**Tutorial de Spring Boot con H2: Sistema de Gestión de Nutrición**

Este tutorial completo te guiará paso a paso en la creación de una aplicación de gestión nutricional utilizando Spring Boot y la base de datos H2. Aprenderás conceptos fundamentales como la configuración de base de datos, modelado de entidades JPA, estrategias de carga (EAGER vs LAZY), patrones de inyección de dependencias y desarrollo de APIs REST.

**Índice**

1. [Introducción y Configuración del Proyecto](https://file+.vscode-resource.vscode-cdn.net/Users/sebgomez/Desktop/nutrition/tutorial-perfect.md#1-introducci%C3%B3n-y-configuraci%C3%B3n-del-proyecto)
2. [Configuración de H2 y Gestión de Entornos](https://file+.vscode-resource.vscode-cdn.net/Users/sebgomez/Desktop/nutrition/tutorial-perfect.md#2-configuraci%C3%B3n-de-h2-y-gesti%C3%B3n-de-entornos)
3. [Modelado de Entidades con Anotaciones JPA](https://file+.vscode-resource.vscode-cdn.net/Users/sebgomez/Desktop/nutrition/tutorial-perfect.md#3-modelado-de-entidades-con-anotaciones-jpa)
4. [Estrategias de Carga: EAGER vs LAZY](https://file+.vscode-resource.vscode-cdn.net/Users/sebgomez/Desktop/nutrition/tutorial-perfect.md#4-estrategias-de-carga-eager-vs-lazy)
5. [Patrones de Inyección de Dependencias](https://file+.vscode-resource.vscode-cdn.net/Users/sebgomez/Desktop/nutrition/tutorial-perfect.md#5-patrones-de-inyecci%C3%B3n-de-dependencias)
6. [Desarrollo de APIs REST](https://file+.vscode-resource.vscode-cdn.net/Users/sebgomez/Desktop/nutrition/tutorial-perfect.md#6-desarrollo-de-apis-rest)
7. [Buenas Prácticas y Solución de Problemas Comunes](https://file+.vscode-resource.vscode-cdn.net/Users/sebgomez/Desktop/nutrition/tutorial-perfect.md#7-buenas-pr%C3%A1cticas-y-soluci%C3%B3n-de-problemas-comunes)
8. [Ejemplos Completos de Código](https://file+.vscode-resource.vscode-cdn.net/Users/sebgomez/Desktop/nutrition/tutorial-perfect.md#8-ejemplos-completos-de-c%C3%B3digo)

**1. Introducción y Configuración del Proyecto**

**1.1 Requisitos Previos**

Para seguir este tutorial necesitarás:

* Java 17+ instalado
* Maven
* Tu IDE favorito (IntelliJ IDEA, Eclipse, VS Code)

**1.2 Creación del Proyecto**

Crea un nuevo proyecto Spring Boot utilizando Spring Initializr (<https://start.spring.io/>) con las siguientes dependencias:

* Spring Web
* Spring Data JPA
* H2 Database
* Spring DevTools (opcional)

**1.3 Estructura del Proyecto**

nutrition/

├── src/

│ ├── main/

│ │ ├── java/

│ │ │ └── com/

│ │ │ └── eafit/

│ │ │ └── nutrition/

│ │ │ ├── NutritionApplication.java

│ │ │ ├── model/

│ │ │ ├── repository/

│ │ │ ├── service/

│ │ │ └── controller/

│ │ └── resources/

│ │ ├── application.properties

│ │ ├── application-dev.properties

│ │ ├── application-staging.properties

│ │ └── data.sql

│ └── test/

└── pom.xml

**2. Configuración de H2 y Gestión de Entornos**

H2 es una base de datos en memoria muy útil para desarrollo y pruebas. Spring Boot la integra fácilmente y ofrece varias formas de configuración.

**2.1 Modos de Operación de H2**

H2 puede funcionar en varios modos:

1. **Modo en memoria**: Los datos se almacenan en memoria y se pierden al reiniciar la aplicación
2. **Modo basado en archivo**: Los datos persisten en un archivo después de reiniciar
3. **Modo servidor**: H2 funciona como un servidor de base de datos independiente
4. **Modo embebido**: Integrado en tu aplicación

**2.2 Configuración de Perfiles para Diferentes Entornos**

Es una buena práctica separar las configuraciones por entorno. Vamos a crear tres perfiles:

**application.properties (Principal)**

spring.application.name=nutrition

spring.profiles.active=dev

**application-dev.properties (Desarrollo con H2 en memoria)**

# Configuración de base de datos H2 en memoria

spring.datasource.url=jdbc:h2:mem:nutrition

spring.datasource.username=sa

spring.datasource.password=

# Configuración de Hibernate

spring.jpa.hibernate.ddl-auto=create-drop

spring.jpa.show-sql=true

spring.jpa.properties.hibernate.format\_sql=true

# Configuración de H2 Console

spring.h2.console.enabled=true

spring.h2.console.path=/h2-console

# Logs detallados de SQL

logging.level.org.hibernate.SQL=DEBUG

logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE

**application-staging.properties (Emulando PostgreSQL con H2 basado en archivo)**

# Configuración de H2 basado en archivo emulando PostgreSQL

spring.datasource.url=jdbc:h2:file:./src/main/resources/db/nutrition\_staging;MODE=PostgreSQL;DATABASE\_TO\_LOWER=TRUE;DB\_CLOSE\_DELAY=-1;DB\_CLOSE\_ON\_EXIT=FALSE

spring.datasource.username=sa

spring.datasource.password=

# Configuración de Hibernate para entorno de staging

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

# Configuración de H2 Console

spring.h2.console.enabled=true

spring.h2.console.path=/h2-console

**2.3 Accediendo a la Consola H2**

La consola H2 es una herramienta web para administrar tu base de datos:

1. Inicia la aplicación
2. Abre tu navegador en http://localhost:8080/h2-console
3. Configura los detalles de conexión:
   * JDBC URL: jdbc:h2:mem:nutrition (para dev) o jdbc:h2:file:./src/main/resources/db/nutrition\_staging (para staging)
   * Usuario: sa
   * Contraseña: (vacía)

**2.4 Selección de Perfil Activo**

Para cambiar el perfil activo, puedes:

Con Maven: ./mvnw spring-boot:run -Dspring-boot.run.profiles=staging

**3. Modelado de Entidades con Anotaciones JPA**

Nuestro sistema gestionará nutricionistas, pacientes, mediciones antropométricas y notas clínicas.

**3.1 Anotaciones Básicas de JPA**

* @Entity: Marca una clase como una entidad persistente
* @Table: Personaliza el nombre de la tabla y otras propiedades
* @Id: Define la clave primaria
* @GeneratedValue: Especifica la estrategia de generación de la clave primaria
* @Column: Personaliza la columna (nombre, restricciones, etc.)

**3.2 Entidad Nutricionista**

package com.eafit.nutrition.model;

import jakarta.persistence.\*;

import java.util.ArrayList;

import java.util.List;

@Entity

@Table(name = "nutricionista")

public class Nutricionista {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@Column(name = "nombre", nullable = false, length = 100)

private String nombre;

@Column(name = "apellido", nullable = false, length = 100)

private String apellido;

@Column(name = "numero\_licencia", nullable = false, length = 50, unique = true)

private String numeroLicencia;

@Column(name = "especialidad", length = 100)

private String especialidad;

@Column(name = "email", nullable = false, length = 150, unique = true)

private String email;

@Column(name = "telefono", length = 20)

private String telefono;

@Column(name = "activo", nullable = false)

private boolean activo = true;

// Relaciones (se definen más adelante)

// Constructor vacío requerido por JPA

public Nutricionista() {

}

// Constructor con parámetros principales

public Nutricionista(String nombre, String apellido, String numeroLicencia, String email) {

this.nombre = nombre;

this.apellido = apellido;

this.numeroLicencia = numeroLicencia;

this.email = email;

}

// Getters y setters (omitidos por brevedad)

}

**3.3 Entidad Paciente**

package com.eafit.nutrition.model;

import jakarta.persistence.\*;

import java.time.LocalDate;

@Entity

@Table(name = "paciente")

public class Paciente {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@Column(name = "nombre", nullable = false, length = 100)

private String nombre;

@Column(name = "apellido", nullable = false, length = 100)

private String apellido;

@Column(name = "fecha\_nacimiento")

private LocalDate fechaNacimiento;

@Column(name = "email", nullable = false, length = 150, unique = true)

private String email;

@Column(name = "telefono", length = 20)

private String telefono;

@Column(name = "activo", nullable = false)

private boolean activo = true;

// Relación con Nutricionista (se define más adelante)

// Constructores, getters y setters (omitidos por brevedad)

}

**3.4 Anotaciones para Relaciones**

Para modelar relaciones entre entidades, JPA proporciona varias anotaciones:

* @OneToMany: Relación uno-a-muchos
* @ManyToOne: Relación muchos-a-uno
* @OneToOne: Relación uno-a-uno
* @ManyToMany: Relación muchos-a-muchos
* @JoinColumn: Especifica la columna de clave foránea
* @JoinTable: Define una tabla de unión (para relaciones @ManyToMany)

**3.5 Configuración de Relaciones entre Entidades**

Vamos a establecer las relaciones entre nuestras entidades:

**Relación Nutricionista-Paciente (OneToMany/ManyToOne)**

En la clase Nutricionista:

// Relación LAZY (carga perezosa) con Pacientes

@OneToMany(mappedBy = "nutricionista", cascade = CascadeType.ALL, fetch = FetchType.LAZY)

private List<Paciente> pacientes = new ArrayList<>();

// Método helper para añadir pacientes

public void addPaciente(Paciente paciente) {

pacientes.add(paciente);

paciente.setNutricionista(this);

}

En la clase Paciente:

// Relación EAGER (carga inmediata) con Nutricionista

@ManyToOne(fetch = FetchType.EAGER)

@JoinColumn(name = "nutricionista\_id", nullable = false)

private Nutricionista nutricionista;

**4. Estrategias de Carga: EAGER vs LAZY**

Una de las decisiones más importantes en JPA es cómo configurar la carga de relaciones.

**4.1 Estrategias de Carga en JPA**

* **EAGER**: Las relaciones se cargan inmediatamente junto con la entidad
* **LAZY**: Las relaciones se cargan solo cuando se accede a ellas explícitamente

**4.2 Valores Predeterminados**

* @OneToMany y @ManyToMany: LAZY por defecto
* @OneToOne y @ManyToOne: EAGER por defecto

**4.3 Entidades con Diferentes Estrategias de Carga**

Vamos a implementar un ejemplo completo con la entidad Nota para mostrar ambas estrategias:

package com.eafit.nutrition.model;

import jakarta.persistence.\*;

import java.time.LocalDateTime;

@Entity

@Table(name = "nota")

public class Nota {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@Column(name = "titulo", nullable = false, length = 255)

private String titulo;

@Column(name = "contenido", nullable = false, columnDefinition = "TEXT")

private String contenido;

@Column(name = "fecha\_creacion", nullable = false)

private LocalDateTime fechaCreacion;

@Column(name = "tipo\_nota", length = 50)

private String tipoNota;

// Relación EAGER con Paciente

@ManyToOne(fetch = FetchType.EAGER)

@JoinColumn(name = "paciente\_id", nullable = false)

private Paciente paciente;

// Relación LAZY con Nutricionista

@ManyToOne(fetch = FetchType.LAZY)

@JoinColumn(name = "nutricionista\_id", nullable = false)

private Nutricionista nutricionista;

// Constructores, getters y setters (omitidos por brevedad)

}

Ahora actualizamos la clase Paciente para incluir sus notas:

// Relación LAZY con las Notas

@OneToMany(mappedBy = "paciente", cascade = CascadeType.ALL, fetch = FetchType.LAZY)

private List<Nota> notas = new ArrayList<>();

// Método helper para añadir notas

public void addNota(Nota nota) {

notas.add(nota);

nota.setPaciente(this);

}

Y la clase Nutricionista:

// Relación EAGER (carga inmediata) con Notas

@OneToMany(mappedBy = "nutricionista", cascade = CascadeType.ALL, fetch = FetchType.EAGER)

private List<Nota> notas = new ArrayList<>();

// Método helper para añadir notas

public void addNota(Nota nota) {

notas.add(nota);

nota.setNutricionista(this);

}

**4.4 El Problema N+1 y Cómo Evitarlo**

El problema N+1 ocurre cuando JPA ejecuta una consulta para cargar entidades principales y luego N consultas adicionales para cargar sus relaciones:

1. Una consulta para obtener las entidades principales (por ejemplo, nutricionistas)
2. N consultas para cargar las relaciones de cada entidad (pacientes de cada nutricionista)

Para visualizar este problema, habilitamos la configuración de logs en application-dev.properties:

logging.level.org.hibernate.SQL=DEBUG

logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE

Soluciones al problema N+1:

1. **Usar FetchType.EAGER** (no recomendado para colecciones grandes)
2. **JOIN FETCH en consultas JPQL**:

@Query("SELECT n FROM Nutricionista n LEFT JOIN FETCH n.pacientes WHERE n.id = :id")

Optional<Nutricionista> findByIdWithPacientes(@Param("id") Long id);

1. **EntityGraph**:

@EntityGraph(attributePaths = {"pacientes"})

@Query("SELECT n FROM Nutricionista n WHERE n.id = :id")

Optional<Nutricionista> findByIdWithPacientesGraph(@Param("id") Long id);

**4.5 Comparación de Consultas SQL Generadas**

Con estrategia LAZY y sin optimización (problema N+1):

-- Consulta inicial

SELECT \* FROM nutricionista WHERE id = 1;

-- Consulta para cada paciente (N consultas adicionales)

SELECT \* FROM paciente WHERE nutricionista\_id = 1;

Con JOIN FETCH:

-- Una sola consulta (eficiente)

SELECT n.\*, p.\* FROM nutricionista n

LEFT JOIN paciente p ON n.id = p.nutricionista\_id

WHERE n.id = 1;

**5. Patrones de Inyección de Dependencias**

Spring Framework proporciona varios métodos para la inyección de dependencias.

**5.1 ¿Qué es la Inyección de Dependencias?**

La inyección de dependencias (DI) es un patrón de diseño en el que una clase recibe sus dependencias de fuentes externas en lugar de crearlas internamente. Esto promueve el bajo acoplamiento y facilita las pruebas.

**5.2 Tres Métodos de Inyección de Dependencias**

Vamos a implementar una nueva entidad Medicion para medidas antropométricas y mostrar los tres métodos de inyección:

package com.eafit.nutrition.model;

import jakarta.persistence.\*;

import java.time.LocalDate;

@Entity

@Table(name = "medicion")

public class Medicion {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@Column(name = "fecha", nullable = false)

private LocalDate fecha;

@Column(name = "peso", nullable = false)

private Double peso; // en kg

@Column(name = "altura", nullable = false)

private Double altura; // en cm

@Column(name = "circunferencia\_cintura")

private Double circunferenciaCintura; // en cm

@Column(name = "circunferencia\_cadera")

private Double circunferenciaCadera; // en cm

@Column(name = "porcentaje\_grasa\_corporal")

private Double porcentajeGrasaCorporal;

@ManyToOne(fetch = FetchType.EAGER)

@JoinColumn(name = "paciente\_id", nullable = false)

private Paciente paciente;

@ManyToOne(fetch = FetchType.EAGER)

@JoinColumn(name = "nutricionista\_id", nullable = false)

private Nutricionista nutricionista;

// Método para calcular el IMC

public Double calcularIMC() {

if (altura == null || peso == null || altura <= 0) {

return null;

}

double alturaEnMetros = altura / 100.0;

return peso / (alturaEnMetros \* alturaEnMetros);

}

// Constructores, getters y setters (omitidos por brevedad)

}

Repositorio para Medicion:

package com.eafit.nutrition.repository;

import com.eafit.nutrition.model.Medicion;

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

import org.springframework.stereotype.Repository;

import java.util.List;

import java.util.Optional;

@Repository

public interface MedicionRepository extends JpaRepository<Medicion, Long> {

List<Medicion> findByPacienteIdOrderByFechaDesc(Long pacienteId);

Optional<Medicion> findFirstByPacienteIdOrderByFechaDesc(Long pacienteId);

}

**5.2.1 Inyección por Constructor**

package com.eafit.nutrition.service;

import com.eafit.nutrition.model.Medicion;

import com.eafit.nutrition.repository.MedicionRepository;

import com.eafit.nutrition.repository.PacienteRepository;

import com.eafit.nutrition.repository.NutricionistaRepository;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import java.util.List;

import java.util.Optional;

@Service

public class MedicionServiceConstructor {

private final MedicionRepository medicionRepository;

private final PacienteRepository pacienteRepository;

private final NutricionistaRepository nutricionistaRepository;

// Constructor que recibe los repositorios necesarios

// Spring automáticamente inyecta las implementaciones apropiadas

public MedicionServiceConstructor(

MedicionRepository medicionRepository,

PacienteRepository pacienteRepository,

NutricionistaRepository nutricionistaRepository) {

this.medicionRepository = medicionRepository;

this.pacienteRepository = pacienteRepository;

this.nutricionistaRepository = nutricionistaRepository;

}

@Transactional(readOnly = true)

public List<Medicion> findAll() {

return medicionRepository.findAll();

}

@Transactional(readOnly = true)

public Optional<Medicion> findById(Long id) {

return medicionRepository.findById(id);

}

// Otros métodos del servicio (omitidos por brevedad)

}

**5.2.2 Inyección por Campo (@Autowired)**

package com.eafit.nutrition.service;

import com.eafit.nutrition.model.Medicion;

import com.eafit.nutrition.repository.MedicionRepository;

import com.eafit.nutrition.repository.PacienteRepository;

import com.eafit.nutrition.repository.NutricionistaRepository;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import java.util.List;

import java.util.Optional;

@Service

public class MedicionServiceAutowired {

// Inyección de dependencias a nivel de campo usando @Autowired

@Autowired

private MedicionRepository medicionRepository;

@Autowired

private PacienteRepository pacienteRepository;

@Autowired

private NutricionistaRepository nutricionistaRepository;

@Transactional(readOnly = true)

public List<Medicion> findAll() {

return medicionRepository.findAll();

}

@Transactional(readOnly = true)

public Optional<Medicion> findById(Long id) {

return medicionRepository.findById(id);

}

// Otros métodos del servicio (omitidos por brevedad)

}

**5.2.3 Inyección por Setter**

package com.eafit.nutrition.service;

import com.eafit.nutrition.model.Medicion;

import com.eafit.nutrition.repository.MedicionRepository;

import com.eafit.nutrition.repository.PacienteRepository;

import com.eafit.nutrition.repository.NutricionistaRepository;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import java.util.List;

import java.util.Optional;

@Service

public class MedicionServiceSetter {

// Declaración de los repositorios necesarios

private MedicionRepository medicionRepository;

private PacienteRepository pacienteRepository;

private NutricionistaRepository nutricionistaRepository;

// Setter para inyectar el MedicionRepository

@Autowired

public void setMedicionRepository(MedicionRepository medicionRepository) {

this.medicionRepository = medicionRepository;

}

// Setter para inyectar el PacienteRepository

@Autowired

public void setPacienteRepository(PacienteRepository pacienteRepository) {

this.pacienteRepository = pacienteRepository;

}

// Setter para inyectar el NutricionistaRepository

@Autowired

public void setNutricionistaRepository(NutricionistaRepository nutricionistaRepository) {

this.nutricionistaRepository = nutricionistaRepository;

}

@Transactional(readOnly = true)

public List<Medicion> findAll() {

return medicionRepository.findAll();

}

@Transactional(readOnly = true)

public Optional<Medicion> findById(Long id) {

return medicionRepository.findById(id);

}

// Otros métodos del servicio (omitidos por brevedad)

}

**5.3 Comparativa de los Métodos de Inyección**

| **Característica** | **Inyección por Constructor** | **Inyección por Campo** | **Inyección por Setter** |
| --- | --- | --- | --- |
| Característica | Inyección por Constructor | Inyección por Campo | Inyección por Setter |
| Inmutabilidad | ✅ (permite campos finales) | ❌ | ❌ |
| Obligatoriedad | ✅ (todas las dependencias deben existir) | ✅ | ❌ (a menos que se use @Required) |
| Dependencias circulares | ❌ (las detecta temprano como error) | ✅ | ✅ |
| Facilidad para pruebas | ✅ | ❌ | ✅ |
| Verbosidad | Media | Baja | Alta |
| Reconfiguración | ❌ | ❌ | ✅ |
| Visibilidad de dependencias | ✅ | ❌ | ✅ |
| Recomendado por Spring | ✅ | ❌ | ⚠️ (solo para casos específicos) |

**6. Desarrollo de APIs REST**

Vamos a implementar controladores REST para exponer nuestras entidades y demostrar los diferentes métodos de inyección.

**6.1 Controlador para Mediciones**

package com.eafit.nutrition.controller;

import com.eafit.nutrition.model.Medicion;

import com.eafit.nutrition.service.MedicionServiceConstructor;

import com.eafit.nutrition.service.MedicionServiceAutowired;

import com.eafit.nutrition.service.MedicionServiceSetter;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

import java.util.Optional;

@RestController

@RequestMapping("/api/mediciones")

public class MedicionController {

// Inyección por constructor

private final MedicionServiceConstructor constructorService;

// Inyección por campo

@Autowired

private MedicionServiceAutowired autowiredService;

// Inyección por setter

private MedicionServiceSetter setterService;

public MedicionController(MedicionServiceConstructor constructorService) {

this.constructorService = constructorService;

}

@Autowired

public void setSetterService(MedicionServiceSetter setterService) {

this.setterService = setterService;

}

@GetMapping("/constructor")

public ResponseEntity<List<Medicion>> getAllMedicionesConstructor() {

return ResponseEntity.ok(constructorService.findAll());

}

@GetMapping("/autowired")

public ResponseEntity<List<Medicion>> getAllMedicionesAutowired() {

return ResponseEntity.ok(autowiredService.findAll());

}

@GetMapping("/setter")

public ResponseEntity<List<Medicion>> getAllMedicionesSetter() {

return ResponseEntity.ok(setterService.findAll());

}

@GetMapping("/compare/{id}")

public ResponseEntity<Map<String, Object>> compareMedicionById(@PathVariable Long id) {

Optional<Medicion> constructorResult = constructorService.findById(id);

Optional<Medicion> autowiredResult = autowiredService.findById(id);

Optional<Medicion> setterResult = setterService.findById(id);

Map<String, Object> response = new HashMap<>();

response.put("constructorService", constructorResult.orElse(null));

response.put("autowiredService", autowiredResult.orElse(null));

response.put("setterService", setterResult.orElse(null));

return ResponseEntity.ok(response);

}

@PostMapping("/constructor/paciente/{pacienteId}/nutricionista/{nutricionistaId}")

public ResponseEntity<Medicion> createMedicionConstructor(

@PathVariable Long pacienteId,

@PathVariable Long nutricionistaId,

@RequestBody Medicion medicion) {

Medicion createdMedicion = constructorService.createMedicion(pacienteId, nutricionistaId, medicion);

return new ResponseEntity<>(createdMedicion, HttpStatus.CREATED);

}

// Otros endpoints (omitidos por brevedad)

}

**6.2 Mostrando Información de Paciente y Nutricionista en las Mediciones**

Para mostrar información del paciente y nutricionista en las mediciones, podemos añadir propiedades virtuales:

import com.fasterxml.jackson.annotation.JsonProperty;

import java.util.HashMap;

import java.util.Map;

// En la clase Medicion

@JsonProperty("pacienteInfo")

public Map<String, Object> getPacienteInfo() {

if (paciente == null) {

return null;

}

Map<String, Object> info = new HashMap<>();

info.put("id", paciente.getId());

info.put("nombre", paciente.getNombre());

info.put("apellido", paciente.getApellido());

info.put("email", paciente.getEmail());

return info;

}

@JsonProperty("nutricionistaInfo")

public Map<String, Object> getNutricionistaInfo() {

if (nutricionista == null) {

return null;

}

Map<String, Object> info = new HashMap<>();

info.put("id", nutricionista.getId());

info.put("nombre", nutricionista.getNombre());

info.put("apellido", nutricionista.getApellido());

info.put("numeroLicencia", nutricionista.getNumeroLicencia());

info.put("especialidad", nutricionista.getEspecialidad());

return info;

}

**7. Buenas Prácticas y Solución de Problemas Comunes**

**7.1 Referencias Circulares en JSON**

Las referencias circulares son un problema común al serializar entidades JPA a JSON. Soluciones:

1. **Usar @JsonManagedReference y @JsonBackReference**:

// En la clase padre

@OneToMany(mappedBy = "nutricionista")

@JsonManagedReference

private List<Paciente> pacientes;

// En la clase hija

@ManyToOne

@JoinColumn(name = "nutricionista\_id")

@JsonBackReference

private Nutricionista nutricionista;

1. **Usar @JsonIgnoreProperties**:

@ManyToOne

@JoinColumn(name = "nutricionista\_id")

@JsonIgnoreProperties({"pacientes", "notas"})

private Nutricionista nutricionista;

**7.2 Optimización de Consultas**

Para optimizar las consultas JPA:

1. **Usa JOIN FETCH para cargar relaciones eficientemente**:

@Query("SELECT n FROM Nutricionista n LEFT JOIN FETCH n.pacientes WHERE n.id = :id")

Optional<Nutricionista> findByIdWithPacientes(@Param("id") Long id);

1. **Utiliza proyecciones para recuperar solo los datos necesarios**:

public interface PacienteResumen {

Long getId();

String getNombre();

String getApellido();

}

@Query("SELECT p.id as id, p.nombre as nombre, p.apellido as apellido FROM Paciente p WHERE p.nutricionista.id = :nutricionistaId")

List<PacienteResumen> findPacienteResumenByNutricionistaId(@Param("nutricionistaId") Long nutricionistaId);

1. **Usa consultas nativas para operaciones complejas**:

@Query(value = "SELECT \* FROM medicion WHERE paciente\_id = :pacienteId ORDER BY fecha DESC LIMIT 1", nativeQuery = true)

Optional<Medicion> findLastMedicionByPacienteId(@Param("pacienteId") Long pacienteId);

**7.3 Manejo de Transacciones**

Spring proporciona la anotación @Transactional para gestionar transacciones:

@Transactional(readOnly = true)

public List<Medicion> findAll() {

return medicionRepository.findAll();

}

@Transactional

public Medicion save(Medicion medicion) {

return medicionRepository.save(medicion);

}

**7.4 Validación de Entradas**

Utiliza Bean Validation para validar las entradas:

@Entity

@Table(name = "nutricionista")

public class Nutricionista {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@NotBlank(message = "El nombre es obligatorio")

@Size(min = 2, max = 100, message = "El nombre debe tener entre 2 y 100 caracteres")

@Column(name = "nombre", nullable = false, length = 100)

private String nombre;

@NotBlank(message = "El apellido es obligatorio")

@Size(min = 2, max = 100, message = "El apellido debe tener entre 2 y 100 caracteres")

@Column(name = "apellido", nullable = false, length = 100)

private String apellido;

@NotBlank(message = "El número de licencia es obligatorio")

@Column(name = "numero\_licencia", nullable = false, length = 50, unique = true)

private String numeroLicencia;

@Email(message = "El formato del email no es válido")

@NotBlank(message = "El email es obligatorio")

@Column(name = "email", nullable = false, length = 150, unique = true)

private String email;

// Resto de la clase...

}

**8. Ejemplos Completos de Código**

**8.1 Datos de Prueba (data.sql)**

-- Datos de prueba para la tabla nutricionista

INSERT INTO nutricionista (nombre, apellido, numero\_licencia, especialidad, email, telefono, activo)

VALUES ('Ana', 'García', 'NUT-2023-001', 'Nutrición Deportiva', 'ana.garcia@ejemplo.com', '555-1234', true);

INSERT INTO nutricionista (nombre, apellido, numero\_licencia, especialidad, email, telefono, activo)

VALUES ('Carlos', 'Rodríguez', 'NUT-2023-002', 'Nutrición Clínica', 'carlos.rodriguez@ejemplo.com', '555-2345', true);

-- Pacientes para la nutricionista Ana García (ID 1)

INSERT INTO paciente (nombre, apellido, fecha\_nacimiento, email, telefono, activo, nutricionista\_id)

VALUES ('Juan', 'Pérez', '1985-05-15', 'juan.perez@ejemplo.com', '555-1001', true, 1);

INSERT INTO paciente (nombre, apellido, fecha\_nacimiento, email, telefono, activo, nutricionista\_id)

VALUES ('Laura', 'Martínez', '1990-07-22', 'laura.martinez@ejemplo.com', '555-1002', true, 1);

-- Notas para el paciente Juan Pérez (ID 1)

INSERT INTO nota (titulo, contenido, fecha\_creacion, tipo\_nota, paciente\_id, nutricionista\_id)

VALUES ('Evaluación inicial', 'Paciente con sobrepeso leve. IMC: 27.5', '2023-01-15T10:30:00', 'Evaluación', 1, 1);

INSERT INTO nota (titulo, contenido, fecha\_creacion, tipo\_nota, paciente\_id, nutricionista\_id)

VALUES ('Plan alimenticio', 'Dieta hipocalórica con restricción moderada de carbohidratos', '2023-01-15T11:00:00', 'Plan', 1, 1);

-- Mediciones para Juan Pérez (ID 1)

INSERT INTO medicion (fecha, peso, altura, circunferencia\_cintura, circunferencia\_cadera, porcentaje\_grasa\_corporal, paciente\_id, nutricionista\_id)

VALUES ('2023-01-15', 85.5, 176.0, 98.0, 102.0, 24.5, 1, 1);

INSERT INTO medicion (fecha, peso, altura, circunferencia\_cintura, circunferencia\_cadera, porcentaje\_grasa\_corporal, paciente\_id, nutricionista\_id)

VALUES ('2023-02-15', 83.0, 176.0, 96.5, 101.0, 23.8, 1, 1);

**8.2 Método para Crear Mediciones (MedicionServiceConstructor)**

@Transactional

public Medicion createMedicion(Long pacienteId, Long nutricionistaId, Medicion medicion) {

// Obtenemos el paciente y el nutricionista usando sus respectivos repositorios

Paciente paciente = pacienteRepository.findById(pacienteId)

.orElseThrow(() -> new RuntimeException("Paciente no encontrado con id: " + pacienteId));

var nutricionista = nutricionistaRepository.findById(nutricionistaId)

.orElseThrow(() -> new RuntimeException("Nutricionista no encontrado con id: " + nutricionistaId));

// Asignamos el paciente y el nutricionista a la medición

medicion.setPaciente(paciente);

medicion.setNutricionista(nutricionista);

// Si no se especificó una fecha, usamos la fecha actual

if (medicion.getFecha() == null) {

medicion.setFecha(LocalDate.now());

}

// Guardamos la medición

return medicionRepository.save(medicion);

}

**8.3 Endpoint para Comparar Implementaciones**

@GetMapping("/compare/paciente/{pacienteId}/ultima-medicion")

public ResponseEntity<Map<String, Object>> compareLastMedicion(@PathVariable Long pacienteId) {

Optional<Medicion> constructorResult = constructorService.findLastMedicionByPacienteId(pacienteId);

Optional<Medicion> autowiredResult = autowiredService.findLastMedicionByPacienteId(pacienteId);

Optional<Medicion> setterResult = setterService.findLastMedicionByPacienteId(pacienteId);

Map<String, Object> response = new HashMap<>();

response.put("constructorService", constructorResult.orElse(null));

response.put("autowiredService", autowiredResult.orElse(null));

response.put("setterService", setterResult.orElse(null));

return ResponseEntity.ok(response);

}

**Conclusión**

En este tutorial, hemos explorado los conceptos fundamentales de Spring Boot y JPA utilizando H2 como base de datos. Aprendimos:

1. Cómo configurar H2 en diferentes modos y entornos
2. El modelado de entidades con relaciones complejas
3. Las estrategias de carga EAGER vs LAZY y cómo evitar el problema N+1
4. Los diferentes métodos de inyección de dependencias y sus pros y contras
5. Implementación de APIs REST y buenas prácticas

**Recursos Adicionales**

* [Documentación oficial de Spring Boot](https://docs.spring.io/spring-boot/docs/current/reference/html/)
* [Documentación de H2 Database](https://www.h2database.com/html/main.html)
* [Guía JPA de Baeldung](https://www.baeldung.com/hibernate-lazy-eager-loading)
* [Patrones de Inyección de Dependencias en Spring](https://www.baeldung.com/spring-dependency-injection)