Guía Paso a Paso para Introducción a Spring y Desarrollo Backend con CRUD y Spring Data JPA

Objetivo General

Presentar de forma evolutiva el desarrollo de aplicaciones Java con Spring Framework desde un proyecto básico de consola con Beans gestionados por el contenedor, pasando por una API REST en memoria, hasta llegar a una aplicación con acceso a base de datos H2 usando Spring Data JPA. Se incluye al final una colección para REST Client (VS Code) que permita probar todos los endpoints.

Etapa 1: Proyecto básico de consola con Spring Initializr

Objetivo de la versión de consola

Mostrar la configuración del contenedor Spring, creación de Beans y aplicación de inyección de dependencias (IoC).

🚀 Spring Initializr: Creación del Proyecto Spring Boot REST con Swagger

1. Paso 1: Generar el proyecto con Spring Initializr

   Ingresá a https://start.spring.io y completá los siguientes campos:

   - **Project:** Maven
   - **Language:** Java
   - **Spring Boot:** 3.2.0 (o versión estable más reciente)
   - **Group:** com.example
   - **Artifact:** paso01-consola
   - **Name:** "Paso 01 - App de Consola"
   - **Description:** Demo de Spring de Consola
   - **Package name:** utnfc.isi.back.spring
   - **Packaging:** Jar
   - **Java:** 17 o superior

2. Descargar y descomprimir el proyecto generado

3. Abrirlo en VS Code, IntelliJ o tu IDE preferido

Ejemplo de Beans

Clase Servicio

```
import org.springframework.stereotype.Component;

// Anotación que indica que esta clase será gestionada como un Bean por el
contenedor de Spring
```

```java
@Component
public class SaludoService {

    // Método público que retorna un saludo personalizado
    public String saludar(String nombre) {
        return "Hola, " + nombre + "! Bienvenido a Spring.";
    }
}
```

🔍 Explicación Este fragmento define un Bean gestionado por Spring. La anotación @Component indica al contenedor de Spring que esta clase debe ser detectada automáticamente durante el escaneo de clases y registrada como un componente disponible para inyección.
SaludoService es un ejemplo de servicio simple, que encapsula una lógica reutilizable (en este caso, generar un saludo personalizado). Es ideal para introducir el concepto de Inversión de Control (IoC), donde la creación y gestión del objeto queda en manos del framework, facilitando el desacoplamiento y la reutilización.

Uso mediante obtención manual del Bean

```java
// Importa el contenedor principal de Spring que maneja los beans
import org.springframework.context.ApplicationContext;

// Proporciona un contexto basado en configuración anotada y escaneo de
componentes
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

// Importa la clase del servicio a utilizar
import com.example.demo.SaludoService;

public class Main {
    public static void main(String[] args) {
        // Crea un contexto de aplicación escaneando el paquete indicado
        ApplicationContext context = new
AnnotationConfigApplicationContext("com.example.demo");

        // Solicita al contenedor una instancia del bean SaludoService
        SaludoService saludo = context.getBean(SaludoService.class);

        // Usa el servicio para mostrar un mensaje en consola
        System.out.println(saludo.saludar("Mundo"));
    }
}
```

🔍 Explicación
Este fragmento muestra cómo utilizar el contenedor de Spring manualmente desde una aplicación de consola.
Se instancia un ApplicationContext, el contenedor que se encarga de gestionar los objetos anotados

> como @Component (u otras anotaciones estereotipo). En este caso, se utiliza AnnotationConfigApplicationContext para inicializar el contexto a partir del escaneo del paquete "com.example.demo", donde debe encontrarse el SaludoService.
> Luego, se solicita explícitamente el bean SaludoService mediante getBean(...), ejemplificando cómo Spring maneja la Inversión de Control (IoC): nosotros no creamos directamente el objeto, sino que se lo pedimos al contenedor.
> Este patrón habilita la inyección automática de dependencias en fases posteriores y promueve una arquitectura desacoplada, testeable y flexible.

Uso mediante Inyección de Dependencia

Este bloque es clave para introducir el uso de inyección de dependencias automática con @Autowired. A continuación te presento:

```
// Permite usar la anotación @Autowired para la inyección automática de
dependencias
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

// Indica que esta clase será registrada como Bean en el contenedor de Spring
@Component
public class Aplicacion {

    // Dependencia inyectada: se declara como final para reforzar la inmutabilidad
    private final SaludoService saludoService;

    // Constructor con anotación @Autowired para inyección de la dependencia
    @Autowired
    public Aplicacion(SaludoService saludoService) {
        this.saludoService = saludoService;
    }

    // Método que utiliza la dependencia inyectada
    public void ejecutar() {
        System.out.println(saludoService.saludar("Felipe"));
    }
}

// Clase Principal
public class Main {
    public static void main(String[] args) {
        // Inicialización del contenedor Spring escaneando el paquete
        ApplicationContext context = new
AnnotationConfigApplicationContext("com.example.demo");

        // Obtención del bean de tipo Aplicacion, con las dependencias ya
inyectadas
        Aplicacion app = context.getBean(Aplicacion.class);

        // Ejecución de la lógica principal de la aplicación
        app.ejecutar();
```

```
    }
}
```

> 🔍 Explicación
> Este bloque introduce el uso de inyección de dependencias automática con la anotación @Autowired en el constructor de la clase Aplicacion.
> Spring detecta automáticamente las dependencias necesarias (en este caso, una instancia de SaludoService) y las inyecta al momento de crear el objeto Aplicacion, gracias a la combinación de @Component y @Autowired.
> La inyección por constructor, como se muestra aquí, es la forma más recomendada por su claridad, simplicidad y compatibilidad con objetos inmutables. Además, facilita la escritura de tests unitarios al poder crear fácilmente instancias simuladas del servicio inyectado (mocks o stubs).
> El método ejecutar() demuestra cómo se puede usar la dependencia inyectada sin necesidad de preocuparse por su inicialización manual.

📊 Comparativa de tipos de inyección de dependencias en Spring

Tipo de Inyección	Ejemplo	Requiere `@Autowired`	Recomendado
Por Constructor	`public Clase(Servicio s) { ... }`	☑ sí	★★★★★
Por Campo (atributo)	`@Autowired Servicio servicio;`	☑ sí	★★
Por Setter (método)	`@Autowired void setServicio(Servicio s) { ... }`	☑ sí	★★★
Por Configuración Java	`@Bean public Servicio servicio() { return new Servicio(); }`	❌ no (`@Bean`)	★★★★

> 🔍 **Nota:** Aunque la inyección por campo es más concisa, oculta las dependencias reales y dificulta la prueba unitaria. La inyección por constructor es la más clara, testable y robusta, por eso es la opción recomendada en la mayoría de los casos.

Etapa 2: Aplicación REST básica con Spring Boot

Objetivo de la versión de servidor

Crear un proyecto Spring Boot con un controlador REST que provea un endpoint de estado y otro con una respuesta creativa personalizada.

Crear un nuevo proyecto con Spring Initializer

Dependencias seleccionadas:

- `Spring Web`: Para construir controladores REST con Spring.

- `Spring Boot DevTools`: Recarga automática durante el desarrollo (opcional).
- `Spring Boot Validation`: Validaciones automáticas con anotaciones (opcional).
- `Springdoc OpenAPI UI`: Generación automática de documentación Swagger/OpenAPI 3.

> 📦 **Nota:** Si no se encuentra `Springdoc OpenAPI UI` en el Initializr, se puede agregar manualmente al `pom.xml` luego.

Hacé clic en **GENERATE** para descargar el ZIP. Descomprimilo y abrilo con tu IDE (VS Code, IntelliJ, Eclipse, etc).

Agregar Swagger (Springdoc OpenAPI) si no lo agregaste al crear el proyecto

Si usaste Initializr sin Swagger, agregá la siguiente dependencia en `pom.xml`:

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.5.0</version>
</dependency>
```

El proyecto

📄 **Archivo src/main/resources/application.properties**

```
# Puerto explícito en el que se ejecuta el servidor
server.port=8080
```

> 🔍 Tip: Este archivo permite configurar no solo el puerto, sino también la conexión a la base de datos, el nombre de la aplicación, configuración de logs, tiempo de espera, y muchas otras propiedades del entorno Spring.

Controlador de ejemplo

```
// Imports necesarios
import org.springframework.beans.factory.annotation.Value;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api")
public class EstadoController {

    // Inyección del valor del puerto configurado en la aplicación
```

```
    @Value("${server.port}")
    private String puerto;

    // Endpoint de estado del servidor
    @GetMapping("/estado")
    public String estado() {
        return "Servidor funcionando correctamente en el puerto " + puerto + ".";
    }

    // Endpoint de saludo personalizado
    @GetMapping("/saludo")
    public String saludo(@RequestParam(defaultValue = "Anónimo") String nombre) {
        return "Hola " + nombre + ", que tengas un día excelente!";
    }
}
```

🔍 Explicación Mediante la anotación @Value("${server.port}"), Spring inyecta automáticamente el valor del puerto configurado en application.properties (o el valor por defecto 8080 si no hay ninguno especificado).

Esto permite que el endpoint /api/estado devuelva un mensaje como:

```
Servidor funcionando correctamente en el puerto 8080.
```

🔍 Tip: Podemos revisar aquí la idea de variables de configuración y cómo Spring puede inyectarlas dinámicamente desde archivos, argumentos o variables de entorno, lo cual será muy útil en etapas más avanzadas del curso.

Prueba

Al ejecutar la aplicación (mvn spring-boot:run o desde el IDE), accedé a:

- 👉 http://localhost:8080/swagger-ui.html

Allí verás una interfaz interactiva donde se puede probar los endpoints definidos en tus controladores sin usar Postman o curl.

🔍 Explicación Spring Boot permite construir APIs RESTful de manera sencilla gracias a su configuración por convención y sus dependencias integradas. Springdoc OpenAPI genera automáticamente documentación Swagger a partir de las anotaciones estándar del framework, sin necesidad de configuración adicional.
Esto permite:

- Probar la API de forma interactiva
- Generar documentación legible para otros equipos
- Exportar descripciones OpenAPI compatibles con herramientas externas (Postman, API Gateway, etc.)
- Swagger se vuelve esencial cuando tu backend expone múltiples endpoints, ya que mejora la comunicación, facilita el testeo y sirve como contrato vivo de la API.

Etapa 3: CRUD en Memoria usando Bean Singleton

Objetivo de una aplicación CRUD

Definir un modelo de entidad (en este caso, una Tarea), inicializar una lista en memoria, y exponer métodos CRUD via REST sin persistencia real. Se busca trabajar con distintos tipos de datos (texto, booleanos, fechas, enums) para ilustrar la riqueza de estructuras posibles.

Modelo de datos

```java
// Ubicación del archivo dentro del proyecto (ajustar según la estructura real)
package utnfc.isi.backend.spring.model;

// Anotaciones de Lombok para evitar escribir constructores y getters/setters
manualmente
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

// Tipo de dato para fechas sin hora
import java.time.LocalDate;

@Data // Genera automáticamente getters, setters, equals, hashCode y toString
@AllArgsConstructor // Genera constructor con todos los campos como parámetros
@NoArgsConstructor // Genera constructor vacío (sin parámetros)
public class Tarea {

    // Identificador único de la tarea
    private Long id;

    // Descripción de la tarea (detalle textual)
    private String descripcion;

    // Indica si la tarea fue completada o no
    private boolean completada;

    // Fecha límite para realizar la tarea
    private LocalDate fechaLimite;

    // Nivel de prioridad (enum interno)
    private Prioridad prioridad;

    // Enumeración que define los niveles posibles de prioridad
    public enum Prioridad {
        BAJA, MEDIA, ALTA
    }
}
```

Servicio Singleton con datos iniciales

```java
// Package base del proyecto (ajustar si cambia el módulo)
package utnfc.isi.backend.spring.service;

// Importa la anotación de Spring para declarar este Bean como un componente de servicio
import org.springframework.stereotype.Service;

// Importa la anotación para ejecutar un método automáticamente después de construir el Bean
import jakarta.annotation.PostConstruct;

// Tipos de datos utilizados en el modelo
import java.time.LocalDate;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.atomic.AtomicLong;
import java.util.*;

// Importación del modelo de dominio
import utnfc.isi.backend.spring.model.Tarea;

@Service // Marca esta clase como un servicio gestionado por el contenedor de Spring
public class TareaService {

    // Mapa en memoria que simula una base de datos (clave -> tarea)
    private Map<Long, Tarea> tareas = new ConcurrentHashMap<>();

    // Generador incremental de IDs único por tarea
    private AtomicLong secuencia = new AtomicLong(1);

    // Inicializa el servicio con un conjunto de tareas predefinidas al iniciar el contexto
    @PostConstruct
    public void init() {
        tareas.put(1L, new Tarea(1L, "Preparar presentación", false, LocalDate.now().plusDays(2), Tarea.Prioridad.ALTA));
        tareas.put(2L, new Tarea(2L, "Leer documentación", true, LocalDate.now().minusDays(1), Tarea.Prioridad.MEDIA));
        tareas.put(3L, new Tarea(3L, "Corregir ejercicios", false, LocalDate.now().plusDays(5), Tarea.Prioridad.BAJA));
        tareas.put(4L, new Tarea(4L, "Responder correos", false, LocalDate.now().plusDays(1), Tarea.Prioridad.MEDIA));
        tareas.put(5L, new Tarea(5L, "Actualizar aula virtual", true, LocalDate.now(), Tarea.Prioridad.ALTA));
    }

    // Retorna la lista de todas las tareas almacenadas
    public List<Tarea> obtenerTodas() {
        return new ArrayList<>(tareas.values());
    }

    // Retorna una tarea por su ID, o null si no existe
```

```
    public Tarea obtenerPorId(Long id) {
        return tareas.get(id);
    }

    // Agrega una nueva tarea, completando valores faltantes y generando ID
    public Tarea agregar(Tarea tarea) {
        if (tarea.getFechaLimite() == null) {
            tarea.setFechaLimite(LocalDate.now().plusDays(7));
        }
        if (tarea.getPrioridad() == null) {
            tarea.setPrioridad(Tarea.Prioridad.MEDIA);
        }
        long id = secuencia.getAndIncrement();
        tarea.setId(id);
        tareas.put(id, tarea);
        return tarea;
    }

    // Reemplaza completamente la tarea existente con el nuevo objeto
    public Tarea actualizar(Long id, Tarea nueva) {
        nueva.setId(id);
        tareas.put(id, nueva);
        return nueva;
    }

    // Elimina la tarea correspondiente al ID
    public void eliminar(Long id) {
        tareas.remove(id);
    }
}
```

Controlador REST CRUD

```
@RestController
@RequestMapping("/api/tareas")
public class TareaController {
    @Autowired
    private TareaService servicio;

    @GetMapping
    public List<Tarea> listar() { return servicio.obtenerTodas(); }

    @GetMapping("/{id}")
    public Tarea obtener(@PathVariable Long id) { return
servicio.obtenerPorId(id); }

    @PostMapping
    public Tarea crear(@RequestBody Tarea tarea) { return servicio.agregar(tarea);
}
```

```
    @PutMapping("/{id}")
    public Tarea actualizar(@PathVariable Long id, @RequestBody Tarea tarea) {
        return servicio.actualizar(id, tarea);
    }

    @DeleteMapping("/{id}")
    public void eliminar(@PathVariable Long id) { servicio.eliminar(id); }
}
```

Objetivo

Definir un modelo de entidad, inicializar una lista en memoria, y exponer métodos CRUD via REST sin persistencia real.

Modelo de datos básico

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Persona {
    private Long id;
    private String nombre;
    private int edad;
}
```

Servicio Singleton con datos iniciales mock

```
@Service
public class PersonaService {
    private Map<Long, Persona> personas = new ConcurrentHashMap<>();
    private AtomicLong secuencia = new AtomicLong(1);

    @PostConstruct
    public void init() {
        personas.put(1L, new Persona(1L, "Ana", 30));
        personas.put(2L, new Persona(2L, "Juan", 25));
        personas.put(3L, new Persona(3L, "Lucia", 35));
        personas.put(4L, new Persona(4L, "Pedro", 28));
        personas.put(5L, new Persona(5L, "Sofia", 22));
    }

    public List<Persona> obtenerTodas() { return new ArrayList<>
(personas.values()); }
    public Persona obtenerPorId(Long id) { return personas.get(id); }
    public Persona agregar(Persona persona) {
        long id = secuencia.getAndIncrement();
        persona.setId(id);
        personas.put(id, persona);
        return persona;
```

```
    }
    public Persona actualizar(Long id, Persona nueva) {
        nueva.setId(id);
        personas.put(id, nueva);
        return nueva;
    }
    public void eliminar(Long id) { personas.remove(id); }
}
```

> 🔍 Explicación Este servicio simula una capa de persistencia en memoria, ideal para los primeros pasos en aplicaciones REST con Spring, sin necesidad de base de datos.
> @Service indica que esta clase forma parte de la capa de lógica de negocio, separándola de los controladores y los modelos.
> @PostConstruct ejecuta el método init() automáticamente cuando Spring termina de crear el bean, precargando la colección con tareas de ejemplo.
> ConcurrentHashMap y AtomicLong aseguran que la estructura sea segura para acceso concurrente (aunque no es esencial en entornos de desarrollo, es buena práctica).
> El método agregar() incorpora lógica de negocio: si la tarea no tiene fecha límite ni prioridad, se asignan valores por defecto (esto sería el equivalente a una validación/preparación de dominio antes de persistir).
> actualizar() y eliminar() actúan sobre el mapa directamente, simulando un acceso a base real.

Controlador REST CRUD en Memoria

```
// Package base del proyecto (ajustar según tu estructura real)
package utnfc.isi.backend.spring.controller;

// Anotaciones de Spring para controladores REST y mapeo de rutas
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

// Tipos utilizados
import java.util.List;

// Modelo y servicio asociado
import utnfc.isi.backend.spring.model.Persona;
import utnfc.isi.backend.spring.service.PersonaService;

@RestController // Marca esta clase como controlador REST que retorna directamente
datos en la respuesta
@RequestMapping("/api/personas") // Define el prefijo común para todos los
endpoints de este controlador
public class PersonaController {

    // Inyección automática del servicio asociado
    @Autowired
    private PersonaService servicio;

    // Endpoint GET /api/personas - devuelve la lista completa de personas
    @GetMapping
```

```
    public List<Persona> listar() {
        return servicio.obtenerTodas();
    }

    // Endpoint GET /api/personas/{id} - busca una persona por ID
    @GetMapping("/{id}")
    public Persona obtener(@PathVariable Long id) {
        return servicio.obtenerPorId(id);
    }

    // Endpoint POST /api/personas - crea una nueva persona
    @PostMapping
    public Persona crear(@RequestBody Persona persona) {
        return servicio.agregar(persona);
    }

    // Endpoint PUT /api/personas/{id} - actualiza una persona existente
    @PutMapping("/{id}")
    public Persona actualizar(@PathVariable Long id, @RequestBody Persona persona)
{
        return servicio.actualizar(id, persona);
    }

    // Endpoint DELETE /api/personas/{id} - elimina una persona por su ID
    @DeleteMapping("/{id}")
    public void eliminar(@PathVariable Long id) {
        servicio.eliminar(id);
    }
}
```

🔍 Explicación Este controlador es un ejemplo típico de cómo exponer una API REST CRUD en Spring Boot:

- Usa @RestController para que Spring serialice automáticamente las respuestas en JSON.
- Los endpoints siguen las convenciones RESTful:
  - GET /api/personas → listar todas
  - GET /api/personas/{id} → obtener por ID
  - POST → crear
  - PUT → actualizar
  - DELETE → eliminar

Spring realiza el mapeo automático de JSON a objetos Java en los métodos que reciben @RequestBody, y resuelve variables en la URL con @PathVariable.

🧪 Pruebas y Swagger

Si está agregada la dependencia de Swagger con springdoc-openapi-ui, los endpoints definidos en este controlador se documentan automáticamente y se pueden probar desde:

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

Ahí se generan formularios interactivos para enviar GET, POST, PUT y DELETE sin necesidad de Postman.

También se puede probar con archivos .rest desde VS Code usando REST Client:

```
### Crear persona
POST http://localhost:8080/api/personas
Content-Type: application/json

{
  "nombre": "Juan",
  "edad": 32
}

### Obtener persona
GET http://localhost:8080/api/personas/1

### Actualizar persona
PUT http://localhost:8080/api/personas/1
Content-Type: application/json

{
  "nombre": "Juan Actualizado",
  "edad": 33
}

### Borrar persona
DELETE http://localhost:8080/api/personas/1
```

Etapa 4: CRUD con Spring Data JPA y H2

Objetivo de la versión JPA

Persistir las entidades en una base de datos embebida H2 usando Spring Data JPA.

Configuración e inicialización de la BD

Agregar dependencias en `pom.xml`

```
<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>
```

application.properties

```
# Base de datos en memoria con nombre específico
spring.datasource.url=jdbc:h2:mem:tareasdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=

# Habilitar consola web de H2
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console

# Inicialización automática de estructura y datos
spring.jpa.hibernate.ddl-auto=create
```

Entidad JPA

Cómo vamos a utilizar validaciones en los atributos de la entidad primero debemos agregar la dependencia a `spring validation`.

Para esto si no lo agregamos previamente, debemos agregar lo siguiente al archivo `pom.xml`

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

Clase de Entidad JPA

Ahora sí vamos con el archivo de Entidad

```
package utnfc.isi.backend.spring.model;

// Anotaciones de persistencia y validación
import jakarta.persistence.*;
import jakarta.validation.constraints.*;

// Anotaciones de Lombok para evitar boilerplate
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.AllArgsConstructor;
```

```java
// Tipo de dato para fechas sin hora
import java.time.LocalDate;

@Entity // Marca esta clase como una entidad JPA persistente
@Data // Genera automáticamente getters, setters, toString, equals y hashCode
@NoArgsConstructor // Constructor vacío requerido por JPA
@AllArgsConstructor // Constructor con todos los campos para uso general
public class Tarea {

    @Id // Identificador primario de la entidad
    @GeneratedValue(strategy = GenerationType.IDENTITY) // Autogenerado por la base de datos
    private Long id;

    @NotBlank(message = "La descripción no puede estar vacía") // No puede ser nula ni cadena vacía
    @Size(min = 5, message = "La descripción debe tener al menos 5 caracteres") // Requiere longitud mínima
    @Column(name = "descripcion") // Mapeo explícito del campo a la columna
    private String descripcion;

    @Column(name = "completada") // Bandera de tarea completada o pendiente
    private boolean completada;

    @Column(name = "fecha_limite") // Fecha de vencimiento de la tarea
    @Future(message = "La fecha límite debe ser futura") // No puede ser pasada
    private LocalDate fechaLimite;

    @NotNull(message = "La prioridad es obligatoria") // No se permite omitir prioridad
    @Enumerated(EnumType.STRING) // Se almacena como texto ('BAJA', 'MEDIA', 'ALTA')
    @Column(name = "prioridad")
    private Prioridad prioridad;

    // Enumeración interna para los niveles de prioridad
    public enum Prioridad {
        BAJA, MEDIA, ALTA
    }
}
```

🔍 Explicación

Esta clase es el modelo Tarea, ahora adaptado a una entidad JPA que puede ser persistida en base de datos.

🚀 Anotaciones clave

- @Entity: habilita que Spring Data JPA pueda mapear esta clase a una tabla en la base de datos.
- @Id + @GeneratedValue: indica que el campo id es la clave primaria y que su valor será generado automáticamente por la base.

> - @Enumerated(EnumType.STRING): guarda el valor textual del enum (por ejemplo 'ALTA') en lugar de su ordinal (2), lo cual mejora la legibilidad y estabilidad de los datos.
> - Lombok (@Data, @NoArgsConstructor, @AllArgsConstructor) ayuda a evitar código repetitivo, pero el constructor vacío es obligatorio para JPA.

Archivo `data.sql`

```
INSERT INTO tarea (descripcion, completada, fecha_limite, prioridad) VALUES
('Preparar presentación final', FALSE, '2025-06-12', 'ALTA'),
('Corregir evaluaciones', FALSE, '2025-06-08', 'MEDIA'),
('Revisar documentación de Swagger', TRUE, '2025-06-05', 'BAJA'),
('Actualizar sitio institucional', FALSE, '2025-06-14', 'ALTA'),
('Leer artículo técnico', TRUE, '2025-06-01', 'BAJA'),
('Subir notas al sistema', FALSE, '2025-06-10', 'MEDIA'),
('Preparar práctica de SQL', FALSE, '2025-06-15', 'ALTA'),
('Reunión de coordinación', TRUE, '2025-06-03', 'MEDIA'),
('Documentar endpoint REST', FALSE, '2025-06-13', 'BAJA'),
('Enviar informe final', FALSE, '2025-06-11', 'ALTA');
```

> ⚠ Asegurarse de que el modelo `Tarea` esté anotado con `@Entity` y tenga los nombres de columna coincidentes con `descripcion`, `completada`, `fecha_limite`, `prioridad` (ver anotaciones JPA si se personalizan).

Estructura esperada en resources

```
src/main/resources/
├── application.properties
└── data.sql
```

Probar el entorno con H2 Console

```
http://localhost:8080/h2-console
```

Datos de conexión:

- JDBC URL: `jdbc:h2:mem:tareasdb`
- User Name: `sa`
- Password: *(vacía)*

Una vez adentro, ejecutar:

```
SELECT * FROM tarea;
```

para visualizar las 10 tareas precargadas.

> 🔍 Este entorno inicial es ideal para prácticas, validación de datos, y revisión de la interacción entre capa de persistencia y exposición de servicios REST.

Repositorio

```
package utnfc.isi.backend.spring.repository;

// Importaciones necesarias para JpaRepository y queries personalizadas
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;

import utnfc.isi.backend.spring.model.Tarea;

import java.time.LocalDate;
import java.util.List;

@Repository // (opcional, pero explícito y claro)
public interface TareaRepository extends JpaRepository<Tarea, Long> {

    // Consulta personalizada con JPQL para obtener tareas vencidas (fecha pasada
y no completadas)
    @Query("SELECT t FROM Tarea t WHERE t.fechaLimite < :fecha AND t.completada =
false")
    List<Tarea> tareasVencidas(@Param("fecha") LocalDate fecha);
}
```

> 🔍 Explicación
> Este repositorio extiende JpaRepository<Tarea, Long>, lo cual habilita:
> Acceso automático a todos los métodos CRUD y más, sin escribir código.
> Creación de queries automáticas basadas en nombres de métodos.
> Definición de consultas personalizadas con @Query usando el lenguaje JPQL.

🧠 Sobre la consulta @Query:
La anotación permite definir una consulta JPQL como si estuvieras escribiendo código sobre entidades, no sobre tablas. En este caso:

```
SELECT t FROM Tarea t
WHERE t.fechaLimite < :fecha
AND t.completada = false
```

Filtra todas las tareas que están vencidas y aún no fueron completadas.

📊 **Métodos disponibles por herencia de `JpaRepository<T, ID>`**

Método	Descripción
`findAll()`	Retorna todas las entidades almacenadas.
`findById(ID id)`	Busca una entidad por su identificador.
`save(T entity)`	Inserta o actualiza una entidad.
`saveAll(Iterable<T> entities)`	Inserta o actualiza una colección de entidades.
`delete(T entity)`	Elimina una entidad específica.
`deleteById(ID id)`	Elimina una entidad a partir de su identificador.
`existsById(ID id)`	Verifica si existe una entidad con ese identificador.
`count()`	Devuelve la cantidad total de entidades.
`findAllById(Iterable<ID> ids)`	Retorna todas las entidades cuyos IDs se encuentren en la lista.
`flush()`	Sincroniza las operaciones pendientes con la base de datos.
`getReferenceById(ID id)`	Retorna una referencia perezosa (proxy) a una entidad.

> 🔍 Además de estos métodos, se puede crear consultas automáticas con nombres derivados como `findByDescripcionContaining`, `findByPrioridadAndCompletadaFalse`, entre muchos otros, sin necesidad de escribir `@Query`.

Servicio TareaService JPA

```java
package utnfc.isi.backend.spring.service;

// Anotaciones de Spring
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

// Tipos de colección y fechas
import java.time.LocalDate;
import java.util.List;

// Modelo y repositorio
import utnfc.isi.backend.spring.model.Tarea;
import utnfc.isi.backend.spring.repository.TareaRepository;

@Service // Indica que esta clase forma parte de la capa de lógica de negocio
public class TareaService {

    @Autowired // Inyecta automáticamente el repositorio correspondiente
    private TareaRepository repository;

    // Devuelve todas las tareas almacenadas
    public List<Tarea> obtenerTodas() {
        return repository.findAll();
    }
```

```
    // Busca una tarea por su ID o retorna null si no existe
    public Tarea obtenerPorId(Long id) {
        return repository.findById(id).orElse(null);
    }

    // Agrega una nueva tarea, aplicando lógica de negocio previa
    public Tarea crear(Tarea tarea) {
        if (tarea.getFechaLimite() == null) {
            tarea.setFechaLimite(LocalDate.now().plusDays(7)); // Por defecto, 1
semana
        }
        if (tarea.getPrioridad() == null) {
            tarea.setPrioridad(Tarea.Prioridad.MEDIA); // Por defecto, prioridad
media
        }
        return repository.save(tarea);
    }

    // Actualiza una tarea existente (reemplazo total)
    public Tarea actualizar(Long id, Tarea tarea) {
        tarea.setId(id); // Asegura que el ID sea el mismo que el que se intenta
actualizar
        return repository.save(tarea); // save actúa como insert/update según el
contexto
    }

    // Elimina una tarea por ID
    public void eliminar(Long id) {
        repository.deleteById(id);
    }

    // Consulta tareas vencidas usando el método definido en el repositorio
    public List<Tarea> tareasVencidas() {
        return repository.tareasVencidas(LocalDate.now());
    }
}
```

🔍 Explicación del Servicio `TareaService`

🧱 Rol del servicio

Esta clase representa la **capa de servicio** en una arquitectura típica de aplicaciones Spring Boot. Su propósito principal es:

- Contener la lógica de negocio de la aplicación.
- Servir de intermediario entre los controladores REST y la capa de persistencia (`TareaRepository`).
- Centralizar reglas, cálculos o validaciones que no deben repetirse ni en el controlador ni en el repositorio.

🛠️ Lógica de negocio incorporada

- En el método `crear(...)`, si no se especifica una fecha límite o una prioridad, se completan con valores por defecto:
  - `fechaLimite`: se asigna la fecha actual + 7 días.
  - `prioridad`: se asigna `MEDIA`.
- En `actualizar(...)`, se asegura que el ID sea el mismo que el de la tarea a modificar, para evitar que `save(...)` cree un nuevo registro.
- En `tareasVencidas()`, se utiliza el método personalizado del repositorio con `@Query` para recuperar las tareas que están vencidas y no fueron completadas.

☑ **Beneficios de esta organización**

- Evita la duplicación de lógica en múltiples controladores.
- Permite testear la lógica de negocio de forma aislada.
- Mejora la legibilidad y separación de responsabilidades.
- Facilita el mantenimiento y escalabilidad del sistema.

Controlador REST

```
package utnfc.isi.backend.spring.controller;

// Anotaciones de Spring para controladores REST y mapeo de rutas
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;

import utnfc.isi.backend.spring.model.Tarea;
import utnfc.isi.backend.spring.service.TareaService;

@RestController // Indica que esta clase expone endpoints REST
@RequestMapping("/api/tareas") // Prefijo común para todas las rutas de este
recurso
public class TareaController {

    @Autowired // Inyección del servicio de lógica de negocio
    private TareaService servicio;

    // GET /api/tareas → lista todas las tareas
    @GetMapping
    public List<Tarea> listar() {
        return servicio.obtenerTodas();
    }

    // GET /api/tareas/{id} → busca una tarea por ID
    @GetMapping("/{id}")
    public Tarea obtener(@PathVariable Long id) {
        return servicio.obtenerPorId(id);
    }

    // POST /api/tareas → crea una nueva tarea
```

```
    @PostMapping
    public Tarea crear(@RequestBody Tarea tarea) {
        return servicio.crear(tarea);
    }

    // PUT /api/tareas/{id} → actualiza una tarea existente
    @PutMapping("/{id}")
    public Tarea actualizar(@PathVariable Long id, @RequestBody Tarea tarea) {
        return servicio.actualizar(id, tarea);
    }

    // DELETE /api/tareas/{id} → elimina una tarea por ID
    @DeleteMapping("/{id}")
    public void eliminar(@PathVariable Long id) {
        servicio.eliminar(id);
    }

    // GET /api/tareas/vencidas → retorna las tareas vencidas
    @GetMapping("/vencidas")
    public List<Tarea> vencidas() {
        return servicio.tareasVencidas();
    }
}
```

🔍 Explicación final

Este controlador REST representa la puerta de entrada al backend desde clientes externos, como el navegador, Postman o una SPA hecha con React.

☑ Convenciones REST aplicadas Cada método se mapea a una ruta y verbo HTTP estándar (GET, POST, PUT, DELETE).

- Se utiliza @RequestBody para recibir objetos JSON.
- Se usa @PathVariable para capturar IDs desde la URL.
- El prefijo /api/tareas ayuda a organizar los recursos REST por entidad.

💡 Ventajas en buenas prácticas

- Este controlador es un ejemplo perfecto de responsabilidad simple: sólo orquesta la petición y delega al servicio.
- Ayuda a entender cómo estructurar una API REST sin contaminarla con lógica interna.
- Es fácil de testear usando herramientas como Swagger UI, REST Client o Postman.

Etapa 5: Colección REST Client para VS Code

Archivo `tareas.rest`

```
### Estado del servidor
# Verifica que el backend esté activo
GET http://localhost:8080/api/estado
```

```
### Saludo personalizado
# Devuelve un mensaje con el nombre provisto por parámetro
GET http://localhost:8080/api/saludo?nombre=Felipe

### Listar tareas
# Recupera todas las tareas existentes en la base
GET http://localhost:8080/api/tareas

### Crear tarea
# Agrega una nueva tarea con todos los campos definidos
POST http://localhost:8080/api/tareas
Content-Type: application/json

{
  "descripcion": "Entregar informe final",
  "completada": false,
  "fechaLimite": "2025-06-10",
  "prioridad": "ALTA"
}

### Crear tarea sin fecha ni prioridad (usa lógica de negocio)
# El servicio completará fecha límite y prioridad por defecto
POST http://localhost:8080/api/tareas
Content-Type: application/json

{
  "descripcion": "Revisar documentación",
  "completada": false
}

### Obtener una tarea por ID
# Busca la tarea cuyo ID sea 1
GET http://localhost:8080/api/tareas/1

### Actualizar tarea
# Reemplaza todos los campos de la tarea con ID 1
PUT http://localhost:8080/api/tareas/1
Content-Type: application/json

{
  "descripcion": "Entregar informe actualizado",
  "completada": true,
  "fechaLimite": "2025-06-12",
  "prioridad": "MEDIA"
}

### Eliminar tarea
# Borra la tarea con ID 1
DELETE http://localhost:8080/api/tareas/1

### Obtener tareas vencidas
# Lista tareas que están vencidas y aún no han sido completadas
```

```
GET http://localhost:8080/api/tareas/vencidas
```

Notas finales

🔄 Evolución de la app:
[✔] Consola con Beans
[✔] API REST básica
[✔] CRUD en memoria
[✔] CRUD con base H2 y JPA
[✔] Documentación Swagger
[✔] Cliente REST con VS Code