

PROYECTO “dao-dto-rest”

¿Por qué usar DAO y DTO en una API REST?

Separar responsabilidades en una aplicación es una **buena práctica** para mejorar la organización y mantenibilidad del código.

Patrón DAO (Data Access Object)

- **Propósito:** Separa la **lógica de acceso a datos** de la lógica de negocio.
- **Ventajas:**
 - Facilita la **reutilización** y el **cambio de base de datos** sin afectar el resto del código.
 - Centraliza la **interacción con la base de datos**.
 - Hace que el código sea más **modular y limpio**.

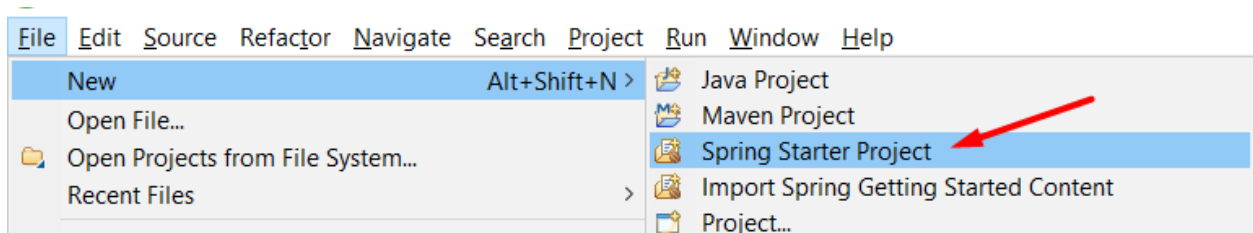
DTO (Data Transfer Object)

- **Propósito:** Separa los **datos de la entidad** de los datos que realmente necesita la API.
- **Ventajas:**
 - Evita exponer directamente las **entidades del modelo**.
 - Permite **optimizar** la respuesta de la API (por ejemplo, enviando solo ciertos campos).
 - Mejora la **seguridad** al ocultar datos sensibles.

Paso 1: Crear el Proyecto en STS

Abrir Spring Tool Suite (STS).

Ir a **File** → **New** → **Spring Starter Project**.



Completar los datos del proyecto:

- **Name:** cliente-api
- **Type:** Maven
- **Java Version:** 17
- **Group:** com.example
- **Artifact:** cliente-api
- **Packaging:** Jar

Service URL:

Name:

☒ Use default location

Location:

Type: Packaging:

Java Version: Language:

Group:

Artifact:

Version:

Description:

Package:

Working sets

☐ Add project to working sets

Working sets:

● Dependencies:

- Spring Web (Para la API REST)
- Spring Data JPA (Para manejar la base de datos)
- H2 Database (Base de datos en memoria)
- Lombok (Para reducir código repetitivo)

☒ H2 Database ☒ Lombok ☐ PostgreSQL Driver

☐ Spring Boot DevTools ☒ Spring Data JPA ☒ Spring Web


☐ Spring Web Services

Hacer clic en **Finish** para generar el proyecto.

Paso 2: Configurar la Base de Datos H2

Spring Boot ya trae H2 integrado con la dependencia, solo necesitamos configurarlo.

1. Abrir **src/main/resources/application.properties** y agregar



```

spring.application.name=cliente-api

spring.datasource.url=jdbc:h2:mem:clientesdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.h2.console.enabled=true

```

Esto configura H2 como una **base de datos en memoria** accesible desde:

link del h2: <http://localhost:8080/h2-console>

JDBC URL: jdbc:h2:mem:clientesdb

Usuario: sa | Sin contraseña

Luego das clic en el botón Connect

English Preferences Tools Help

Login

Saved Settings: Generic H2 (Embedded) Save Remove

Setting Name: Generic H2 (Embedded)

Driver Class: org.h2.Driver

JDBC URL: jdbc:h2:mem:clientesdb

User Name: sa

Password:

Connect Test Connection

jdbc:h2:mem:clientesdb
CLIENTES
INFORMATION_SCHEMA
Users
H2 2.3.232 (2024-08-11)

Run Run Selected Auto complete Clear SQL statement:

SELECT * FROM CLIENTES

SELECT * FROM CLIENTES;

ID	EMAIL	NOMBRE
1	juan@example.com	Juan Pérez

(4 rows, 0 ms)

Paso 3: Crear la Entidad Cliente

En `com.example.clienteapi.model`, crear `Cliente.java`:

```
package com.example.clienteapi.model;

import jakarta.persistence.*;
import lombok.Data;

@Data
@Entity
@Table(name = "clientes")
public class Cliente {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nombre;
    private String email;
}
```

Explicación:

- **@Entity**: Marca la clase como una tabla en la BD.
- **@Table(name = "clientes")**: Especifica el nombre de la tabla.
- **@Id @GeneratedValue**: Genera automáticamente el ID.

Paso 4: Crear el DTO (ClienteDTO)

Solo devolveremos id y nombre, sin exponer email.

1. En **com.example.clienteapi.dto**, crear **ClienteDTO.java**



```
package com.example.clienteapi.dto;

import lombok.AllArgsConstructor;
import lombok.Data;

@Data
@AllArgsConstructor
public class ClienteDTO {
    private Long id;
    private String nombre;
}
```

Explicación:

- Solo tiene id y nombre (no email).
- **@AllArgsConstructor** genera un **constructor automático**.

Paso 5: Crear la Capa DAO (Repositorio)

DAO es la capa de acceso a datos.

1. En **com.example.clienteapi.dao**, crear **ClienteRepository.java**:



```
package com.example.clienteapi.dao;

import com.example.clienteapi.model.Cliente;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface ClienteRepository extends JpaRepository<Cliente, Long> {
}
```

Explicación:

- **JpaRepository<Cliente, Long>**: Proporciona métodos CRUD listos para usar.
- **@Repository**: Marca la clase como DAO.

Paso 6: Crear la Capa de Servicio

Convierte Cliente en ClienteDTO.

1. En **com.example.clienteapi.service**, crear **ClienteService.java**:



```
package com.example.clienteapi.service;

import com.example.clienteapi.dao.ClienteRepository;
import com.example.clienteapi.dto.ClienteDTO;
import com.example.clienteapi.model.Cliente;
import org.springframework.stereotype.Service;

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

@Service
public class ClienteService {

    private final ClienteRepository clienteRepository;

    public ClienteService(ClienteRepository clienteRepository) {
        this.clienteRepository = clienteRepository;
    }

    public List<ClienteDTO> obtenerTodos() {
        return clienteRepository.findAll().stream()
            .map(cliente -> new ClienteDTO(cliente.getId(), cliente.getNombre()))
            .collect(Collectors.toList());
    }

    public Cliente guardar(Cliente cliente) {
        return clienteRepository.save(cliente);
    }
}
```

Explicación:

- **obtenerTodos():** Convierte **Cliente** a **ClienteDTO** usando stream().
- **guardar():** Guarda un nuevo cliente.

Paso 7: Crear el Controlador REST

En `com.example.clienteapi.controller`, crear `ClienteController.java`:



```
package com.example.clienteapi.controller;

import com.example.clienteapi.dto.ClienteDTO;
import com.example.clienteapi.model.Cliente;
import com.example.clienteapi.service.ClienteService;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/clientes")
public class ClienteController {

    private final ClienteService clienteService;

    public ClienteController(ClienteService clienteService) {
        this.clienteService = clienteService;
    }

    @GetMapping
    public List<ClienteDTO> obtenerTodos() {
        return clienteService.obtenerTodos();
    }

    @PostMapping
    public ResponseEntity<Cliente> guardar(@RequestBody Cliente cliente) {
        return ResponseEntity.ok(clienteService.guardar(cliente));
    }
}
```

Explicación:

- **GET /api/clientes:** Retorna la lista de clientes en formato DTO.
- **POST /api/clientes:** Guarda un cliente y lo devuelve.

Paso 8: Ejecutar y Probar

Crear un Cliente (POST):

- **POST** `http://localhost:8080/api/clientes`



```
{  
  "nombre": "Juan Pérez",  
  "email": "juan@example.com"  
}
```

POST http://localhost:8080/api/clientes

Params Authorization Headers (8) Body Scripts Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1 {  
2   "nombre": "Juan Pérez",  
3   "email": "juan@example.com"  
4 }
```

Body Cookies Headers (5) Test Results

200 OK • 273 ms • 222 B

Save Response

JSON Preview Visualize

```
1 {  
2   "id": 1,  
3   "nombre": "Juan Pérez",  
4   "email": "juan@example.com"  
5 }
```

Obtener Clientes (GET):

- **GET** http://localhost:8080/api/clientes

GET http://localhost:8080/api/clientes

Params Authorization Headers (6) Body Scripts Tests Settings

Query Params

Key	Value	Description
Key	Value	Description

Body Cookies Headers (5) Test Results

200 OK • 64 ms • 197 B

Save Response

JSON Preview Visualize

```
1 [  
2   {  
3     "id": 1,  
4     "nombre": "Juan Pérez"  
5   }  
6 ]
```