PROCESO DIRECCIÓN DE FORMACIÓN PROFESIONAL INTEGRAL

FORMATO GUÍA DE APRENDIZAJE

IDENTIFICACIÓN DE LA GUIA DE APRENDIZAJE

- Denominación del Programa de Formación: ANALISIS Y DESARROLLO DE SISTEMAS DE INFORMACION.
- Código del Programa de Formación: 228106
- Nombre del Proyecto:
- Fase del Proyecto: PLANEACIÓN
- Actividad de Proyecto: DISEÑAR LA ESTRUCTURA TECNOLÓGICA DEL SISTEMA INTEGRAL
- Competencia: PARTICIPAR EN EL PROCESO DE NEGOCIACIÓN DE TECNOLOGÍA INFORMÁTICA PARA PERMITIR LA IMPLEMENTACIÓN DEL SISTEMA DE INFORMACIÓN.
- Resultados de Aprendizaje: DEFINIR ESTRATEGIAS PARA LA ELABORACIÓN DE TÉRMINOS DE REFERENCIA Y PROCESOS DE EVALUACIÓN DE PROVEEDORES, EN LA ADQUISICIÓN DE TECNOLOGÍA, SEGÚN PROTOCOLOS ESTABLECIDOS.
- Duración de la Guía

2. PRESENTACIÓN

Para muchos, dominar la arquitectura de software es uno de los objetivos que han buscado durante algún tiempo, sin embargo, no siempre es claro el camino para dominarla, ya que la arquitectura de software es un concepto abstracto que cada persona lo interpreta de una forma diferente, dificultando con ello comprender y diseñar con éxito una arquitectura de software.

3. FORMULACIÓN DE LAS ACTIVIDADES DE APRENDIZAJE

- Materiales:
  Git, Visual code
  Portatil ó Computador de Escritorio
- Ambiente Requerido
- Descripción de la(s) Actividad(es)

ESTILOS ARQUITECTONICOS

Un estilo arquitectónico establece un marco de referencia a partir del cual es posible construir aplicaciones que comparten un conjunto de atributos y características mediante el cual es posible identificarlos y clasificarlos.

ARQUITECTURA MICROSERVICIOS

El estilo arquitectónico de Microservicios se ha convertido en el más popular de los últimos años, y es que no importa la conferencia o eventos de software te dirijas, en todos están hablando de los Microservicios, lo que la hace el estilo arquitectónico más relevante de la actualidad. El estilo de Microservicios consiste en crear pequeños componentes de software que solo hacen una tarea, la

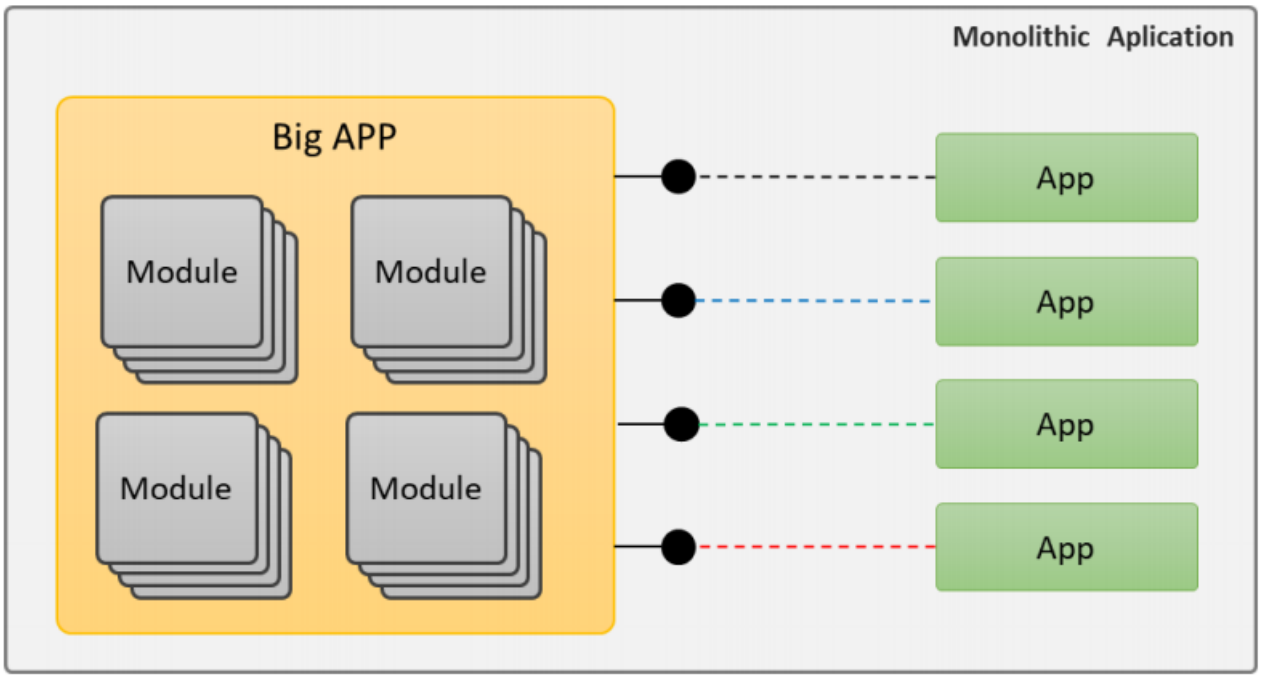
hace bien y son totalmente autosuficientes, lo que les permite evolucionar de forma totalmente independiente del resto de componentes.

El nombre de “Microservicios” se puede mal interpretar pensando que se trata de un servicio reducido o pequeño, sin embargo, ese es un concepto equivocado, los Microservicios son en realidad pequeñas aplicaciones que brindan un servicio, y observa que he dicho “brinda un servicio” en lugar de “exponen un servicio”, la diferencia radica en que los componentes que implementan un estilo de Microservicios no tiene por qué ser necesariamente un Web Services o REST, y en su lugar, puede ser un procesos que se ejecuta de forma programada, procesos que mueva archivos de una carpeta a otra, componentes que responde a eventos, etc. En este sentido, un Microservicios no tiene porqué exponer necesariamente un servicio, sino más bien, proporciona un servicio para resolver una determinada tarea del negocio. Un Microservicios es un pequeño programa que se especializa en realizar una pequeña tarea y se enfoca únicamente en eso, por ello, decimos que los Microservicios son Altamente Cohesivos, pues toda las operaciones o funcionalidad que tiene dentro está extremadamente relacionadas para resolver un único problema.

En este sentido, podemos decir que los Microservicios son todo lo contrario a las aplicaciones Monolíticas, pues en una arquitectura de Microservicios se busca desmenuzar una gran aplicación en muchos y pequeños componentes que realizar de forma independiente una pequeña tarea de la problemática general. Una de las grandes ventajas de los Microservicios es que son componentes totalmente encapsulados, lo que quiere decir que la implementación interna no es de interés para los demás componentes, lo que permite que estos evolucionen a la velocidad necesaria, además, cada Microservicios puede ser desarrollado con tecnologías totalmente diferentes, incluso, es normal que utilicen diferentes bases de datos.

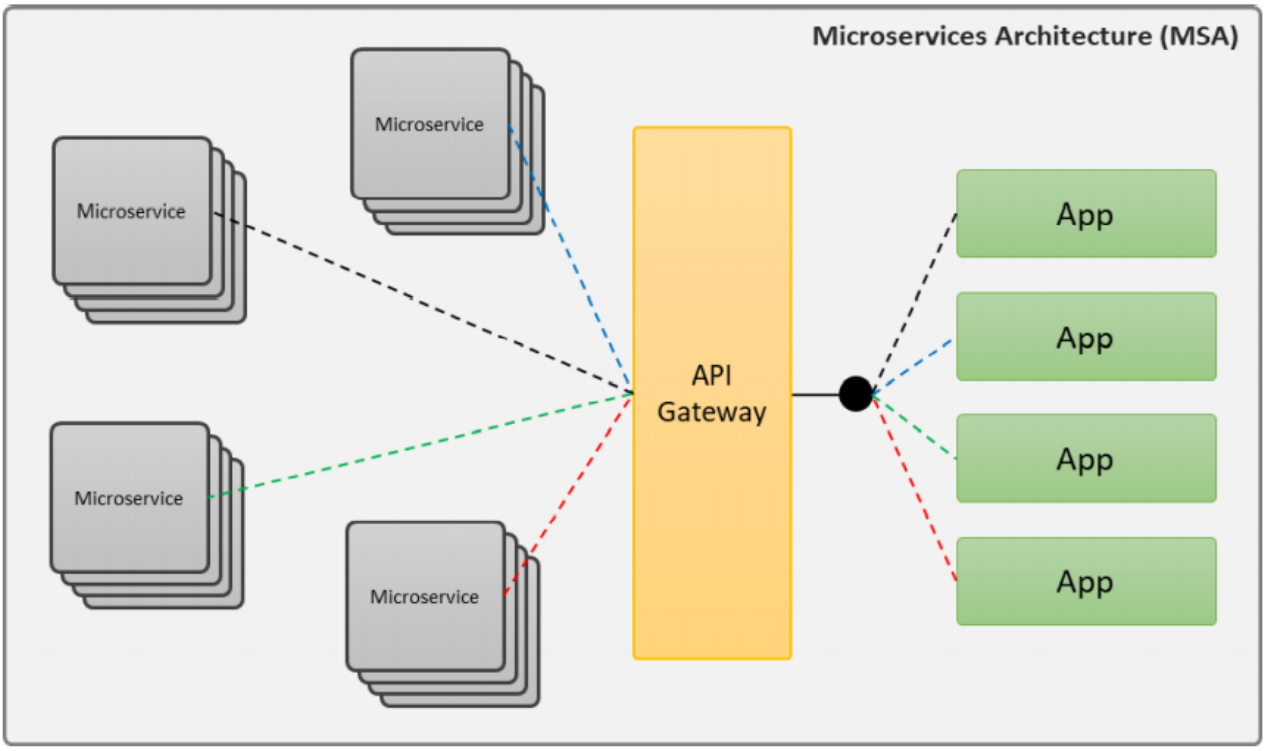
Como se estructura un Microservicios

Para comprender los Microservicios es necesario regresar a la arquitectura Monolítica, donde una solo aplicación tiene toda la funcionalidad para realizar una tarea de punta a punta, además, una arquitectura Monolítica puede exponer servicios, tal como lo vemos en la siguiente imagen:



Arquitectura monolítica

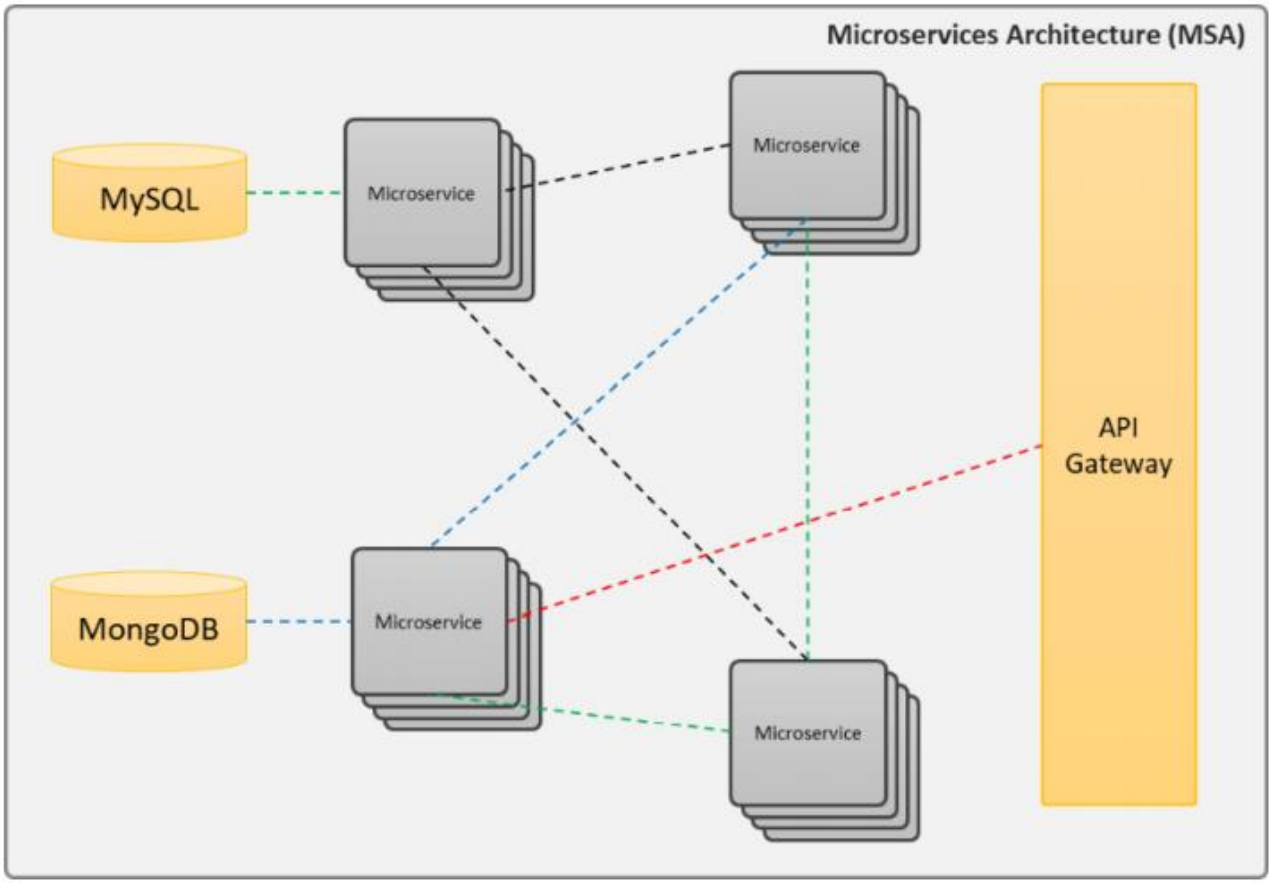
Una aplicación Monolítica tiene todos los módulos y funcionalidad necesario dentro de ella, lo que lo hace una aplicación muy grande y pesada, además, en una arquitectura Monolítica, es Todo o Nada, es decir, si la aplicación está arriba, tenemos toda la funcionalidad disponible, pero si está abajo, toda la funcionalidad está inoperable. La arquitectura de Microservicios busca exactamente lo contrario, dividiendo toda la funcionalidad en pequeños componentes autosuficientes e independientes del resto de componentes, tal y como lo puedes ver en la imagen anterior.



Arquitectura microservicios

En la arquitectura de Microservicios es común ver algo llamado API Gateway, el cual es un componente que se posiciona de cara a los microservicios para servir como puerta de entrada a los Microservicios, controlando el acceso y la funcionalidad que deberá ser expuesta a una red pública.

Debido a que los Microservicios solo realizan una tarea, es imposible que por sí solos no puedan realizar una tarea de negocio completa, por lo que es común que los Microservicios se comuniquen con otros Microservicios para delegar ciertas tareas, de esta forma, podemos ver que todos los Microservicios crean una red de comunicación entre ellos mismos, incluso, podemos apreciar que diferentes Microservicios funcionan con diferentes bases de datos.



Comunicación entre microservicios

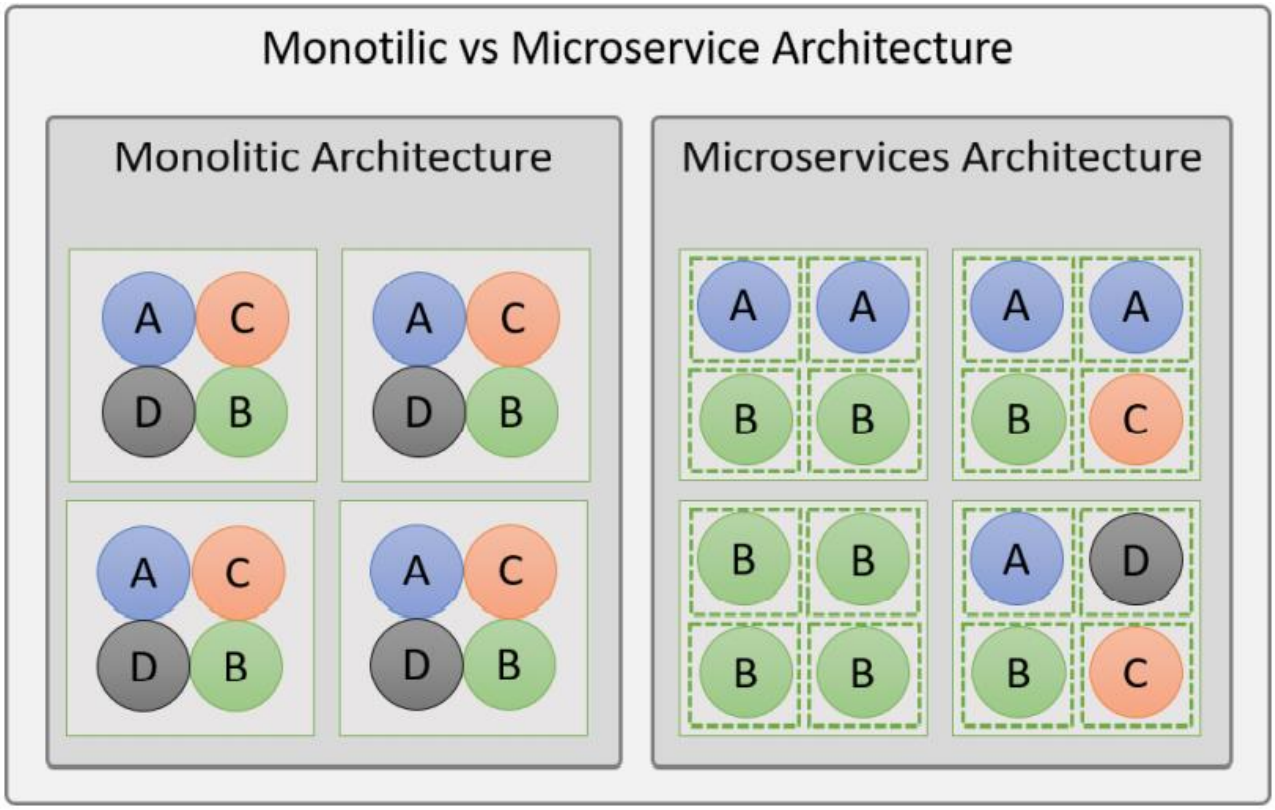
Algo a tomar en cuenta es que los Microservicios trabajan en una arquitectura distribuida, lo que significa todos los Microservicios son desplegados de forma independiente y están desacoplados entre sí. Además, deben ser accedidos por medio de protocolos de acceso remoto, como colas de mensajes, SOAP, REST, RPC, etc. Esto permite que cada Microservicio funcione o pueda ser desplegado independientemente de si los demás están funcionando.

Escalabilidad Monolítica vs escalabilidad de Microservicios

Una de las grandes problemáticas cuando trabajamos con arquitecturas Monolíticas es la escalabilidad, ya que son aplicaciones muy grandes que tiene mucha funcionalidad, la cual consume muchos recursos, se use o no, estos recursos tienen un costo financiero pues hay que tener el hardware suficiente para levantar todo, ya que recordemos que una arquitectura Monolítica es TODO o NADA, pero hay algo más, el problema se agudiza cuando necesitamos escalar la aplicación Monolítica, lo que requiere levantar un nuevo servidor con una réplica de la aplicación completa.

En una arquitectura Monolítica no podemos decidir qué funcionalidad desplegar y que no, pues la aplicación viene como un TODO, lo que nos obliga a escalar de forma Monolítica, es decir,

escalamos todos los componentes de la aplicación, incluso si es funcionalidad que casi no se utiliza o que no se utiliza para nada.



Escalamiento monolotico vs microservicios

Del lado izquierdo de la imagen anterior podemos ver como una aplicación Monolítica es escalada en 4 nodos, lo que implica que la funcionalidad A, B, C y D sean desplegadas por igual en los 4 nodos. Obviamente esto es un problema si hay funcionalidad que no se requiere, pero en un Monolítico no tenemos otra opción.

Del lado derecho podemos ver cómo es desplegado los Microservicios. Podemos ver que tenemos exactamente las mismas funcionalidades A, B, C y D, con la diferencia de que esta vez, cada funcionalidad es un Microservicio, lo que permite que cada funcionalidad sea desplegada de forma independiente, lo que permite que decidamos que componentes necesitan más instancias y cuales menos. Por ejemplo, pude que la funcionalidad B sea la más crítica, que es donde realizamos las ventas, entonces creamos 8 instancias de ese Microservicio, por otro lado, tenemos el componente que envía Mail, el cual no tiene tanda demanda, así que solo dejamos una instancia.

Como podemos ver, los Microservicios permite que controlemos de una forma más fina el nivel de escalamiento de los componentes, en lugar de obligarnos a escalar toda la aplicación de forma Monolítica.

Características de un Microservicio

En esta sección analizaremos las características que distinguen al estilo de Microservicios del resto. Las características no son ventajas o desventajas, si no que más bien, nos sirven para identificar cuando una aplicación sigue un determinado estilo arquitectónico.

Las características se pueden convertir en ventajas o desventajas solo cuando analizamos la problemática que vamos a resolver, mientras tanto, son solo características:

- Son componentes altamente cohesivos que se enfocan en realizar una tarea muy específica.
- Son componentes autosuficientes y no requieren de ninguna otra dependencia para funcionar.
- Son componentes distribuidos que se despliegan de forma totalmente independiente de otras aplicaciones o Microservicios.
- Utilizan estándares abiertos ligeros para comunicarse entre sí.
- Es normal que existan múltiples instancias del Microservicios funcionando al mismo tiempo para aumentar la disponibilidad.
- Los Microservicios son capaces de funcionar en hardware limitado, pues no requieren de muchos recursos para funcionar.
- Es común que una arquitectura completa requiere de varios Microservicios para ofrecer una funcionalidad de negocio completa.
- Es común ver que los Microservicios están desarrollados en tecnologías diferentes, incluido el uso de bases de datos distintas.

Ventajas

- Alta escalabilidad: Los Microservicios es un estilo arquitectónico diseñado para ser escalable, pues permite montar numerosas instancias del mismo componente y balancear la carga entre todas las instancias.
- Agilidad: Debido a que cada Microservicios es un proyecto independiente, permite que el componente tenga ciclo de desarrollo diferente del resto, lo que permite que se puedan hacer despliegues rápidos a producción sin afectar al resto de componentes.
- Facilidad de despliegue: Las aplicaciones desarrolladas como Microservicios encapsulan todo su entorno de ejecución, lo que les permite ser desplegadas sin necesidad de dependencias externas o requerimientos específicos de Hardware.
- Testabilidad: Los Microservicios son especialmente fáciles de probar, pues su funcionalidad es tan reducida que no requiere mucho esfuerzo, además, su naturaleza de exponer o brindar servicios hace que sea más fácil de crear casos específicos para probar esos servicios.
- Fácil de desarrollar: Debido a que los Microservicios tiene un alcance muy corto, es fácil para un programador comprender el alcance del componente, además, cada Microservicios puede ser desarrollado por una sola persona o un equipo de trabajo muy reducido.
- Reusabilidad: La reusabilidad es la médula espinal de la arquitectura de Microservicios, pues se basa en la creación de pequeños componentes que realice una única tarea, lo que hace que sea muy fácil de reutilizar por otras aplicaciones o Microservicios.

- Interoperabilidad: Debido a que los Microservicios utilizan estándares abiertos y ligeros para comunicarse, hace que cualquier aplicación o componente pueda comunicarse con ellos, sin importar en que tecnología está desarrollado.

Desventajas

- Performance: La naturaleza distribuida de los Microservicios agrega una latencia significativa que puede ser un impedimento para aplicaciones donde el performance es lo más importante, por otra parte, la comunicación por la red puede llegar a ser incluso más tardado que el proceso en sí.
- Múltiples puntos de falla: La arquitectura distribuida de los Microservicios hace que los puntos de falla de una aplicación se multipliquen, pues cada comunicación entre Microservicios tiene una posibilidad de fallar, lo cual hay que gestionar adecuadamente.
- Trazabilidad: La naturaleza distribuida de los Microservicios complica recuperar y realizar una traza completa de la ejecución de un proceso, pues cada Microservicio arroja de forma separa su traza o logs que luego deben de ser recopilados y unificados para tener una traza completa.
- Madurez del equipo de desarrollo: Una arquitectura de Microservicios debe ser implementada por un equipo maduro de desarrollo y con un tamaño adecuado, pues los Microservicios agregan muchos componentes que deben ser administrados, lo que puede ser muy complicado para equipo poco maduros.

Cuando debo de utilizar un estilo de Microservicios

Debido a que los Microservicios se han puesto muy moda en estos años, es normal ver que en todas partes nos incentivan a utilizarlos, sin embargo, su gran popularidad ha llegado a segar a muchos, pues alientan y justifican la implementación de microservicios para casi cualquier cosa y ante cualquier condición, muy parecido al anti-patrón Golden Hammer.

Los Microservicios son sin duda una excelente arquitectura, pero debemos tener en claro que no sirve para resolver cualquier problema, sobre todos en los que el performance es el objetivo principal.

Un error común es pensar que una aplicación debe de nacer desde el inicio utilizando Microservicios, lo cual es un error que lleva por lo general al fracaso de la aplicación, tal como lo comenta Martin Fowler en uno de sus artículos. Lo que él recomienda es que una aplicación debe nacer como un Monolítico hasta que llegue el momento que el Monolítico sea demasiado complicado de administrar y todos los procesos de negocio están muy maduros, en ese momento es cuando Martin Fowler recomienda que debemos de empezar a partir muestro Monolítico en Microservicios.

Puede resultar estúpido crear un Monolítico para luego partirlo en Microservicios, sin embargo, tiene mucho sentido. Cuando una aplicación nace, no se tiene bien definido el alcance y los procesos de negocio no son maduros, por lo que comenzar a fraccionar la lógica en Microservicios desde el inicio puede llevar a un verdadero caos, pues no se sabe a ciencia cierta qué dominio debe atacar cada

Microservicio, además, Empresas como Netflix, que es una de las que más promueve el uso de Microservicios también concuerda con esto.

Dicho lo anterior y analizar cuando NO debemos utilizar Microservicios, pasemos a analizar los escenarios donde es factible utilizar los Microservicios. De forma general, los Microservicios se prestan más para ser utilizadas en aplicaciones que:

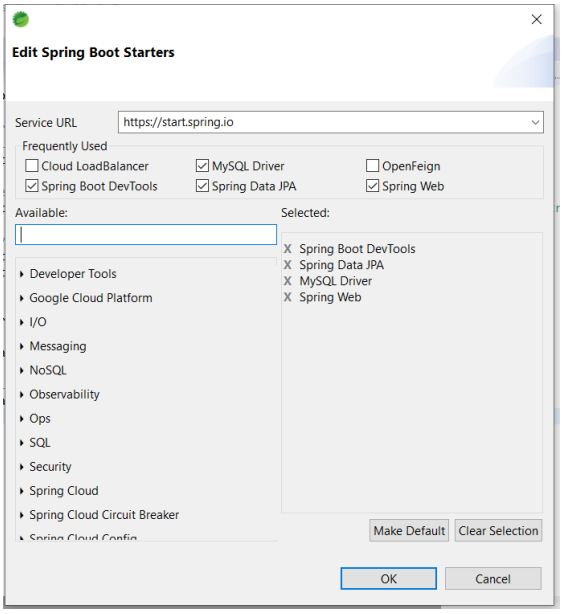
- Aplicaciones pensadas para tener un gran escalamiento
- Aplicaciones grandes que se complica ser desarrolladas por un solo equipo de trabajo, lo que hace complicado dividir las tareas si entrar en constante conflicto.
- Donde se busca agilidad en el desarrollo y que cada componente pueda ser desplegado de forma independiente.

Hoy en día es común escuchar que las empresas más grandes del mundo utilizan los Microservicios como base para crear sus aplicaciones de uso masivo, como es el caso claro de Netflix, Google, Amazon, Apple, etc.

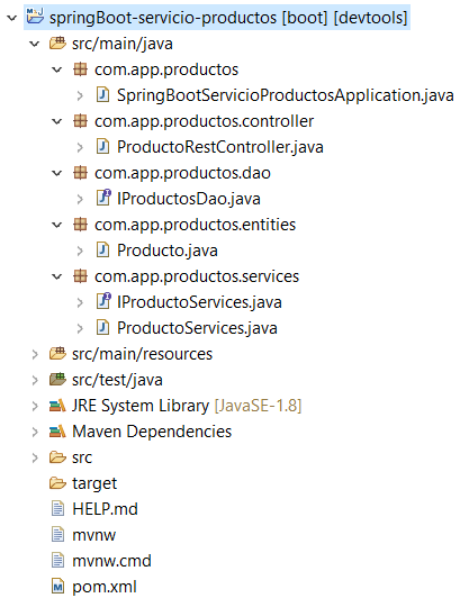
EJERCICIO

Suba el servicio de MySql, abrir el editor de mysql y cree un esquema llamado proyecto_microservicios.

Creación de servicio: Tenga en cuenta los 4 elementos seleccionados



Estructura del proyecto:



Application properties

```
spring.application.name=servicio-productos

#Data source
#Indica el driver/lib para conectar java a mysql
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

#Url donde esta el servicio de tu mysql y el nombre de la base de datos
spring.datasource.url=jdbc:mysql://localhost:3306/proyecto_microservicios?useUnicode=true&useJDBCCompliantTimezoneShift=true&

#Usuario y contrasena para tu base de datos descrita en la linea anterior
spring.datasource.username=root
spring.datasource.password=1234

#Port
server.port=8082

spring.jpa.hibernate.ddl-auto=update

#[opcional]Imprime en tu consola las instrucciones hechas en tu base de datos.
spring.jpa.show-sql = true
```

```
spring.datasource.url=jdbc:mysql://localhost:3306/proyecto_microservicios?useUnicode=true&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=UTC
```

Entidad:

```java
package com.app.productos.entities;

import java.io.Serializable;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table( name = "PRODUCTOS" )
public class Producto implements Serializable{

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue( strategy = GenerationType.IDENTITY )
    @Column( name = "ID_PRODUCTO_PK", nullable = false )
    private Long id;

    @Column( name = "NOMBRE", nullable = false )
    private String nombre;

    @Column( name = "PRECIO", nullable = false )
    private Double precio;

public Producto() {
}

public Long getId() {
    return id;
}
public void setId(Long id) {
    this.id = id;
}
public String getNombre() {
    return nombre;
}
public void setNombre(String nombre) {
    this.nombre = nombre;
}
public Double getPrecio() {
    return precio;
}
public void setPrecio(Double precio) {
    this.precio = precio;
}
@Override
public String toString() {
    return "Producto [id=" + id + ", nombre=" + nombre + ", precio=" + precio + "]";
}
```

IProductoDao

```java
package com.app.productos.dao;

import org.springframework.data.repository.CrudRepository;

import com.app.productos.entities.Producto;

public interface IProductosDao extends CrudRepository<Producto, Long> {

}
```

ProductoServices

```java
package com.app.productos.services;

import java.util.List;
import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import com.app.productos.dao.IProductosDao;
import com.app.productos.entities.Producto;

@Service
public class ProductoServices implements IProductoServices{

    @Autowired
    private IProductosDao productoDao;

    @Transactional(readOnly = true)
    public List<Producto> findAll() {
        return (List<Producto>) productoDao.findAll();
    }

    @Transactional(readOnly = true)
    public Optional<Producto> findById(Long id) {
        return productoDao.findById(id);
    }
```

IProductoServices

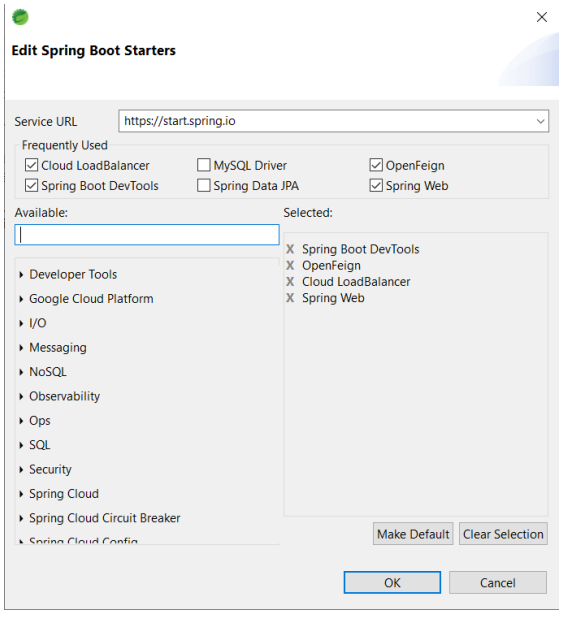
```java
package com.app.productos.services;

import java.util.List;
import java.util.Optional;

import com.app.productos.entities.Producto;

public interface IProductoServices {

    public List<Producto> findAll();

    public Optional<Producto> findById(Long id);

    public Producto create(Producto producto);
}
```

ProductoRestController

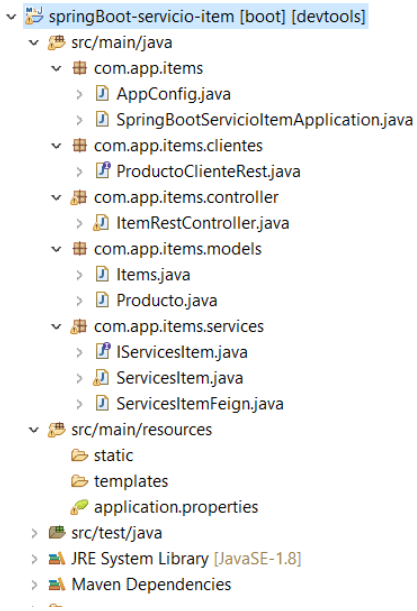
```java
package com.app.productos.controller;

import java.util.List;
import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.app.productos.entities.Producto;
import com.app.productos.services.IProductoServices;

@RestController
@RequestMapping("/api/productos")
public class ProductoRestController {

    @Autowired
    private IProductoServices productoService;

    @GetMapping("")
    public List<Producto> listar(){
        return productoService.findAll();
    }

    @GetMapping("/{id}")
    public Optional<Producto> ver(@PathVariable Long id){
        return productoService.findById(id);
    }
```

Agregue datos a la base de datos y pruebe con postman en la url = localhost:8082/api/productos/ o localhost:8082/api/productos/2

Cree otro proyecto dentro de la misma carpeta (otro microservicio) con las opciones seleccionadas



Estructura del proyecto:



Aplication properties

```
spring.application.name=servicio-items

#Port
server.port=8083

servicio-productos.loadbalancer.listOfServers=localhost:8082, localhost:9092
```

AppConfig

```
package com.app.items;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.client.RestTemplate;

@Configuration
public class AppConfig {

    @Bean("ClienteRest")
    public RestTemplate registrarRestTemplate() {
        return new RestTemplate();
    }
```

SpringBootServicioItemApplication

```java
package com.app.items;

import org.springframework.boot.SpringApplication;

@LoadBalancerClient(name = "servicio-productos")
@EnableFeignClients
@SpringBootApplication
public class SpringBootServicioItemApplication {

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

ProductoClienteRest

```java
package com.app.items.clientes;

import java.util.List;
import java.util.Optional;

import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;

import com.app.items.models.Producto;

@FeignClient(name = "servicio-productos")
public interface ProductoClienteRest {

    @GetMapping("")
    public List<Producto> listar();

    @GetMapping("/{id}")
    public Optional<Producto> ver(@PathVariable Long id);
}
```

ItemRestController

```java
package com.app.items.controller;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.app.items.models.Items;
import com.app.items.services.ServicesItem;

@RestController
@RequestMapping("/api/item")
public class ItemRestController {

    @Autowired
    private ServicesItem servicesItem;

    @GetMapping("")
    public List<Items> listar(){
        return servicesItem.findAll();
    }

    @GetMapping("/{id}/cantidad/{cantidad}")
    public Items detalle(@PathVariable Long id, @PathVariable Integer cantidad) {
        return servicesItem.findById(id, cantidad);
    }
}
```

Items

```java
package com.app.items.models;

import java.util.Optional;

public class Items {

    private Producto producto;

    private Integer cantidad;

    public Items() {
    }

    public Items(Producto producto, Integer cantidad) {
        this.producto = producto;
        this.cantidad = cantidad;
    }

    public Items(Optional<Producto> ver, Integer cantidad) {
        this.producto = ver.get();
        this.cantidad = cantidad;
    }

    public Producto getProducto() {
        return producto;
    }

    public void setProducto(Producto producto) {
        this.producto = producto;
    }


    public Integer getCantidad() {
        return cantidad;
    }

    public void setCantidad(Integer cantidad) {
        this.cantidad = cantidad;
    }

    public Double getTotal() {
        return producto.getPrecio() * cantidad.doubleValue();
    }

}
```

Producto

```java
package com.app.items.models;

public class Producto {

    private Long id;

    private String nombre;

    private Double precio;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getNombre() {
        return nombre;
    }

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

    public Double getPrecio() {
        return precio;
    }

    public void setPrecio(Double precio) {
        this.precio = precio;
    }
```

IServicesItem

```java
package com.app.items.services;

import java.util.List;

import com.app.items.models.Items;

public interface IServicesItem {

    public List<Items> findAll();

    public Items findById(Long id, Integer cantidad);
}
```

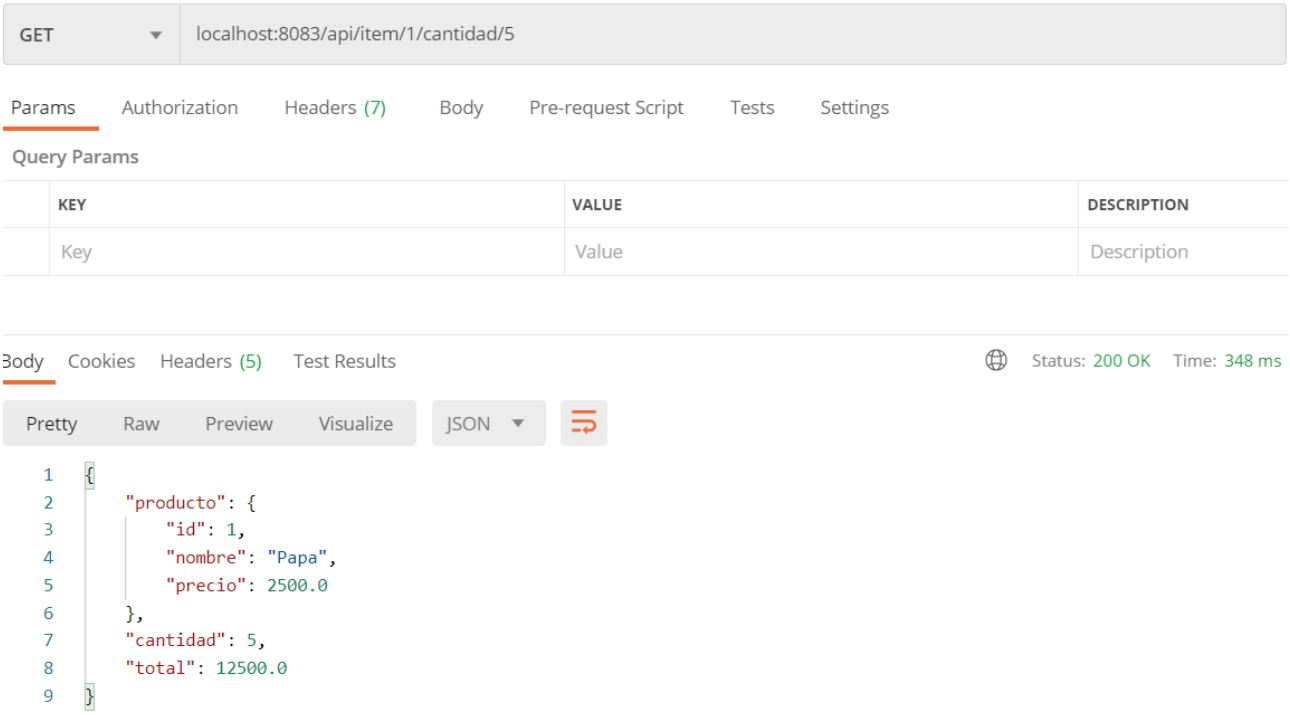
ServicesItem

```java
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Primary;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

import com.app.items.models.Items;
import com.app.items.models.Producto;

@Service
public class ServicesItem implements IServicesItem{

    @Autowired
    private RestTemplate clienteRest;

    @Override
    public List<Items> findAll() {
        List<Producto> productos = Arrays.asList(clienteRest.getForObject("http://localhost:8082/api/productos", Producto[].class));
        return productos.stream().map(p -> new Items(p,1)).collect(Collectors.toList());
    }

    @Override
    public Items findById(Long id, Integer cantidad) {
        Map<String, String> pathVariables = new HashMap<String, String>();
        pathVariables.put("id",id.toString());
        Producto producto = clienteRest.getForObject("http://localhost:8082/api/productos/{id}", Producto.class, pathVariables);
        return new Items(producto, cantidad);
    }
```

ServicesItemFeign

```java
package com.app.items.services;

import java.util.List;
import java.util.stream.Collectors;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Primary;
import org.springframework.stereotype.Service;

import com.app.items.clientes.ProductoClienteRest;
import com.app.items.models.Items;

@Service
@Primary
public class ServicesItemFeign implements IServicesItem {

    @Autowired
    private ProductoClienteRest clienteFeign;

    @Override
    public List<Items> findAll() {
        return clienteFeign.listar().stream().map(p -> new Items(p,1)).collect(Collectors.toList());
    }

    @Override
    public Items findById(Long id, Integer cantidad) {
        return new Items(clienteFeign.ver(id), cantidad);
    }

}
```

Pruebe el servicio en postman, debe subir primero el servicio de producto y luego el de ítems



4. ACTIVIDADES DE EVALUACIÓN

Evidencia de Conocimiento: Re-Diseñe el diagrama de secuencia y de despliegue del caso de estudio.

Evidencia de Desempeño: Realiza la sustentación de la implementación de un proyecto de microservicios.

Evidencia de Producto: Entrega de los documentos requeridos en las evidencias de conocimiento y el repositorio de github con un commit cada que realice la creación de un package con su respectivo comentario.

Evidencias de Aprendizaje	Criterios de Evaluación	Técnicas e Instrumentos de Evaluación
Evidencias de Conocimiento:	**Reconoce las principales características de la arquitectura por capas**	Redacción
Evidencias de Desempeño:	**Interpreta el uso de una aplicación arquitectura por capas en un entorno empresarial**	Presentación
Evidencias de Producto	**Realiza el entregable con la documentación completa**	Entrega de la guía con todas las evidencias requeridas.

1. GLOSARIO DE TÉRMINOS

De acuerdo a la práctica realizar su propio glosario de términos.

6. REFERENTES BIBLIOGRÁFICOS

Construya o cite documentos de apoyo para el desarrollo de la guía, según lo establecido en la guía de desarrollo curricular

7. CONTROL DEL DOCUMENTO

	Nombre	Cargo	Dependencia	Fecha
Autor (es)	**Néstor Rodríguez**	**Instructor**	**Teleinformática**	**NOVIEMBRE-2020**

8. CONTROL DE CAMBIOS (diligenciar únicamente si realiza ajustes a la guía)

	Nombre	Cargo	Dependencia	Fecha	Razón del Cambio
Autor (es)					