Departamento de Tecnología Digital

Tecsup TECNOLOGÍA CON SENTIDO

LABORATORIO N° 13

Desarrollo de aplicaciones Web: Microservicios.

DOCENTE:

Coello Palomino, Ricardo

CURSO:

Desarrollo de aplicaciones Web
5 - C24 - Sección A B C D

DESARROLLO DE APLICACIONES WEB AVANZADO: **Microservicios.**

I. Capacidades

Implementa una aplicación utilizando microservicios con Spring Boot.

II. Seguridad

- En este laboratorio está prohibida la manipulación del hardware, conexiones eléctricas o de red. Así como la ingesta de alimentos y bebidas.
- Ubicar maletines y/o mochilas en lugar destinado para tal fin.
- Dejar la mesa de trabajo y la silla utilizada limpias y ordenadas.

III. Fundamento teórico

- Revise el material de la semana correspondiente antes del desarrollo del laboratorio.

IV. Normas empleadas

- *No aplica*

V. Recursos

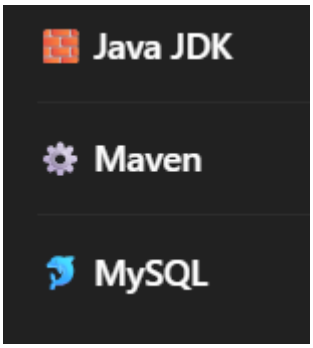
- En este laboratorio, cada estudiante trabajará con una computadora con Windows 10.
- La instalación del software requerido se realizará en el equipo virtual.

VI. Metodología para el desarrollo de la tarea

- El desarrollo del laboratorio es individual

VII. Procedimiento

REQUERIMIENTOS



PROCEDIMIENTO

- ***El laboratorio se ha diseñado para ser desarrollado en grupos de 2.***
- ***Solo un integrante sube el documento al Canvas.***

MARCO TEÓRICO

Los microservicios en **Spring Boot** son una arquitectura de desarrollo de software donde una aplicación se divide en pequeños servicios independientes, cada uno con una función específica, que se comunican entre sí a través de APIs ligeras (generalmente REST o gRPC). Spring Boot, como framework de Java, facilita la creación, configuración y despliegue de estos microservicios gracias a su enfoque en la simplicidad y la convención sobre configuración.

📊 Tabla Comparativa: MVC vs Hexagonal vs Microservicios

Criterio	MVC	Arquitectura Hexagonal	Microservicios
Nivel de complejidad	Bajo	Medio	Alto
Separación de capas	Modelo - Vista - Controlador	Núcleo (Dominio) + Puertos + Adaptadores	Cada servicio encapsula su propia lógica
Acoplamiento	Moderado (puede haber dependencia entre capas)	Bajo (núcleo desacoplado de tecnología)	Muy bajo entre servicios independientes
Escalabilidad	Limitada a escala monolítica	Modular, con posibilidad de transición a microservicios	Altamente escalable por servicio
Mantenibilidad	Disminuye con el tamaño del proyecto	Alta – cambios externos no afectan el núcleo	Alta – cada servicio evoluciona de forma aislada
Facilidad de pruebas	Aceptable, pero menos flexible	Muy alta – permite pruebas del dominio sin dependencias externas	Compleja – requiere pruebas distribuidas e integración
Curva de aprendizaje	Suave	Requiere mayor entendimiento de diseño de software	Empinada, tanto a nivel técnico como organizacional
Tecnologías comunes	Laravel, Spring MVC, Django	Clean Architecture, DDD, puertos/adaptadores manuales o con frameworks	Spring Boot, Node.js, Docker, Kubernetes, API Gateway, Eureka
Flexibilidad tecnológica	Baja – está muy ligado al framework utilizado	Alta – puedes cambiar bases de datos, UIs o frameworks sin afectar el núcleo	Muy alta – distintos lenguajes, bases de datos y stacks por servicio
Despliegue	Un único bloque (monolito)	Monolito modular o en capas	Despliegue independiente por servicio
Casos de uso ideales	Aplicaciones pequeñas o medianas con necesidades simples	Aplicaciones críticas, mantenibles, orientadas a dominio	Sistemas grandes, distribuidos y altamente disponibles
Desventajas clave	Difícil de escalar, alto acoplamiento en proyectos grandes	Mayor complejidad inicial, curva de diseño	Alta complejidad operativa y de infraestructura

Características principales de los microservicios en Spring Boot:

1. **Independencia**: Cada microservicio se desarrolla, despliega y escala de forma independiente. Por ejemplo, un microservicio puede manejar la autenticación, otro la gestión de usuarios, y otro el procesamiento de pagos.
2. **Modularidad**: Cada servicio tiene su propia base de datos (si aplica) y lógica de negocio, lo que permite usar diferentes tecnologías según las necesidades.
3. **Comunicación ligera**: Los microservicios se comunican mediante protocolos como HTTP/REST, gRPC o mensajería (por ejemplo, con Kafka o RabbitMQ).
4. **Despliegue flexible**: Pueden ejecutarse en contenedores (como Docker) y orquestarse con herramientas como Kubernetes.
5. **Escalabilidad**: Cada microservicio puede escalarse de forma independiente según la demanda, optimizando recursos.

Desafíos:

- **Complejidad distribuida**: Gestionar múltiples servicios requiere herramientas como orquestadores y monitoreo.
- **Latencia**: La comunicación entre servicios puede añadir retrasos.
- **Consistencia de datos**: Cada microservicio con su propia base de datos puede complicar la consistencia.

Ventajas de esta Arquitectura

Aislamiento: Cada contenedor (aplicación y base de datos) es independiente, con su propio entorno y dependencias.

Portabilidad: La configuración en `docker-compose.yml` y `Dockerfile` permite replicar el entorno en cualquier máquina con Docker.

Escalabilidad básica: Aunque este ejemplo usa un solo contenedor por servicio, Docker Compose permite escalar instancias de `app` (con ajustes adicionales, como un balanceador de carga).

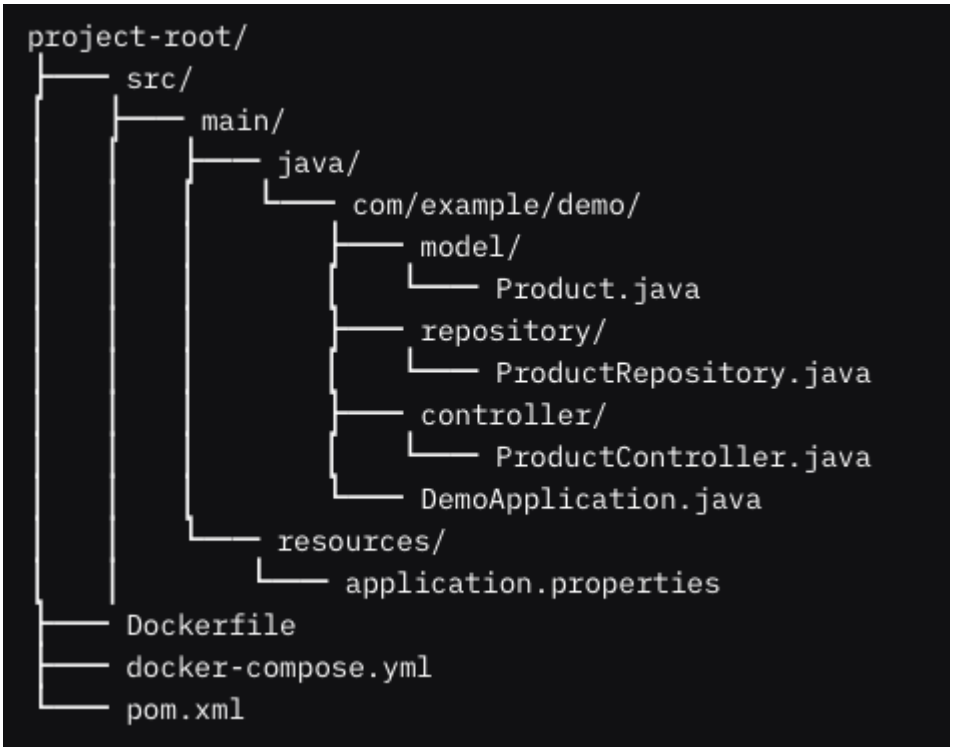
Persistencia: El volumen asegura que los datos de MySQL no se pierdan.

Facilidad de desarrollo: Los contenedores se configuran automáticamente, y la red interna simplifica la comunicación.

A. Creación de una aplicación en Spring Boot con dos contenedores (aplicación y base de datos MySQL) usando Docker. La aplicación tendrá un contenedor

para el backend Spring Boot y otro para MySQL, con un ejemplo simple de persistencia.

Estructura del proyecto:



1. Poner las dependencias.
   - JPA
   - MySQL
   - Spring Web

2. Creación del Modelo: Product.java

```
package com.example.demo.model;


import jakarta.persistence.Entity;

import jakarta.persistence.GeneratedValue;

import jakarta.persistence.GenerationType;

import jakarta.persistence.Id;


@Entity

@Table(name = "products")

public class Product {
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String warehouse;
    private Integer quantity;
    private Double price;

    // Getters and Setters
    public Long getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getWarehouse() {
        return warehouse;
    }

    public void setWarehouse(String warehouse) {
        this.warehouse = warehouse;
    }
```

```
    public Integer getQuantity() {
        return quantity;
    }

    public void setQuantity(Integer quantity) {
        this.quantity = quantity;
    }

    public Double getPrice() {
        return price;
    }

    public void setPrice(Double price) {
        this.price = price;
    }
}
```

3. Crear Repositorio: ProductRepository.java

```
package com.example.demo.repository;

import com.example.demo.model.Product;
import org.springframework.data.jpa.repository.JpaRepository;

public interface ProductRepository extends JpaRepository<Product, Long> {
}
```

4. Crear el Controlador: ProductController.java

```
package com.example.demo.controller;
```

```
import com.example.demo.model.Product;
import com.example.demo.repository.ProductRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

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

@RestController
@RequestMapping("/api/products")
public class ProductController {

    @Autowired
    private ProductRepository productRepository;

    // Listar todos los productos
    @GetMapping
    public List<Product> getAllProducts() {
        return productRepository.findAll();
    }

    // Obtener un producto por ID
    @GetMapping("/{id}")
    public ResponseEntity<Product> getProductById(@PathVariable Long id) {
        Optional<Product> product = productRepository.findById(id);
        return product.map(ResponseEntity::ok)
                      .orElseGet(() -> ResponseEntity.notFound().build());
    }

    // Crear un nuevo producto
    @PostMapping
    public Product createProduct(@RequestBody Product product) {
        return productRepository.save(product);
    }
```

```
    // Actualizar un producto
    @PutMapping("/{id}")
    public ResponseEntity<Product> updateProduct(@PathVariable Long id, @RequestBody Product
productDetails) {
        Optional<Product> product = productRepository.findById(id);
        if (product.isPresent()) {
            Product updatedProduct = product.get();
            updatedProduct.setName(productDetails.getName());
            updatedProduct.setPrice(productDetails.getPrice());
            return ResponseEntity.ok(productRepository.save(updatedProduct));
        }
        return ResponseEntity.notFound().build();
    }

    // Eliminar un producto
    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteProduct(@PathVariable Long id) {
        if (productRepository.existsById(id)) {
            productRepository.deleteById(id);
            return ResponseEntity.noContent().build();
        }
        return ResponseEntity.notFound().build();
    }
}
```

5. Configurar el application.properties

```
spring.datasource.url=jdbc:mysql://db:3306/microservicio_db
spring.datasource.username=root spring.datasource.password=rootpassword
spring.jpa.hibernate.ddl-auto=update spring.jpa.show-sql=true
```

6. Docker Compose: docker-compose.yml

```
version: '3.8'
services:
  app:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "8080:8080"
    environment:
      - SPRING_DATASOURCE_URL=jdbc:mysql://db:3306/microservicio_db
      - SPRING_DATASOURCE_USERNAME=root
      - SPRING_DATASOURCE_PASSWORD=rootpassword
    depends_on:
      - db
    networks:
      - app-network

  db:
    image: mysql:8.0
    environment:
      - MYSQL_ROOT_PASSWORD=rootpassword
      - MYSQL_DATABASE=microservicio_db
    ports:
      - "3306:3306"
    volumes:
      - db-data:/var/lib/mysql
    networks:
      - app-network

volumes:
```

```
  db-data:


networks:
  app-network:
    driver: bridge
```

7. Dockerfile.

```dockerfile
# Etapa de construcción
FROM maven:3.8.5-openjdk-17 AS build
WORKDIR /app
COPY pom.xml .
COPY src ./src
RUN mvn clean package -DskipTests

# Etapa de ejecución
FROM openjdk:17-jdk-slim
WORKDIR /app
COPY --from=build /app/target/*.jar app.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Detalles de los Contenedores:

- **Contenedor** `app`:
  - **Imagen**: Construida a partir del archivo `Dockerfile`, que define un proceso en dos etapas:
    1. **Etapa de compilación**: Usa `maven:3.8.5-openjdk-17` para compilar el proyecto Spring Boot (copiando `pom.xml` y el código fuente en `src`, y ejecutando `mvn clean package -DskipTests`).
    2. **Etapa de ejecución**: Usa `openjdk:17-jdk-slim` para ejecutar el JAR generado (`app.jar`) con el comando `java -jar app.jar`.
  - **Puertos**: Expone el puerto 8080 (mapeado al puerto 8080 del host), permitiendo acceder a la API REST en `http://localhost:8080`.
  - **Variables de entorno**: Configura la conexión a MySQL con:
    - `SPRING_DATASOURCE_URL=jdbc:mysql://db:3306/example_db`: Apunta al contenedor `db` usando su nombre como hostname.
    - `SPRING_DATASOURCE_USERNAME=root` y `SPRING_DATASOURCE_PASSWORD=rootpassword`: Credenciales para acceder a MySQL.
  - **Dependencias**: Usa `depends_on` para asegurar que el contenedor `db` esté listo antes de iniciar `app`.

- **Contenedor** `db`:
  - **Imagen**: Usa `mysql:8.0`, una imagen oficial de MySQL.
  - **Puertos**: Expone el puerto 3306 (mapeado al puerto 3306 del host), aunque en este caso la comunicación principal es interna a través de la red.
  - **Variables de entorno**:
    - `MYSQL_ROOT_PASSWORD=rootpassword`: Contraseña del usuario root.
    - `MYSQL_DATABASE=example_db`: Crea una base de datos llamada `example_db` al iniciar.
  - **Volumen**: Monta `db-data` en `/var/lib/mysql` para persistir los datos de la base de datos, evitando que se pierdan al reiniciar el contenedor.
  - **Red**: Comparte la misma red (`app-network`) que `app`.

Red de Comunicación:

- Docker Compose crea automáticamente una red puente (`bridge`) llamada `app-network`.
- Los contenedores `app` y `db` se comunican dentro de esta red usando sus nombres de servicio como hostnames (`app` y `db`).
- La aplicación Spring Boot se conecta a MySQL usando la URL `jdbc:mysql://db:3306/example_db`, donde `db` se resuelve al contenedor MySQL gracias a la red interna.
- La red aísla los contenedores del host, pero los puertos mapeados (8080 para la API y 3306 para MySQL) permiten acceso externo.

Persistencia de Datos:

3. Persistencia de Datos

- El volumen `db-data` asegura que los datos de MySQL (tablas, registros, etc.) persistan incluso si el contenedor `db` se elimina o reinicia.
- Sin este volumen, los datos se almacenarían en el sistema de archivos efímero del contenedor y se perderían al detenerlo.
- El volumen se monta en `/var/lib/mysql`, donde MySQL almacena sus archivos de datos.

8. Ejecución de los contenedores:
   - Instalar Docker.
   - Entrar https://labs.play-with-docker.com/

1. **Inicio de Docker Compose**:
   - Al ejecutar `docker-compose up -d`, Docker Compose lee `docker-compose.yml` y:
     - Crea la red `app-network`.
     - Crea el volumen `db-data`.
     - Construye la imagen para `app` usando el `Dockerfile` (si no existe).
     - Descarga la imagen `mysql:8.0` para `db` (si no está en caché).
     - Inicia los contenedores `db` y `app` en orden (primero `db` por `depends_on`).

2. **Inicialización de MySQL**:
   - El contenedor `db` inicia MySQL, crea la base de datos `example_db`, y configura el usuario root con la contraseña especificada.
   - Los datos se almacenan en el volumen `db-data`.
3. **Inicio de la Aplicación**:
   - El contenedor `app` ejecuta el JAR de Spring Boot.
   - Spring Boot lee `application.properties` para configurar la conexión a MySQL (`jdbc:mysql://db:3306/example_db`).
   - Spring Data JPA (con Hibernate) crea automáticamente la tabla `product` en `example_db` gracias a `spring.jpa.hibernate.ddl-auto=update`.
   - La aplicación expone la API REST en el puerto 8080.

4. **Interacción con la API**:
   - Al enviar una solicitud POST a `http://localhost:8080/api/products` con un JSON como `{"name":"Laptop","warehouse":"Main","quantity":10,"price":999.99}`:
     - El controlador `ProductController` recibe el JSON y lo mapea a un objeto `Product`.
     - El repositorio `ProductRepository` (JPA) guarda el objeto en la base de datos MySQL.
     - MySQL, en el contenedor `db`, persiste el registro en la tabla `product`.
   - Una solicitud GET a http://localhost:8080/api/products` consulta la tabla `product` en MySQL y retorna los registros.

5. Gestión de Contenedores

- **Construcción y ejecución**: `docker-compose up -d` inicia los contenedores en modo detach. Si cambias el código fuente, usa `docker-compose up -d --build` para reconstruir `app`.
- **Parada**: `docker-compose down` detiene y elimina los contenedores y la red, pero conserva el volumen `db-data`.
- **Parada con eliminación de datos**: `docker-compose down -v` elimina también el volumen, borrando los datos de MySQL.
- **Logs**: Usa `docker-compose logs` para depurar problemas en los contenedores.

Para trabajar con **GitHub** y **Play with Docker** (PWD) utilizando los documentos (`docker-compose.yml` y `Dockerfile`) que proporcionaste anteriormente, puedes seguir un flujo que combine el control de versiones con GitHub y la ejecución en PWD. A continuación, te detallo los pasos:

Paso 1: Configura tu proyecto en GitHub

1. **Crea un repositorio en GitHub**:
   - Ve a GitHub, inicia sesión y haz clic en "New repository".
   - Nombra el repositorio (ej. `spring-boot-microservice`) y hazlo público o privado según prefieras.
   - No inicialices el repositorio con un README, .gitignore ni licencia por ahora (lo haremos localmente).

Paso 2: Usa Play with Docker con el repositorio de GitHub

1. **Accede a Play with Docker**:
   - Ve a play-with-docker.com y autentícate con tu cuenta de Docker Hub o GitHub.
   - Haz clic en "Add new instance" para abrir un terminal.
2. **Clona tu repositorio**:
   - En el terminal de PWD, clona tu repositorio de GitHub:

```text
git clone https://github.com/tu-usuario/spring-boot-microservice.git
```

   - Cambia al directorio clonado:

```text
cd spring-boot-microservice
```

3. **Inicia los servicios con Docker Compose**:
   - Ejecuta el siguiente comando para construir e iniciar los contenedores:

```
text                                   Contraer  Ajuste  Copiar
docker-compose up --build
```

   - Esto construirá la imagen de la aplicación usando el `Dockerfile` y levantará los servicios `app` y `db` definidos en `docker-compose.yml`.
4. **Verifica y accede a los servicios**:
   - Lista los contenedores para confirmar que están corriendo:

```
text                                   Contraer  Ajuste  Copiar
docker ps
```

   - Haz clic en el ícono de "Ports" en la interfaz de PWD y abre los puertos 8080 (para la app) y 3306 (para MySQL, si necesitas conectarte).
   - Visita la URL proporcionada para el puerto 8080 (ej. `http://ip:8080`) para verificar que la aplicación Spring Boot responde.

Paso 3: Flujo de trabajo continuo

- **Actualiza el código en GitHub**:
  - Haz cambios en tu proyecto local (ej. modifica el controlador o añade nuevas funcionalidades).
  - Confirma y sube los cambios:

```
text                                   Contraer  Ajus
git add .
git commit -m "Añadido nueva funcionalidad"
git push origin master
```

- **Actualiza en Play with Docker:**
  - En PWD, vuelve al directorio del proyecto y actualiza el repositorio:

```
text                                                    Contraer

cd spring-boot-microservice
git pull origin master
```

  - Detén los contenedores anteriores:

```
text                                                    Contraer

docker-compose down
```

  - Vuelve a iniciarlos:

```
text                                                    Contraer

docker-compose up --build
```

B. Probar los siguientes endpoints de los dos microservicios en Docker o en Play with Docker.

- **GET /api/products**: Lista todos los productos.
- **GET /api/products/{id}**: Obtiene un producto por ID.
- **POST /api/products**: Crea un nuevo producto.
- **PUT /api/products/{id}**: Actualiza un producto.
- **DELETE /api/products/{id}**: Elimina un producto.

Nota: Abrir el puerto 8080 en el Docker Playground y copiar la URL.

Ejemplo:

- Método: `POST`
- URL: `http://ip-xxxxxx-xxxx-xxxxxxxxx-xxxxx-8080.direct.labs.play-with-docker.com/api/products`
- En la pestaña "Body", selecciona "raw" y elige "JSON".
- Ingresa un cuerpo como:

json — Contraer — Ajuste

```
{
  "name": "Laptop",
  "price": 999.99
}
```

- Haz clic en "Send".
- Deberías recibir una respuesta con el producto creado (incluyendo un ID generado).

Para acceder a la base de datos MySQL, usa el cliente MySQL desde la terminal de PWD. Conéctate con el siguiente comando, usando las credenciales definidas en el archivo `docker-compose.yml` (usuario `root`, contraseña `rootpassword`, y base de datos `microservicio_db`):

text | Contraer | Ajuste | Copiar

```
mysql -h 127.0.0.1 -P 3306 -u root -p microservicio_db
```

```
mysql> SHOW TABLES;
+----------------------------+
| Tables_in_microservicio_db |
+----------------------------+
| products                   |
+----------------------------+
1 row in set (0.00 sec)

mysql> SELECT * FROM products;
+----+------------------+--------+----------+-----------+
| id | name             | price  | quantity | warehouse |
+----+------------------+--------+----------+-----------+
|  1 | Teclado Mec?nico | 149.99 |       25 | B2        |
+----+------------------+--------+----------+-----------+
1 row in set (0.00 sec)

mysql>
```

Conclusiones:

Ejercicios de aplicación

Realizar un CRUD de una relación de muchos a muchos(Realizar el Frontend y Backend).

Nota: Subir el proyecto a GitHub y desplegar en Render, compartir los enlaces.

Conclusiones:

Indicar 5 conclusiones que llegó después de los temas tratados de manera práctica en este laboratorio.