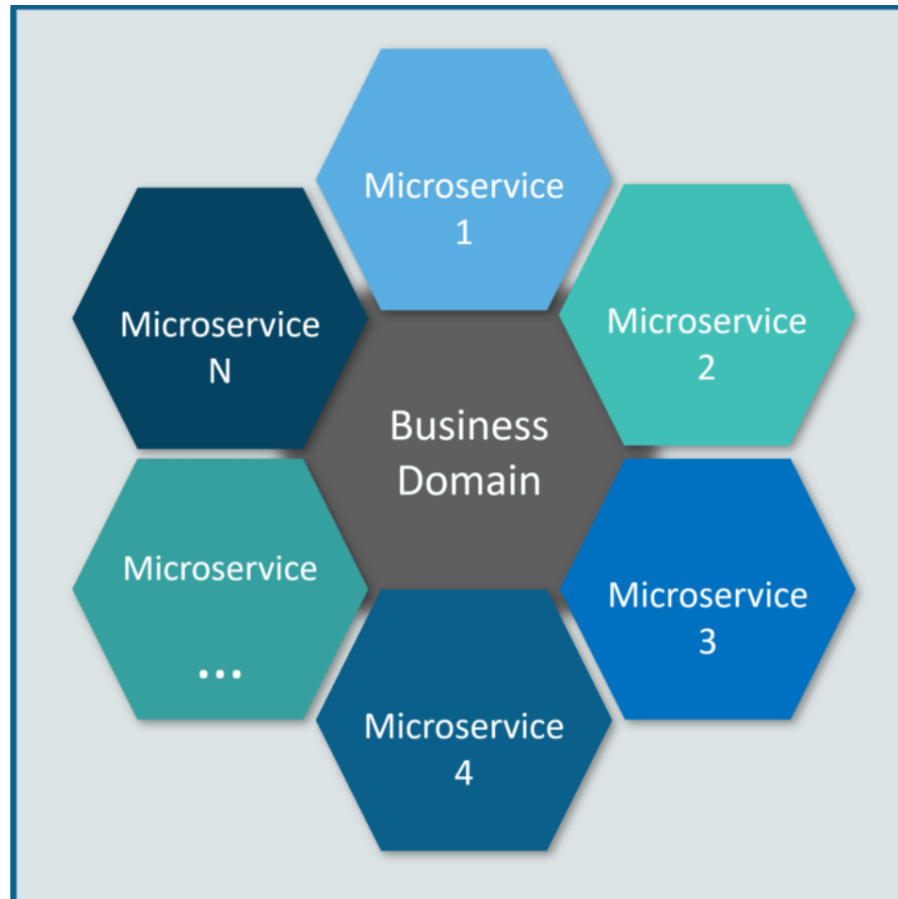
Microservicios con Spring Cloud

Parte 8

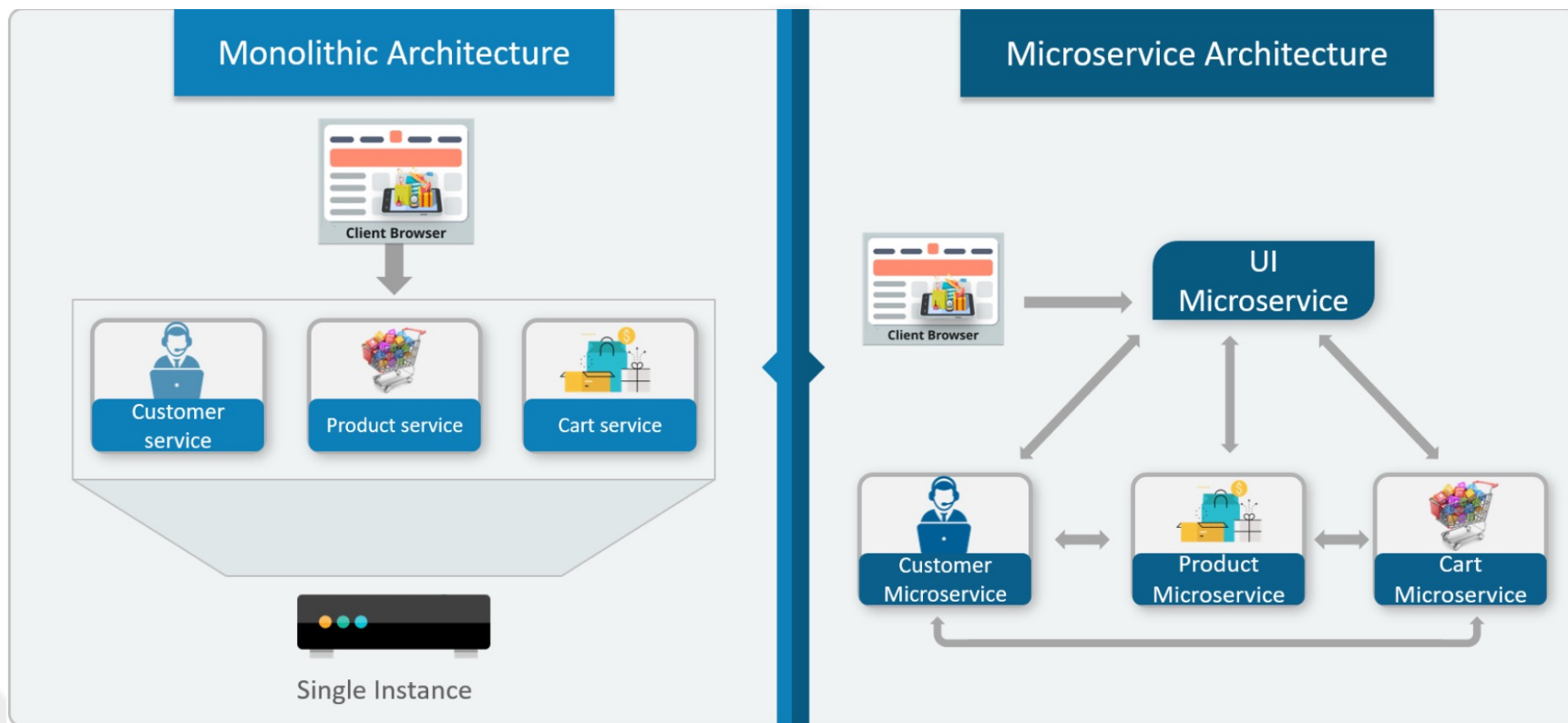
Que es un microservicio?

- 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**.

Que es un microservicio?



Arquitectura monolitica vs Arquitectura microservicios



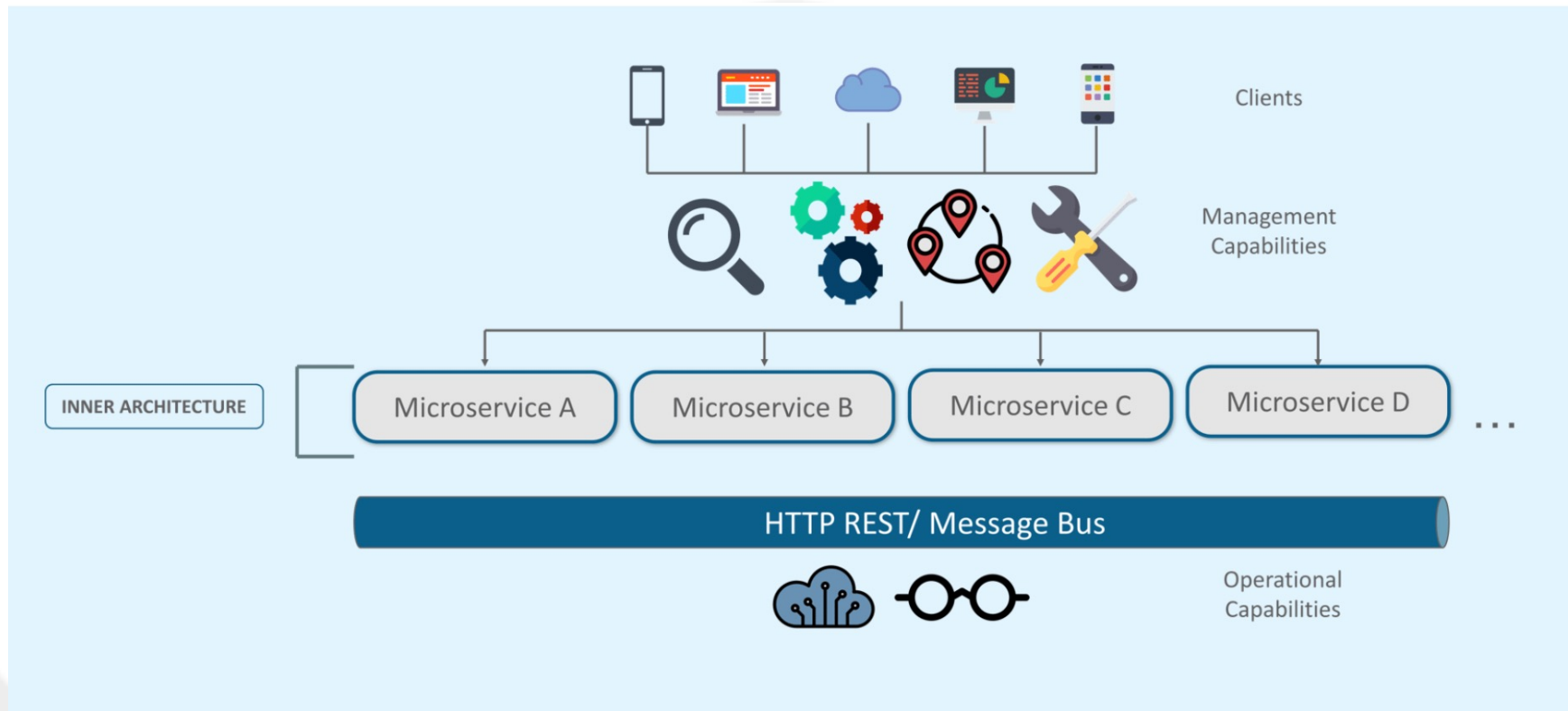
Tendencia en el desarrollo

- 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

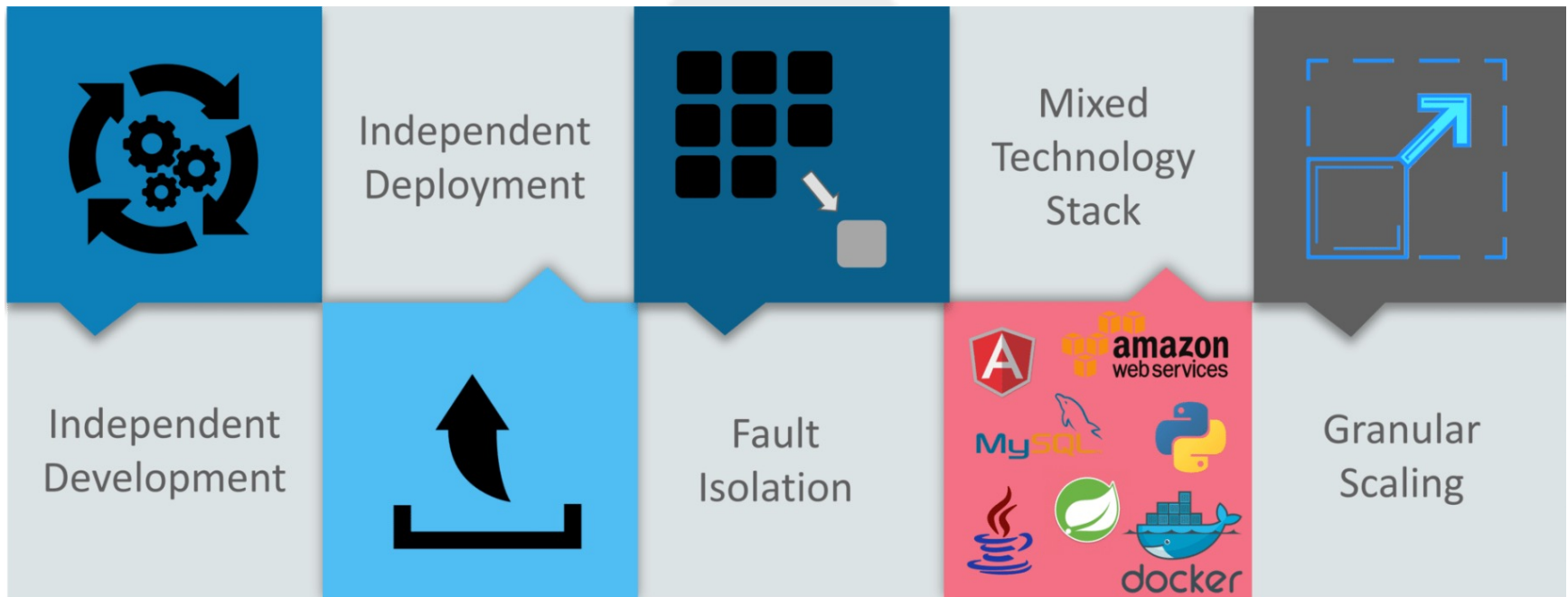
Características de los microservicios



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

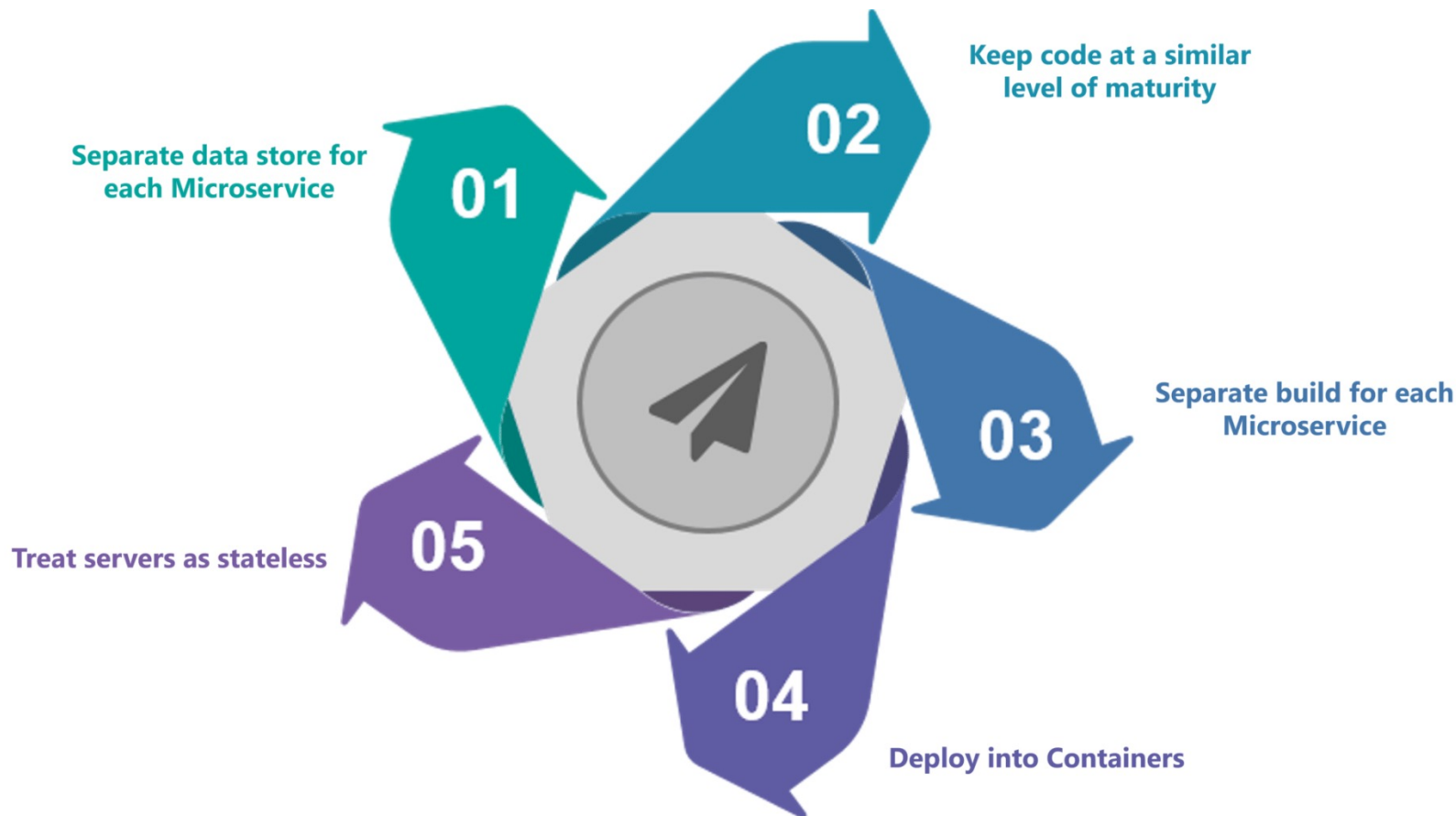
Ventajas de los microservicios



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 juntos

Buenas practicas diseño



Quien los utiliza?

amazon.com®

NETFLIX

GILT

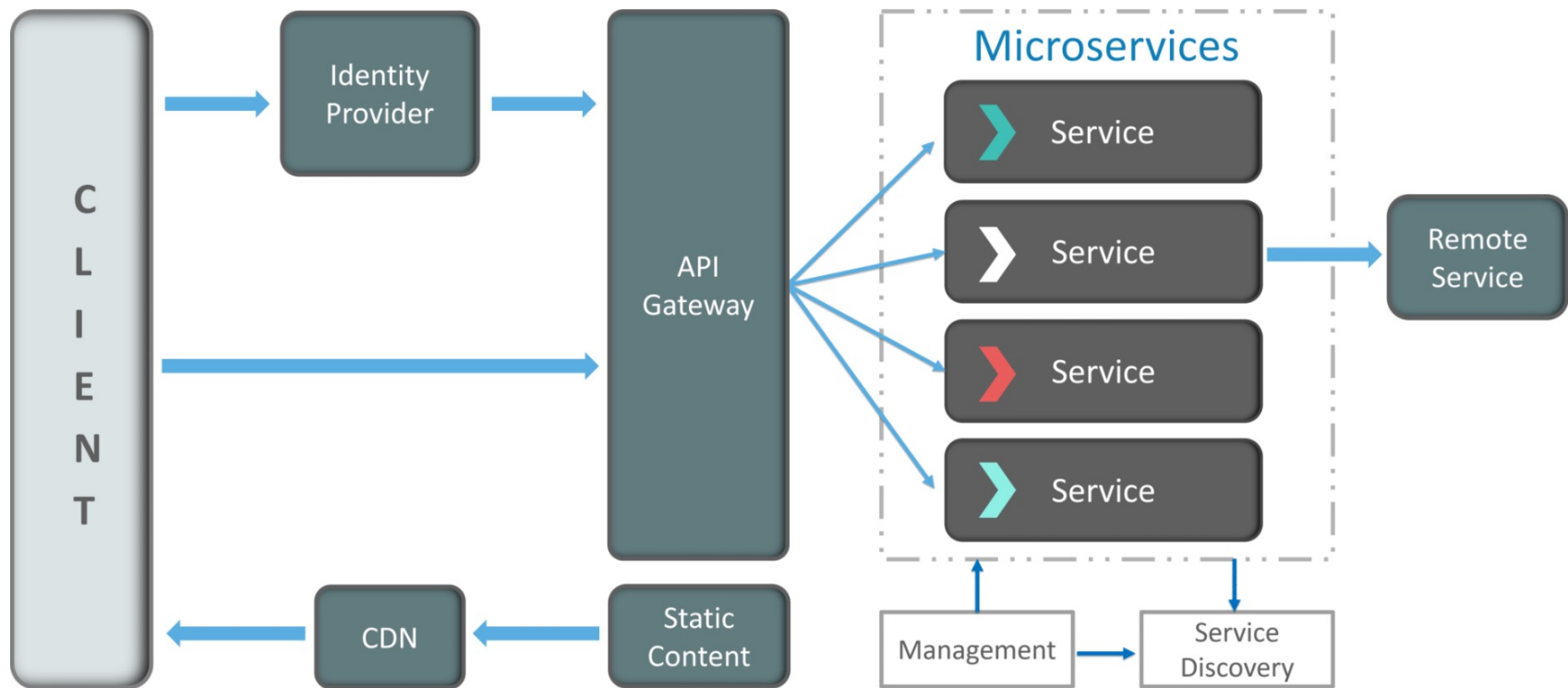


ebay

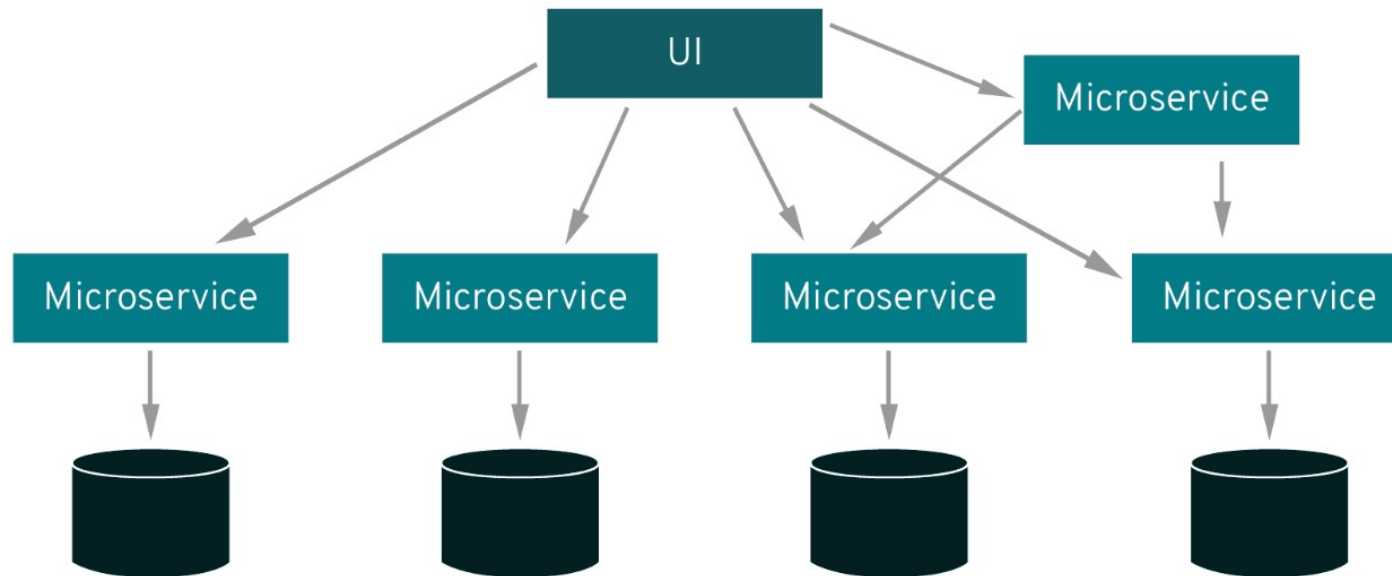

NORDSTROM

theguardian

Componentes de arquitectura



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

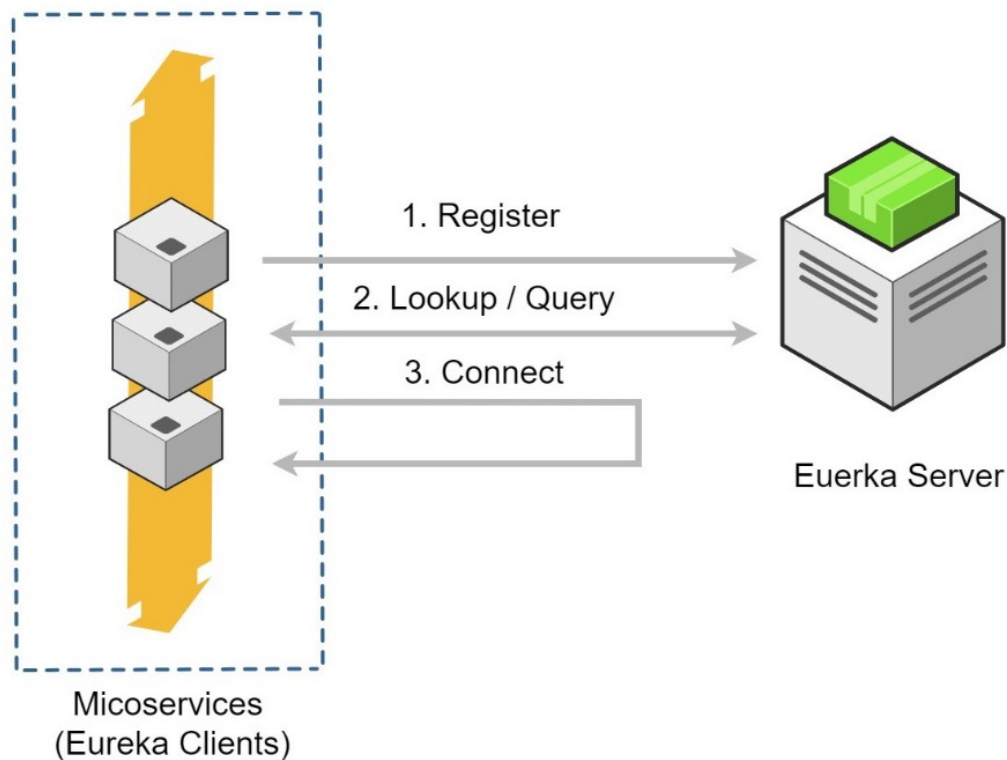
Balanceo de carga con Ribbon



Balanceo de carga con 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 asíncrono 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

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")
```

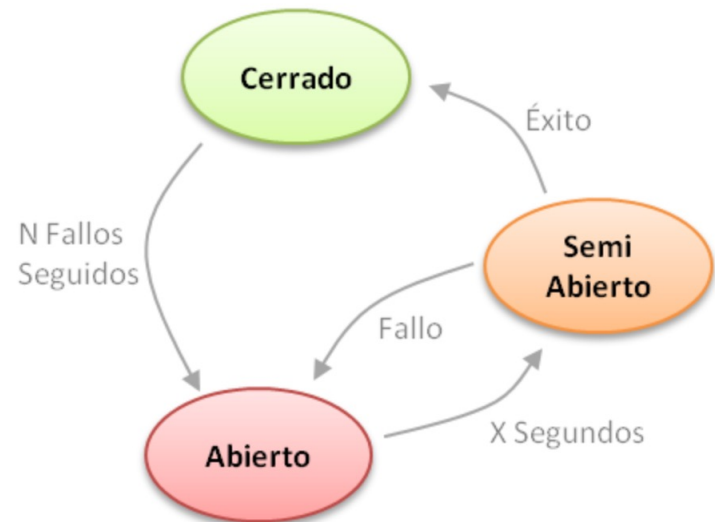
Resilience4J

- Resilience4J viene a sustituir a Hystrix.
- Dependencia necesaria

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

Resilience4J

- 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.



Resilience4J

- 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 segundos 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.

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>
```

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

Gracias por su asistencia