DuocUC® INFORMÁTICA Y TELECOMUNICACIONES

Integración y Comunicación REST

- Desarrollo Fullstack II
- DSY1104

FORMAMOS PERONAS PARA UNA SOCIEDAD MEJOR

01

Repaso de la sesión anterior

¿Estás de acuerdo con estas afirmaciones?

1. *La configuración de Jasmine y Karma es demasiado complicada para aprender.*

2. *Jasmine y Karma son solo herramientas para pruebas unitarias y no son útiles para pruebas integradas.*

3. **No se necesita entender JavaScript para usar Jasmine y Karma.**

02

Integración y Comunicación REST

• ¿Qué es Spring Boot?

Framework de Java para crear aplicaciones stand-alone y de producción con Spring.

Características principales:

- Configuración automática
- Opinión por defecto
- Fácil de usar y rápido de configurar

• ¿Qué es Swagger?

Conjunto de herramientas para diseñar, construir, documentar y consumir APIs RESTful.

Beneficios:

- Documentación interactiva
- Generación de cliente y servidor
- Pruebas de API simplificadas

• Configuración del entorno

Requisitos de software:

- Java Development Kit (JDK) 8 o superior
- Maven o Gradle (gestor de dependencias)
- IDE (IntelliJ IDEA, Eclipse, VS Code)

• Configuración de Spring Boot

Agregar dependencias en pom.xml (para Maven):

pom.xml

C: > Users > Donacion > Desktop > pom.xml

```xml
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.5.5</version>
</parent>

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

• Creación de un proyecto Spring Boot

Uso de Spring Initializr

1. Ir a https://start.spring.io/
2. Seleccionar:

   Project: Maven

   Language: Java

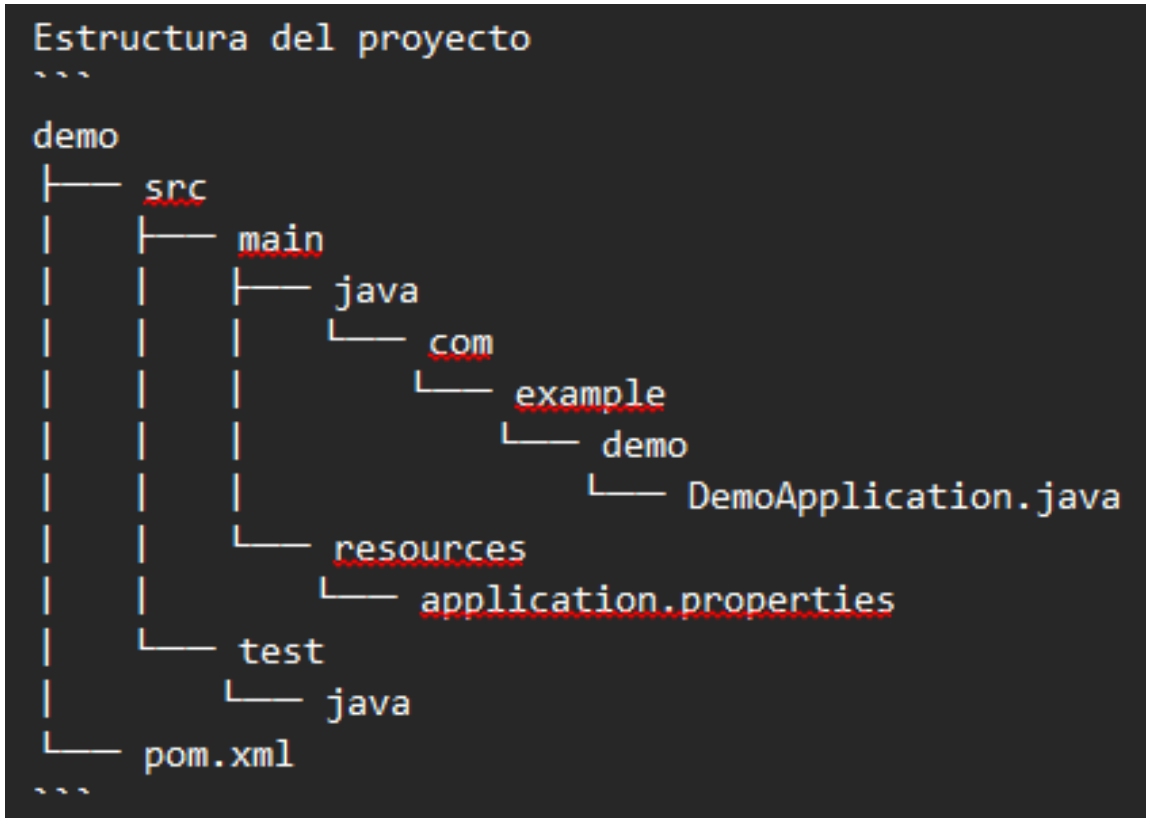
   Spring Boot: 2.5.5

   Group: com.example

   Artifact: demo

   Dependencies: Spring Web, JPA, H2

Estructura del proyecto



• Configuración básica de Spring Boot

application.properties.

Ejemplo de configuración básica:

```properties
server.port=8080
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.jpa.databaseplatform=org.hibernate.dialect.H2Dialect
```

Dependencias principales.

En pom.xml:

Users > Donacion > Desktop > pom.xml

```xml
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

Creación de modelos (entidades)

Ejemplo de una entidad simple

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

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

@Entity
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;
    private double price;

    // Constructores, getters y setters
}
```

• Implementación de Repositorios

¿Qué es un Repositorio?

Un repositorio es una clase que se encarga de la interacción con la base de datos. Spring Data JPA proporciona interfaces predefinidas como JpaRepository para simplificar esta tarea.

Uso de JpaRepository

```java
import org.springframework.data.jpa.repository.JpaRepository;

public interface ProductoRepository extends JpaRepository<Producto, Long> {
    // Métodos de consulta personalizados pueden ser añadidos aquí
}
```

• ¿Qué es un Servicio?

Un servicio en una aplicación, una capa que contiene la lógica de negocio. Se utiliza para mantener el código organizado, separando la lógica de negocio del acceso a datos y la interfaz de usuario.

Estructura Básica de un Servicio

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

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

@Service
public class ProductoService {

    @Autowired
    private ProductoRepository productoRepository;

    // Método para obtener todos los productos
    public List<Producto> obtenerTodosLosProductos() {
        return productoRepository.findAll();
    }

    // Método para obtener un producto por su ID
    public Optional<Producto> obtenerProductoPorId(Long id) {
        return productoRepository.findById(id);
    }

    // Método para guardar un producto
    public Producto guardarProducto(Producto producto) {
        return productoRepository.save(producto);
    }

    // Método para eliminar un producto por su ID
    public void eliminarProducto(Long id) {
        productoRepository.deleteById(id);
    }

    // Aquí se pueden agregar más métodos con la lógica de negocio específica
}
```

Ejemplo de Lógica de Negocio

```java
public Producto aplicarDescuento(Long id, Double porcentajeDescuento) {
    Optional<Producto> productoOpt = productoRepository.findById(id);

    if (productoOpt.isPresent()) {
        Producto producto = productoOpt.get();
        Double nuevoPrecio = producto.getPrecio() * (1 - porcentajeDescuento / 100);
        producto.setPrecio(nuevoPrecio);
        return productoRepository.save(producto);
    } else {
        throw new RuntimeException("Producto no encontrado");
    }
}
```

• ¿Qué es un Controlador REST?

Un controlador REST en una aplicación Spring Boot es responsable de manejar las solicitudes HTTP y devolver respuestas en formato JSON o XML. Permite que la lógica de negocio sea accesible a través de endpoints API.2.

Anotaciones Principales

@RestController: Se usa para definir una clase como un controlador REST, lo que implica que cada método en la clase devolverá datos directamente en el cuerpo de la respuesta (normalmente en formato JSON).

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

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

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

    @Autowired
    private ProductoService productoService;

    // Endpoint para obtener todos los productos
    @GetMapping
    public List<Producto> obtenerTodosLosProductos() {
        return productoService.obtenerTodosLosProductos();
    }

    // Endpoint para obtener un producto por su ID
    @GetMapping("/{id}")
    public Optional<Producto> obtenerProductoPorId(@PathVariable Long id) {
        return productoService.obtenerProductoPorId(id);
    }

    // Endpoint para crear un nuevo producto
    @PostMapping
    public Producto crearProducto(@RequestBody Producto producto) {
        return productoService.guardarProducto(producto);
    }

    // Endpoint para actualizar un producto existente
    @PutMapping("/{id}")
    public Producto actualizarProducto(@PathVariable Long id, @RequestBody Producto producto) {
        producto.setId(id);
        return productoService.guardarProducto(producto);
    }

    // Endpoint para eliminar un producto por su ID
    @DeleteMapping("/{id}")
    public void eliminarProducto(@PathVariable Long id) {
        productoService.eliminarProducto(id);
    }
}
```

Ejemplo de un Controlador REST Básico

@RequestMapping: Se utiliza para mapear solicitudes HTTP a métodos específicos dentro de un controlador. Puede aplicarse a nivel de clase o de método para definir la URL base o rutas específicas, y se puede personalizar para diferentes tipos de solicitudes HTTP (GET, POST, PUT, DELETE, etc.).

• Implementación de Operaciones CRUD

Las operaciones CRUD (Create, Read, Update, Delete) son fundamentales para cualquier aplicación que maneje datos persistentes. En un controlador REST, estas operaciones se implementan utilizando los métodos HTTP correspondientes: GET, POST, PUT y DELETE.

Operación GET: Leer Recursos

Objetivo: Obtener uno o varios recursos desde el servidor.

Anotación: @GetMappingEjemplo:

```java
// Obtener todos los productos
@GetMapping
public List<Producto> obtenerTodosLosProductos() {
    return productoService.obtenerTodosLosProductos();
}

// Obtener un producto por su ID
@GetMapping("/{id}")
public Optional<Producto> obtenerProductoPorId(@PathVariable Long id) {
    return productoService.obtenerProductoPorId(id);
}
```

Operación POST: Crear Recursos

Objetivo: Enviar datos al servidor para crear un nuevo recurso.

Anotación: @PostMappingEjemplo:

```java
// Crear un nuevo producto
@PostMapping
public Producto crearProducto(@RequestBody Producto producto) {
    return productoService.guardarProducto(producto);
}
```

Operación PUT: Actualizar Recursos

Objetivo: Actualizar un recurso existente en el servidor.

Anotación: @PutMapping

```
// Actualizar un producto existente
@PutMapping("/{id}")
public Producto actualizarProducto(@PathVariable Long id, @RequestBody Producto producto) {
    producto.setId(id);
    return productoService.guardarProducto(producto);
}
```

Operación DELETE: Eliminar Recursos

Objetivo: Eliminar un recurso del servidor.

Anotación: @DeleteMapping

```
// Eliminar un producto por su ID
@DeleteMapping("/{id}")
public void eliminarProducto(@PathVariable Long id) {
    productoService.eliminarProducto(id);
}
```

• Manejo de Excepciones

El manejo de excepciones en una aplicación REST es crucial para proporcionar respuestas significativas y coherentes a los clientes cuando ocurren errores. Spring proporciona varias herramientas para manejar excepciones de manera centralizada, siendo una de las más útiles. ***@ControllerAdvice***

• ¿Qué es @ControllerAdvice?

@ControllerAdvice es una anotación en Spring que permite manejar excepciones de manera global para todos los controladores REST. Proporciona una manera de capturar excepciones y devolver respuestas personalizadas sin necesidad de manejar las excepciones en cada controlador individualmente.

Ejemplo de Excepción Personalizada

```java
public class ResourceNotFoundException extends RuntimeException {
    public ResourceNotFoundException(String message) {
        super(message);
    }
}
```

• Configuración de Swagger en Spring Boot

Swagger es una herramienta útil para documentar y probar APIs REST de manera interactiva. Integrar Swagger en una aplicación Spring Boot permite generar una documentación automática de las APIs y proporciona una interfaz web para interactuar con ellas.

Dependencias Necesarias

Para configurar Swagger en Spring Boot, debes agregar las siguientes dependencias en tu archivo pom.xml (para proyectos Maven) o en build.gradle (para proyectos Gradle).

Para Maven

Users > Donacion > Desktop > pom.xml

```xml
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.2.0</version>
</dependency>
```

Para Gradle

```
dependencies {
    implementation 'org.springdoc:springdoc-openapi-starter-webmvc-ui:2.2.0'
}
```

Nota: springdoc-openapi es una implementación moderna de Swagger para Spring Boot 2.x y 3.x. Si estás utilizando una versión más antigua de Spring Boot, deberías considerar usar springfox-swagger2 y springfox-swagger-ui.

• Clase de Configuración (Opcional)

Con springdoc-openapi, la configuración mínima es automática, y no necesitas crear una clase de configuración manualmente. Sin embargo, si deseas personalizar aspectos de la documentación, puedes definir una clase de configuración.

```java
import org.springdoc.core.models.GroupedOpenApi;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class SwaggerConfig {

    @Bean
    public GroupedOpenApi api() {
        return GroupedOpenApi.builder()
                .group("v1")
                .pathsToMatch("/api/**")
                .build();
    }
}
```

• Acceder a la Documentación de Swagger

Una vez configurado, puedes acceder a la interfaz de Swagger UI en tu navegador. Por defecto, estará disponible en la siguiente URL:

```bash
http://localhost:8080/swagger-ui.html
```

Personalización Adicional (Opcional)

Puedes personalizar más aspectos de Swagger, como agregar una descripción global, información de contacto, etc., utilizando anotaciones como @OpenAPIDefinition en tu clase de configuración.

```java
import io.swagger.v3.oas.annotations.OpenAPIDefinition;
import io.swagger.v3.oas.annotations.info.Info;

@OpenAPIDefinition(info = @Info(title = "API de Productos", version = "1.0",
description = "Documentación de la API de Productos"))
public class SwaggerConfig {
    // Configuración adicional
}
```

03

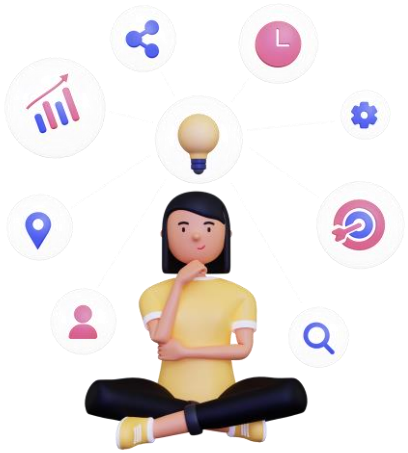
Actividad 3.1.2

ESCUELA DE INFORMÁTICA Y TELECOMUNICACIONES

Actividad 3.1.2

Ingresa al AVA de esta Actividad y desarrolla la actividad descrita en la guía 3.1.2 Desarrollo de Aplicaciones Web con Frameworks Backend

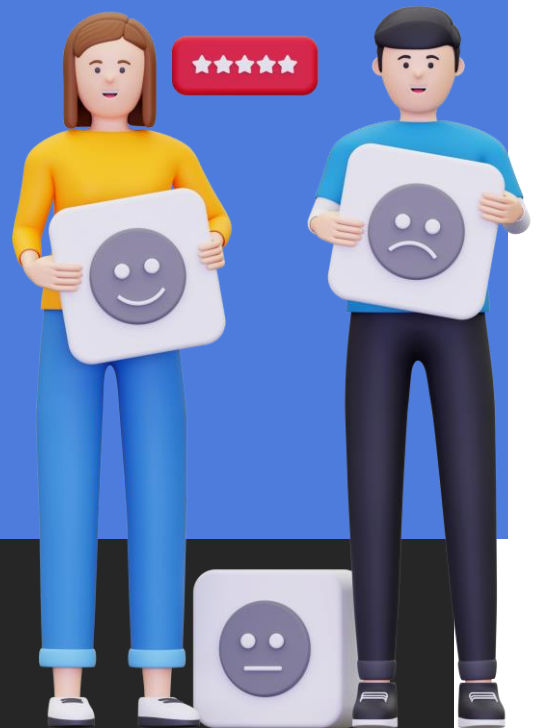
Consulta con tu docente las dudas que tengas al desarrollar la actividad.



DuocUC®

ESCUELA DE INFORMÁTICA Y TELECOMUNICACIONES

Cierre



¿Cómo podrías aplicar la integración y comunicación REST en desarrollo fullstack de las siguientes situaciones?

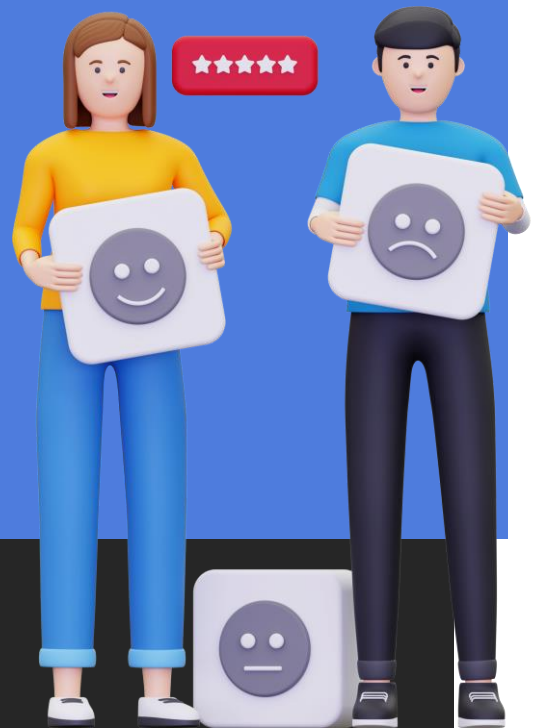
App de lista de tareas

•Frontend (React): Muestra la lista de tareas y permite al usuario agregar, editar o eliminar tareas.
•Backend (Node.js con Express): Proporciona endpoints REST para realizar operaciones CRUD en la base de datos de tareas.
•Comunicación REST:

- GET /api/tasks: Obtiene todas las tareas
- POST /api/tasks: Crea una nueva tarea
- PUT /api/tasks/ ID Actualiza una tarea existente
- DELETE /api/tasks/ ID Elimina una tarea

DuocUC®

Cierre



¿Cómo podrías aplicar la integración y comunicación REST en desarrollo fullstack de las siguientes situaciones?

App de comercio electrónico

•Frontend (React): Muestra la lista de tareas y permite al usuario agregar, editar o eliminar tareas.
•Backend (Node.js con Express): Proporciona endpoints REST para realizar operaciones CRUD en la base de datos de tareas.
•Comunicación REST:

- GET /api/tasks: Obtiene todas las tareas
- POST /api/tasks: Crea una nueva tarea
- PUT /api/tasks/ ID Actualiza una tarea existente
- DELETE /api/tasks/ ID Elimina una tarea

• Bibliografía

Libros Digitales Biblioteca Duoc. (Con tu cuenta de Duoc puedes consultar)

- Larsson, M. (2023). Microservices with Spring Boot 3 and Spring Cloud: Build resilient and scalable microservices using Spring Cloud, Istio, and Kubernetes (3a ed.). Packt Publishing.

Recursos de información.

- Spring Boot: “https://start.spring.io/

Duoc UC®

CERCANÍA. LIDERAZGO. FUTURO.

duoc.cl

7 AÑOS ACREDITADO

Comisión Nacional de Acreditación CNA-Chile

DESDE AGOSTO 2017 HASTA AGOSTO 2024.
DOCENCIA DE PREGRADO. GESTIÓN INSTITUCIONAL. VINCULACIÓN CON EL MEDIO.