Universidad ICESI

Spring Boot, Lombok & Spring Data JPA

Computación en Internet 2

Universidad ICESI

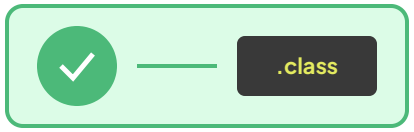
Agenda Temática de la Presentación

1. Arquitectura Spring Boot

Tomcat embebido, DispatcherServlet, Front Controller, estructura de carpetas y perfiles CLI.

2. Proyecto Lombok

Procesador de anotaciones en compilación, constructores, @Builder y prevención de referencias circulares.

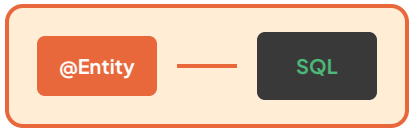


3. Spring Data JPA

Starters de persistencia, Datasource H2 vs Postgres, ddl-auto e inicialización data.sql.

4. JPA & Hibernate

Especificación JPA vs Motor Hibernate ORM, anotaciones de entidades y relaciones 1:N.



Universidad ICESI

Arquitectura Spring Boot y Servlets

Tomcat Embebido, DispatcherServlet y Configuración de Entorno

Universidad ICESI

Standalone JAR

Zero XML Config

Opinionated Starters

¿Qué es Spring Boot y por qué usarlo?

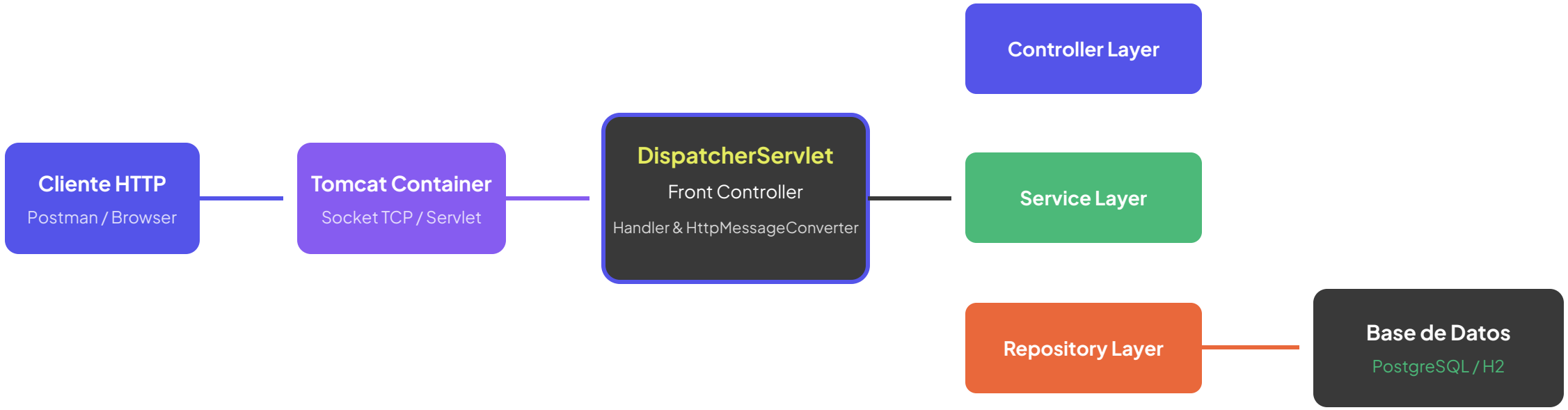
Spring Boot simplifica el desarrollo empresarial Java al proporcionar una capa de auto-configuración sobre el ecosistema Spring.

Auto-configuración:
Registra automáticamente los beans de Spring según los starters incluidos en el proyecto.

Tomcat Embebido:
Permite arrancar la aplicación como un ejecutable JAR independiente sin requerir Tomcat externo.

Flujo de una Petición HTTP en Spring Boot

Trayectoria paso a paso desde el cliente HTTP hasta la base de datos a través del **DispatcherServlet**:



Universidad ICESI

El Rol del DispatcherServlet

Front Controller en Spring MVC

- Punto de entrada centralizado para interceptar todas las peticiones HTTP entrantes.
- Consulta los `HandlerMapping` para identificar la clase controladora asignada a la URI.
- Utiliza `HttpMessageConverter` para serializar y deserializar objetos a JSON o XML.

Funciones del Servlet Container

Gestión de Ciclo de Vida:

Tomcat embebido inicializa el DispatcherServlet durante el arranque del contexto Spring.

Manejo de Excepciones:

Transforma excepciones no capturadas en respuestas de error HTTP estructuradas.

Modos de Despliegue: JAR vs WAR

1. Executable JAR (Recomendado)

Spring Boot arranca con Tomcat embebido mediante:

```
java -jar target/demo-app-0.0.1.jar
```

Ideal para contenedores Docker y microservicios independientes.

2. Servidor Externo (WAR)

Para desplegar en Tomcat independiente o WildFly, la clase principal extiende `SpringBootServletInitializer`:

```
public class ServletInitializer extends SpringBootServletInitializer {
  @Override
  protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
    return application.sources(DemoApplication.class);
  }
}
```

Universidad ICESI

Capas de la Arquitectura en Spring Boot

Model (Entidad / DTO)

Representa los datos del dominio y las tablas físicas. Sin lógica de infraestructura.

Repository (Persistencia)

Interfaz que extiende JpaRepository para gestionar la persistencia y consultas CRUD.

Service (Negocio)

Lógica de negocio pura, aislamiento de transacciones y reglas del dominio.

Controller (HTTP)

Recepción de solicitudes, validación de parámetros y retorno de modelo/JSON.

Ejemplo Integrado: Service & Repository

Capa de Servicio (ServiceImpl)

```java
@Service
@RequiredArgsConstructor
public class UserServiceImpl implements UserService {

  private final UserRepository userRepository;

  @Override
  public User createUser(User user) {
    if (user.getEmail() == null || user.getEmail().isBlank()) {
      throw new IllegalArgumentException("Email obligatorio");
    }
    return userRepository.save(user);
  }
}
```

Capa de Repositorio (Interface)

```java
package com.icesi.app.repository;

import com.icesi.app.model.User;
import org.springframework.data.jpa.repository.JpaRepository;
import java.util.Optional;

public interface UserRepository extends JpaRepository<User, Long> {
  Optional<User> findByEmail(String email);
}
```

Estructura de Carpetas & Regla Raíz

```
demo-app/
├── pom.xml
└── src/main/java/com/icesi/app/
    ├── DemoApplication.java (<-- Paquete Raíz)
    ├── controller/
    ├── service/
    │   └── impl/
    ├── repository/
    ├── model/
    └── config/
```

Regla de Escaneo (@ComponentScan)

Regla Raíz:

La clase anotada con `@SpringBootApplication` debe estar en la raíz de los paquetes.

Spring Boot escanea componentes únicamente de manera **descendente**. Si pones un controlador fuera de este árbol, no será detectado.

Ejecución con Maven & Gradle Wrappers

1. Maven Wrapper (mvnw)

Permite compilar y ejecutar sin instalar Maven globalmente:

```
# Linux / macOS:
./mvnw clean spring-boot:run

# Windows:
.\mvnw.cmd clean spring-boot:run
```

2. Gradle Wrapper (gradlew)

Ejecución mediante Gradle Wrapper:

```
# Linux / macOS:
./gradlew clean bootRun

# Windows:
.\gradlew.bat clean bootRun
```

Gestión de Perfiles de Entorno CLI

Perfil Maven CLI

Activa el perfil dev al arrancar:
`./mvnw spring-boot:run -Dspring-boot.run.profiles=dev`

Perfil Gradle CLI

Paso de argumentos en Gradle:
`./gradlew bootRun --args='--spring.profiles.active=dev'`

Ejecución de JAR

Pasando propiedad JVM:
`java -Dspring.profiles.active=dev -jar app.jar`

Universidad ICESI

Proyecto Lombok

Eliminación de Código Repetitivo y Anotaciones en Spring Boot

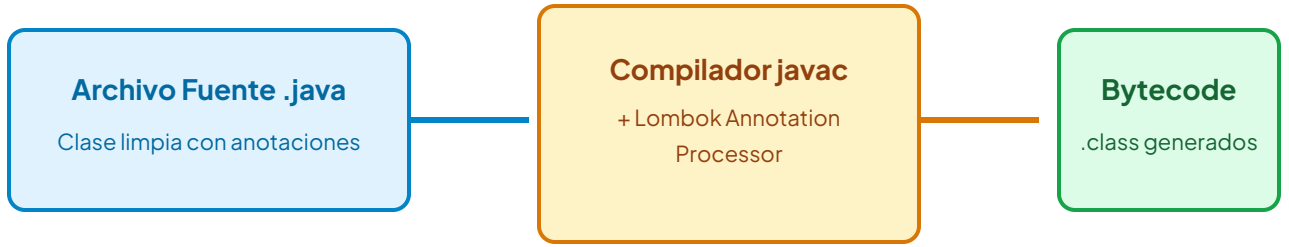
Universidad ICESI

Zero Boilerplate

Compile-Time Only

¿Qué es Lombok y cómo actúa?

Project Lombok es un procesador de anotaciones en tiempo de compilación (`javac`) que inyecta bytecode directamente en el archivo `.class`.



Configuración de Lombok

Maven (pom.xml)

```xml
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <optional>true</optional>
</dependency>
```

Gradle (build.gradle)

```
dependencies {
  compileOnly 'org.projectlombok:lombok'
  annotationProcessor 'org.projectlombok:lombok'
}
```

Encapsulamiento: @Getter y @Setter

Uso a Nivel de Clase y Atributo

```java
package com.icesi.app.model;

import lombok.Getter;
import lombok.Setter;
import lombok.AccessLevel;

@Getter
@Setter
public class Product {
  private Long id;
  private String name;

  // Sobrescribe visibilidad solo para este setter
  @Setter(AccessLevel.PRIVATE)
  private Double price;
}
```

Ventajas Clave

- Genera automáticamente métodos `getName()`, `setName()` en tiempo de compilación.
- Permite restringir la visibilidad a `PRIVATE` o `PROTECTED` por campo.

Universidad ICESI

Anotaciones de Constructores en Lombok

@NOARGSCONSTRUCTOR

Genera un constructor sin parámetros. Es requerido obligatoriamente por JPA e Jackson para instanciación por reflexión.

@ALLARGSCONSTRUCTOR

Genera un constructor que acepta un parámetro por cada atributo declarado en la clase en el mismo orden.

@REQUIREDARGSCONSTRUCTOR

Genera un constructor únicamente para los campos marcados como `final` o `@NonNull`.

Inyección por Constructor con Lombok

Antes (Forma Tradicional)

```
@Service
public class UserServiceImpl implements UserService {

  private final UserRepository userRepository;

  public UserServiceImpl(UserRepository userRepository) {
    this.userRepository = userRepository;
  }
}
```

Ahora (Con @RequiredArgsConstructor)

```
@Service
@RequiredArgsConstructor
public class UserServiceImpl implements UserService {

  // Spring inyecta automáticamente por constructor
  private final UserRepository userRepository;
}
```

Garantiza la inmutabilidad de dependencias obligatorias y elimina el código repetitivo.

Patrón de Diseño: @Builder

Definición del Modelo

```java
@Getter
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Customer {
  private Long id;
  private String firstName;
  private String lastName;
  private String email;
  private Boolean active;
}
```

Uso Fluido del Builder

```java
Customer customer = Customer.builder()
    .id(1L)
    .firstName("Kevin")
    .lastName("Coes")
    .email("kevin@icesi.edu.co")
    .active(true)
    .build();
```

Permite instanciar objetos complejos de manera legible sin importar el orden de los parámetros.

Logging con @Slf4j e Inmutabilidad con @Value

Logging Automático (@Slf4j)

```java
@Service
@Slf4j
public class NotificationService {

  public void sendEmail(String to, String subject) {
    log.info("Enviando correo a: {}", to);
    log.debug("Detalle del envío...");
  }
}
```

Clases Inmutables (@Value)

```java
@Value
public class UserLocation {
  String city;
  String country;
  // Hace todos los campos 'private final'
  // Genera getters, toString, hashCode, sin setters
}
```

DTOs Mutables con @Data

Anotación Agrupadora @Data

```java
package com.icesi.app.dto;

import lombok.Data;

@Data
public class UserResponseDto {
  private Long id;
  private String name;
  private String email;
}
```

¿Qué incluye @Data internamente?

- `@Getter` (para todos los campos)
- `@Setter` (para todos los campos)
- `@ToString`
- `@EqualsAndHashCode`
- `@RequiredArgsConstructor`

Universidad ICESI

Precaución Crítica: @Data en JPA

No @Data en Entidades

Usar @Getter/@Setter

Riesgo de StackOverflowError:

`@Data` genera `hashCode()` y `equals()` sobre TODOS los campos. En relaciones bidireccionales JPA (`@OneToMany` / `@ManyToOne`), provoca bucles infinitos.

Recomendación Correcta para Entidades JPA:

```
@Entity
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class User { ... }
```

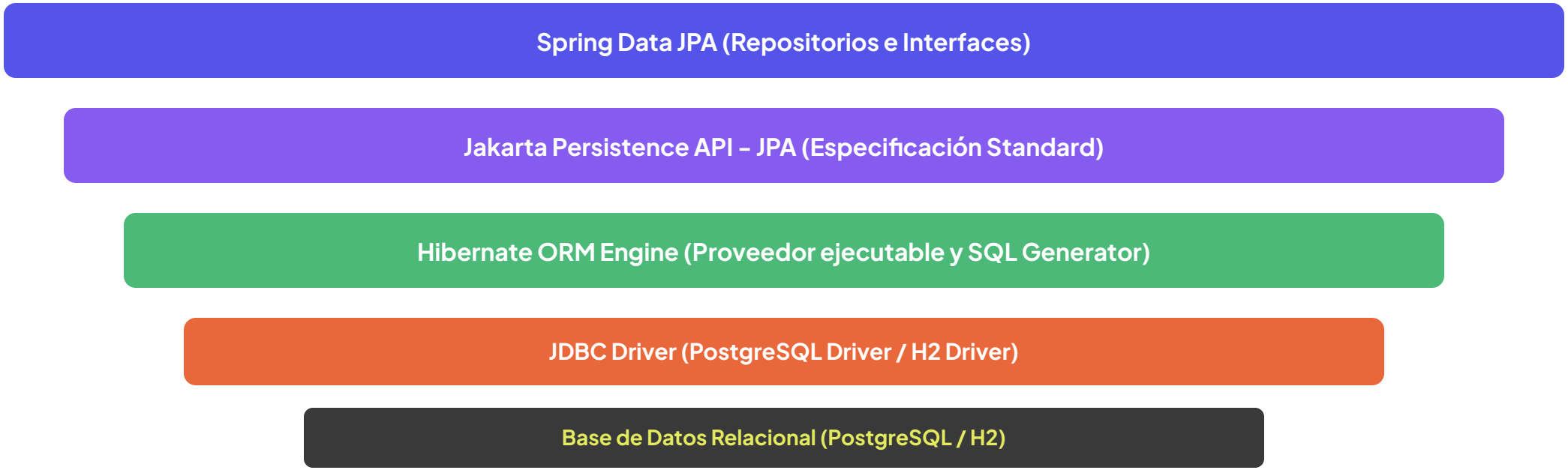
Universidad ICESI

Spring Data JPA y Persistencia

Capa de Persistencia, Drivers H2 vs PostgreSQL, ddl-auto y Scripts SQL

¿Qué es Spring Data JPA? Capas de Persistencia

Spring Data JPA es una capa de abstracción de alto nivel que se sitúa por encima del estándar JPA y del motor ORM (Hibernate):



Universidad ICESI

Dependencias Starters en Maven y Gradle

Maven (pom.xml)

```xml
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <scope>runtime</scope>
</dependency>
```

Gradle (build.gradle)

```
dependencies {
  implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
  runtimeOnly 'com.h2database:h2'
  runtimeOnly 'org.postgresql:postgresql'
}
```

Desglose de application.properties para Persistencia

```
# ========================================
# 1. CONFIGURACIÓN DE DATASOURCE (PostgreSQL / H2)
# ========================================
spring.datasource.url=jdbc:postgresql://localhost:5432/boardgame
spring.datasource.username=postgres
spring.datasource.password=postgres
spring.datasource.driver-class-name=org.postgresql.Driver

# Dialecto de Hibernate para optimizar SQL generado
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect

# ========================================
# 2. GESTIÓN DE ESQUEMA DDL (ddl-auto)
# ========================================
spring.jpa.hibernate.ddl-auto=create-drop

# ========================================
# 3. INICIALIZACIÓN DE SCRIPTS DATA.SQL
# ========================================
spring.sql.init.mode=always
spring.jpa.defer-datasource-initialization=true
```

Configuración de H2 y DB_CLOSE_DELAY=-1

1. H2 en Memoria vs Archivo

Opción A (RAM):

```
spring.datasource.url=jdbc:h2:mem:testdb;DB_CLOSE_DELAY=-1
```

Opción B (Archivo Local):

```
spring.datasource.url=jdbc:h2:./testdb;DB_CLOSE_DELAY=-1
```

2. Importancia de DB_CLOSE_DELAY=-1

Mantiene viva la BD:

Por defecto H2 destruye la base de datos en RAM apenas se cierra la última conexión. `DB_CLOSE_DELAY=-1` fuerza a mantener la BD viva durante toda la JVM de Spring Boot.

Estrategias de Generación DDL (ddl-auto)

create-drop

Crea el esquema al iniciar la aplicación y lo destruye completamente al cerrar la JVM.

update

Actualiza la estructura de la base de datos agregando nuevas columnas sin borrar datos.

validate

Valida que el esquema de la BD coincida con las entidades JPA (falla si hay diferencias).

none

No realiza ninguna modificación en la base de datos. Recomendado en producción con Flyway/Liquibase.

Inicialización de Datos con data.sql

Propiedad Defer Initialization

```
spring.sql.init.mode=always
spring.jpa.defer-datasource-initialization=true
```

Al colocar `defer-datasource-initialization=true`, Spring Boot fuerza a esperar a que Hibernate cree las tablas (`ddl-auto`) **ANTES** de intentar ejecutar las sentencias `data.sql`.

Archivo data.sql

```
-- Ubicación: src/main/resources/data.sql
INSERT INTO usuarios (nombre, email)
VALUES ('Kevin', 'kevin@icesi.edu.co');

INSERT INTO usuarios (nombre, email)
VALUES ('Ana', 'ana@icesi.edu.co');
```

Consola Web de H2 Database

Habilitación en application.properties

```
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
```

Permite ingresar a la interfaz web interactiva desde el navegador para consultar tablas en tiempo real.

Acceso y Credenciales

- URL de acceso: `http://localhost:8080/h2-console`
- JDBC URL: `jdbc:h2:mem:testdb`
- Usuario: `sa` / Password: *(vacío)*

Universidad ICESI

JPA, Hibernate y Mapeo ORM

Mapeo Objeto-Relacional (ORM) y Anotaciones de Entidades

Universidad ICESI

API Standard Spec

Hibernate Engine

Especificación JPA vs Motor Hibernate

JPA define la especificación estándar Java; Hibernate es el motor ORM que traduce las operaciones sobre objetos Java a sentencias SQL físicas.

Anotaciones Fundamentales de Mapeo JPA

@Entity & @Table

`@Entity` declara que la clase es administrada por JPA. `@Table(name="productos")` especifica la tabla física en la BD.

@Id & @GeneratedValue

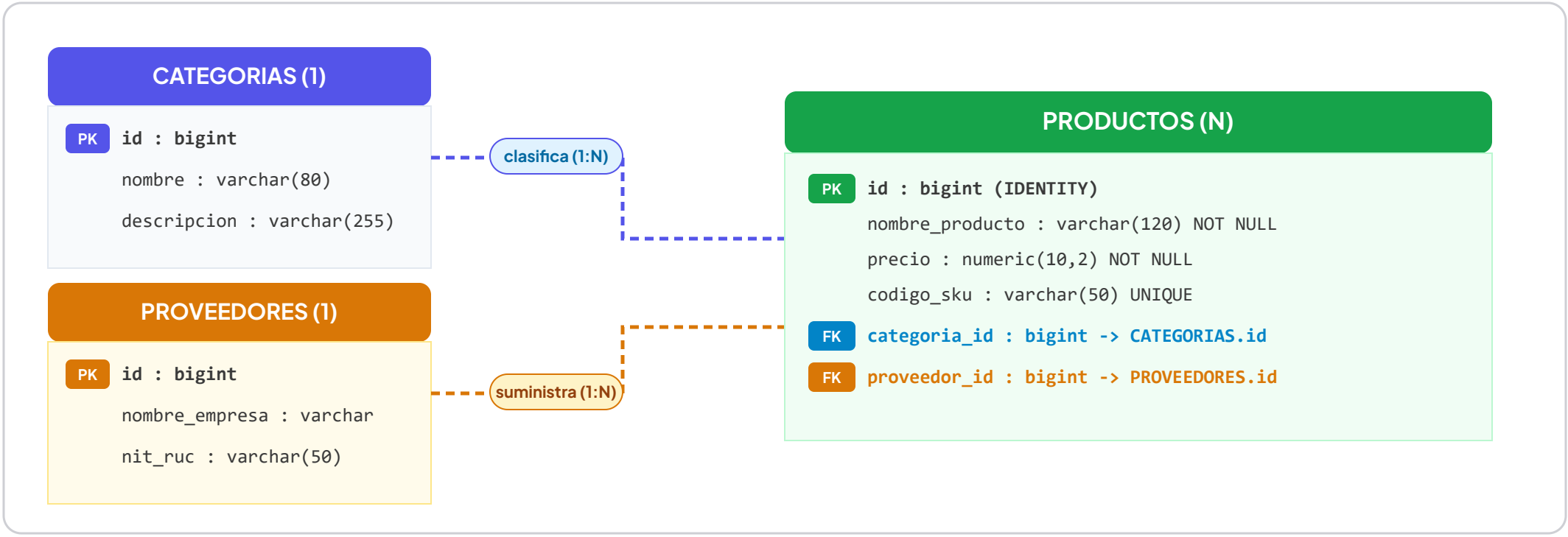
`@Id` marca la clave primaria. `@GeneratedValue(strategy = GenerationType.IDENTITY)` usa autoincremento de la BD.

@Column & @Transient

`@Column` define restricciones (nullable, length, unique). `@Transient` omite el atributo de la persistencia.

Modelo Entidad–Relación (Diagrama ER)

Esquema relacional de la base de datos para el dominio de productos:



Mapeo de Relaciones @ManyToOne

Entidad Product con Claves Foráneas

```java
@Entity
@Table(name = "productos")
@Getter @Setter @NoArgsConstructor @AllArgsConstructor @Builder
public class Product {

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

  @Column(name = "nombre_producto", nullable = false, length = 120)
  private String name;

  @Transient
  private String temporaryDiscountCode;

  @ManyToOne(fetch = FetchType.LAZY)
  @JoinColumn(name = "categoria_id", nullable = false)
  private Category category;
}
```

Buenas Prácticas en Mapeo Relacional

- `FetchType.LAZY` evita cargar la categoría de la BD hasta que se invoque `getCategory()`.
- `@JoinColumn(name = "categoria_id")` establece explícitamente el nombre de la columna FK física.
- `@Transient` le indica a JPA que ignore ese atributo durante la persistencia.