JPA, ARQUITECTURA POR CAPAS Y ESTRATEGIA DE PRUEBAS

Una guía académica sobre persistencia de datos, diseño arquitectónico en Spring Boot y metodologías integrales de pruebas de software.

INGENIERÍA DE SOFTWARE

Spring Boot & Persistencia Avanzada

28 de julio de 2026

GENERALIDADES DE JPA

¿Qué es JPA?

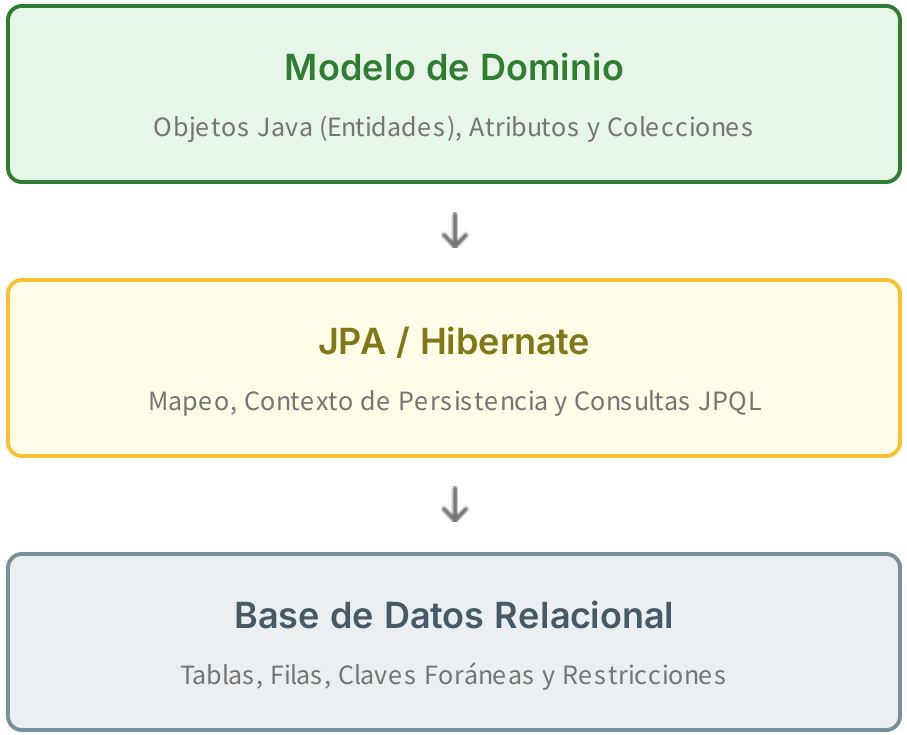
Jakarta Persistence API es una especificación estándar de Java para el mapeo objeto-relacional (ORM). Actúa como un contrato que define cómo persistir objetos Java en bases de datos relacionales sin depender de implementaciones específicas.

¿Por qué usar JPA?

- **Reduce el código JDBC repetitivo y propenso a errores.**
- **Centraliza las reglas de persistencia en el modelo.**
- **Mejora la mantenibilidad y portabilidad del software.**
- **Integración natural con transacciones y validaciones.**

Nota: JPA no elimina la necesidad de entender SQL; optimiza la interacción pero requiere un buen diseño de base de datos.

Capas de Persistencia



Implementaciones: Hibernate es el proveedor más común, pero JPA también es implementado por EclipseLink u OpenJPA.

JPA FRENTE AL PATRÓN DAO

Patrón DAO Tradicional

- Encapsula operaciones de acceso tras una interfaz abstracta.
- Implementación manual mediante JDBC, MyBatis o EntityManager.
- Control total sobre el SQL generado y el mapeo de resultados.
- Requiere mayor volumen de código repetitivo (Boilerplate).
- Ideal para sistemas heterogéneos o desacoplamiento tecnológico estricto.

JPA / Spring Data JPA

- Especificación ORM enfocada en el modelado del dominio.
- Abstracción de Repositorios (JpaRepository) que reduce código.
- CRUD, paginación y ordenamiento gestionados automáticamente.
- Consultas declarativas, derivadas por nombre y soporte JPQL.
- Alta productividad en dominios relacionales y entornos empresariales.

Mensaje Clave: Un repositorio de Spring Data JPA puede desempeñar el rol de DAO, centralizando el acceso a datos sin mezclar lógica de negocio en el controlador.

MAPA DE ARQUITECTURA JPA POR CAPAS

Separación de responsabilidades: cada capa transforma, valida o persiste sin asumir el trabajo de las demás.

01

Entidades

Modelo persistente y relaciones JPA.

→

02

Repository

Acceso a datos mediante Spring Data JPA.

→

03

Servicios

Reglas de negocio y transacciones.

→

04

DTO

Contrato de entrada y salida de la API.

→

05

Controladores

Frontera HTTP y códigos de respuesta.

Regla de diseño: el controlador no accede al repositorio; el servicio orquesta el caso de uso y los DTO evitan exponer directamente las entidades.

ENTIDADES Y ANOTACIONES DE MAPEO

Anotación	Propósito y Uso
`@Entity`	Define la clase como una entidad persistente de JPA.
`@Table`	Especifica el nombre físico de la tabla en la BD.
`@Id`	Marca el atributo que actúa como Clave Primaria (PK).
`@GeneratedValue`	Define la estrategia de generación de IDs (Identity, Sequence).
`@Column`	Configura restricciones (nullable, length, unique, name).
`@Enumerated`	Persiste enums como String o Ordinal (preferir String).
`@Transient`	Excluye el atributo del mapeo y la persistencia.

ESTUDIANTE.JAVA

```java
@Entity
@Table(name = "estudiantes")
public class Estudiante {

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

    @Column(nullable = false, length = 120)
    private String nombre;

    @Version
private Long version;

    // Getters, Setters, Constructor...
}
```

Control de Concurrencia Optimista

La anotación **`@Version`** detecta actualizaciones conflictivas concurrentes, lanzando una `OptimisticLockException` si la versión en BD ha cambiado desde que se leyó la entidad.

RELACIONES ENTRE ENTIDADES · DEFINICIÓN Y BUENAS PRÁCTICAS

Asociaciones Simples

`@ManyToOne` : Lado propietario frecuente de la FK. Atributo `optional=false` para validación en BD.

`@OneToMany` : Lado inverso. Obligatorio usar `mappedBy` para evitar tablas duplicadas.

`@JoinColumn` : Define el nombre físico de la columna de unión y sus restricciones de nulidad.

Estrategias de Carga (Fetch)

LAZY Carga bajo demanda. **Preferido** para colecciones y relaciones pesadas para optimizar memoria.

EAGER Carga inmediata. Riesgo de consultas innecesarias y el problema de rendimiento N+1.

Solución N+1: JOIN FETCH, @EntityGraph o proyecciones DTO.

Relaciones Complejas

`@ManyToMany` : Requiere una tabla intermedia para romper la relación de muchos a muchos.

`@JoinTable` : Personaliza la tabla de unión, definiendo `joinColumns` e `inverseJoinColumns` .

Recomendación: Utilizar Set en lugar de List para evitar duplicados en ManyToMany.

Operaciones en Cascada

Define cómo se propagan los cambios: `PERSIST` , `MERGE` , `REMOVE` , `REFRESH` , `DETACH` .

⚠ **Evitar CascadeType.ALL por costumbre.**

Usar cascada solo cuando el ciclo de vida de la entidad hija dependa estrictamente del padre.

RELACIONES JPA: MANYTOONE Y ONETOMANY

PROGRAMA.JAVA · ENTIDAD PRINCIPAL

```java
@Entity
@Getter @NoArgsConstructor
class Programa {
  @Id @GeneratedValue
private Long id;

  @OneToMany(mappedBy = "programa",
      cascade = CascadeType.PERSIST,
      orphanRemoval = false)
  private Set<Estudiante> estudiantes = new HashSet<>();

  public void agregar(Estudiante estudiante) {
    estudiantes.add(estudiante);
    estudiante.setPrograma(this);
  }
}

@Entity
@ManyToOne(fetch = FetchType.LAZY,
    optional = false)
@JoinColumn(name = "programa_id", nullable = false)
private Programa programa;
```

¿Quién es el propietario?

`@ManyToOne` mantiene la clave foránea en la tabla de **Estudiante**. Por eso es el lado propietario de la relación.

Lectura del ejemplo

mappedBy = "programa" evita que el lado inverso cree una segunda relación.

LAZY evita cargar todos los estudiantes al consultar un programa.

optional=false y **nullable=false** refuerzan la obligatoriedad.

El método **agregar** mantiene sincronizados ambos lados.

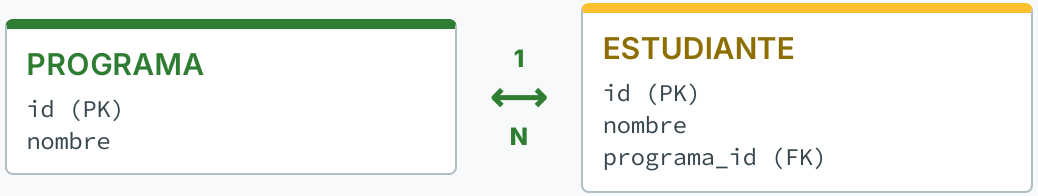
Regla práctica: configure la relación desde el lado que posee la FK y use métodos de dominio para mantener la consistencia bidireccional.

RELACIONES JPA: ONETOMANY Y MANYTOONE

PROGRAMA.JAVA + ESTUDIANTE.JAVA · MAPEO 1:N

```java
@Entity
@Getter @NoArgsConstructor
class Programa {
  @Id @GeneratedValue
private Long id;

  @OneToMany(mappedBy = "programa",
      fetch = FetchType.LAZY)
  private Set<Estudiante> estudiantes =
      new HashSet<>();

  public void agregar(Estudiante estudiante) {
    estudiantes.add(estudiante);
    estudiante.setPrograma(this);
  }
}

@Entity
@Getter @NoArgsConstructor
class Estudiante {
  @Id @GeneratedValue
private Long id;

  @ManyToOne(fetch = FetchType.LAZY,
      optional = false)
  @JoinColumn(name = "programa_id",
      nullable = false)
  private Programa programa;
}
```

Mapeo entidad–relación



Cardinalidad: un programa puede tener muchos estudiantes; cada estudiante pertenece a un programa.

Lectura del mapeo

@ManyToOne: mantiene la FK `programa_id`; es el lado propietario.

@OneToMany: usa `mappedBy = "programa"`; no crea una segunda relación.

LAZY: evita cargar la colección completa cuando no se necesita.

optional=false / nullable=false: expresan la obligatoriedad en objeto y base de datos.

Regla práctica: configure ambos lados con un método de dominio como **agregar** para conservar la consistencia bidireccional.

RELACIÓN JPA: MANYTOMANY

¿Cuándo usarla?

Cuando varias instancias de una entidad pueden asociarse con varias instancias de otra. En el dominio: un **Curso** puede tener muchos **Estudiantes** y un estudiante puede matricularse en muchos cursos.

Modelo relacional

La tabla intermedia materializa la asociación mediante dos claves foráneas.

Tabla intermedia

curso_id: FK hacia `curso.id`.

estudiante_id: FK hacia `estudiante.id`.

La combinación de ambas columnas puede formar la clave primaria.

Propietario e inverso

El lado que declara `@JoinTable` es el **propietario** de la relación.

El otro lado usa `mappedBy` para reflejar el mapeo sin crear otra tabla.

Preferir `Set` para evitar asociaciones duplicadas.

Regla práctica: mantener ambos lados sincronizados con métodos de dominio y cargar la relación con **LAZY** salvo que exista una necesidad explícita.

MANYTOMANY: MAPEO BIDIRECCIONAL

CURSO.JAVA + ESTUDIANTE.JAVA · TABLA CURSO_ESTUDIANTE

```java
@Entity
class Curso {
  @Id @GeneratedValue
private Long id;

  @ManyToMany(fetch = FetchType.LAZY)
  @JoinTable(name = "curso_estudiante",
    joinColumns = @JoinColumn(name = "curso_id"),
    inverseJoinColumns = @JoinColumn(name = "estudiante_id"))
  private Set<Estudiante> estudiantes = new HashSet<>();

  public void agregarEstudiante(Estudiante e) {
    estudiantes.add(e);
    e.getCursos().add(this);
  }

  public void retirarEstudiante(Estudiante e) {
    estudiantes.remove(e);
    e.getCursos().remove(this);
  }
}

@Entity
class Estudiante {
  @ManyToMany(mappedBy = "estudiantes", fetch = FetchType.LAZY)
  private Set<Curso> cursos = new HashSet<>();
}
```

Lectura de @JoinTable

`joinColumns` referencia la entidad propietaria: **Curso**.

`inverseJoinColumns` referencia la entidad asociada: **Estudiante**.

`mappedBy` apunta al atributo **estudiantes** del propietario.

Esquema resultante

curso	estudiante
id (PK)	id (PK)

curso_estudiante
curso_id (FK) · estudiante_id (FK)

Nota: no usar **CascadeType.ALL** por costumbre. En asociaciones entre entidades independientes, el ciclo de vida debe controlarse de forma explícita.

ONETOONE: RELACIÓN Y CICLO DE VIDA

ESTUDIANTE.JAVA + PERFILESTUDIANTE.JAVA · FK ÚNICA

```java
@Entity
class Estudiante {
  @Id @GeneratedValue
private Long id;

  @OneToOne(mappedBy = "estudiante",
      fetch = FetchType.LAZY,
      cascade = {CascadeType.PERSIST, CascadeType.MERGE},
      orphanRemoval = true)
  private PerfilEstudiante perfil;

  public void asignarPerfil(PerfilEstudiante perfil) {
    this.perfil = perfil;
    perfil.setEstudiante(this);
  }
}

@Entity
class PerfilEstudiante {
  @OneToOne(fetch = FetchType.LAZY, optional = false)
  @JoinColumn(name = "estudiante_id",
      nullable = false, unique = true)
  private Estudiante estudiante;
}
```

¿Dónde vive la relación?

ESTUDIANTE
`id (PK)`
`perfil`

1 ⟷ 1

PERFIL_ESTUDIANTE
`id (PK)`
`estudiante_id (FK, UNIQUE)`

La FK se almacena en **PerfilEstudiante**; `unique=true` garantiza como máximo un perfil por estudiante.

Decisiones de diseño

`optional=false` y `nullable=false` solo si el perfil es obligatorio.

`orphanRemoval=true` solo si el perfil no tiene ciclo de vida independiente.

Cascadas limitadas a `PERSIST` y `MERGE`; evitar **ALL** sin justificación.

Rendimiento: mantener **LAZY** y definir consultas de lectura de forma explícita cuando se necesite el perfil.

REPOSITORIOS Y ESTRATEGIAS DE CONSULTA

¿Qué aporta un repositorio?

Es la frontera de acceso a datos. **Spring Data JPA** ofrece CRUD, paginación y ordenamiento a través de **JpaRepository**, evitando código repetitivo y manteniendo el servicio desacoplado del detalle de persistencia.

```
interface EstudianteRepository
extends JpaRepository<Estudiante, Long> { }
```

Consultas derivadas para filtros simples.

JPQL o proyecciones para consultas sobre el modelo.

SQL nativo solo cuando existe una necesidad validada.

Buenas prácticas de consulta

- Parametrizar siempre; nunca concatenar datos de entrada.
- Usar `Pageable` para colecciones grandes.
- Prevenir N+1 con `JOIN FETCH`, `@EntityGraph` o proyecciones DTO.
- Revisar duplicados y paginación al combinar `join fetch`.
- Validar SQL nativo con plan de ejecución y pruebas de rendimiento.

Decisión rápida:
Derivada → filtro simple · JPQL → asociaciones y dominio · SQL nativo → dialecto, legado u optimización comprobada.

REPOSITORIOS: EJEMPLOS JPQL Y SQL NATIVO

JPQL sobre el modelo de dominio

Consulta entidades y atributos Java. Es la opción preferida para asociaciones y reglas que deben conservar portabilidad entre motores.

```
@Query("select distinct e from Estudiante e"
       + " join fetch e.programa"
       + " where e.estado = :estado")
List<Estudiante> buscarConPrograma(
    @Param("estado") Estado estado);

// También puede proyectar directamente a un DTO
@Query("select new ...EstudianteResumen(e.id, e.nombre) "
       + "from Estudiante e where e.programa.id = :id")
```

- **JOIN FETCH:** carga la asociación en la misma consulta y ayuda a prevenir N+1.
- **Proyección DTO:** trae solo las columnas necesarias para lecturas.
- Parametrice siempre los valores con **@Param**.

Preferir JPQL cuando: la consulta expresa relaciones del dominio y no depende de un dialecto específico.

SQL nativo para necesidades validadas

Consulta tablas, columnas y funciones del motor. Úselo para legado, características exclusivas o una optimización comprobada.

```
@Query(value = "select e.id, e.nombre, p.nombre as programa "
       + "from estudiantes e join programas p "
       + "on p.id = e.programa_id "
       + "where e.estado = :estado",
       nativeQuery = true)
List<EstudianteRow> buscarNativo(
    @Param("estado") String estado);
```

- Valide el plan con **EXPLAIN** y una prueba de rendimiento.
- Considere la menor portabilidad y el acoplamiento al esquema físico.
- **Nunca concatene** datos de entrada; use parámetros nombrados.

Reservar SQL nativo: no es un sustituto de JPQL por defecto; debe existir una razón técnica documentada.

Selección: consulta derivada → filtro simple · JPQL → entidades, asociaciones y DTO · SQL nativo → motor, legado u optimización comprobada.

CAPA DE SERVICIO Y TRANSACCIONALIDAD

Responsabilidades

- Implementación de reglas de negocio complejas.
- Orquestación de múltiples repositorios.
- Mapeo entre Entidades y DTOs.
- Validaciones de estado y dominio.

Gestión Transaccional

Asegura la atomicidad (ACID) de las operaciones mediante la anotación **@Transactional**.

Escritura	Lectura
Por defecto. Abre transacción para persistencia.	readOnly = true. Optimiza consultas JPA.

Evitar que el servicio sea un simple "pasamanos" del repositorio.

```java
@Service
@RequiredArgsConstructor
@Transactional
public class EstudianteService {
  private final EstudianteRepository repository;
  private final ProgramaService programaService;
  private final ModelMapper modelMapper;

  public EstudianteResponse crear(CrearEstudianteRequest request) {
    // 1. Regla de negocio y asociación en el servicio
    Programa programa = programaService.buscarPorId(request.programaId());

    // 2. DTO de entrada → entidad persistente
    Estudiante estudiante = modelMapper.map(request, Estudiante.class);
    estudiante.setPrograma(programa);

    // 3. Persistencia y entidad → DTO de salida
var guardado = repository.save(estudiante);
    return modelMapper.map(guardado, EstudianteResponse.class);
  }

  @Transactional(readOnly = true)
  public EstudianteResponse buscarPorId(Long id) { ... }
}
```

DATA TRANSFER OBJECTS (DTO)

↔ El DTO como contrato

- **Desacopla:** separa la entidad persistente del contrato público de la API.
- **Valida:** aplica Bean Validation en el punto de entrada.
- **Protege:** evita exponer passwordHash, relaciones internas o IDs sensibles.
- **Separa sentidos:** Request DTO para entrada y Response DTO para salida.

```
public record EstudianteRequest(
   @NotBlank String nombre,
   @NotNull Long programaId
) {}
public record EstudianteResponse(
   Long id, String nombre, String programa
) {}
```

Flujo de transformación

- **Request DTO → Entity:** el servicio mapea los datos validados y resuelve asociaciones.
- **Entity → Response DTO:** se seleccionan solo los campos que el cliente necesita.
- **Pruebas:** verificar nulos, enums, fechas, colecciones y campos sensibles.

```
var entity = modelMapper.map(request, Estudiante.class);
entity.setPrograma(programaService.buscarPorId(request.programaId()));
return modelMapper.map(entity, EstudianteResponse.class);
```

> **Regla:** una entidad JPA no es un contrato REST. El DTO define qué entra y qué sale.

💡 DTO = contrato explícito y estable.

Request ≠ Response ≠ Entity

CONTROLADORES RESTFUL: GENERALIDADES Y RESPONSABILIDADES

Enlace de la solicitud

`@PathVariable`
Identifica un recurso en la URI: `/estudiantes/{id}`.

`@RequestParam`
Filtros, búsqueda, orden y paginación: `?page=0&size=20`.

`@RequestBody`
Deserializa el JSON en un **Request DTO**; se combina con `@Valid`.

Cabeceras y contexto
`@RequestHeader` recibe metadatos; la respuesta puede incluir `Location`, ETag o caché.

Anotaciones de contrato

`@RestController` `@RequestMapping` `@GetMapping` `@PostMapping` `@PutMapping` `@PatchMapping` `@DeleteMapping` `@Valid`

Lo que NO debe hacer

No contiene reglas de negocio, no abre transacciones manuales, no accede al repositorio y no retorna entidades JPA como contrato público.

Responsabilidad paso a paso

1. **Interpretar HTTP**
   Selecciona la operación según método, URI, parámetros, cabeceras y formato del cuerpo.
2. **Validar el contrato**
   `@Valid` ejecuta Bean Validation; los errores de forma se responden como **400**.
3. **Delegar al servicio**
   Entrega DTO e identidad del recurso; el servicio aplica reglas, autorización de dominio y transacción.
4. **Construir la respuesta**
   Recibe un **Response DTO** y define estado, cabeceras y cuerpo mediante `ResponseEntity`.
5. **Mantener coherencia**
   Versiona rutas, documenta con OpenAPI y conserva nombres, códigos y formatos homogéneos.

Separación estricta: Controlador = protocolo HTTP · Servicio = reglas y transacción · Repositorio = persistencia.

CONTROLADORES RESTFUL: EJEMPLO Y SEMÁNTICA

ESTUDIANTECONTROLLER.JAVA

```java
@RestController
@RequestMapping("/api/v1/estudiantes")
@RequiredArgsConstructor
class EstudianteController {
  private final EstudianteService service;

  @GetMapping("/{id}")
  ResponseEntity<EstudianteResponse> buscar(
      @PathVariable Long id) {
    return ResponseEntity.ok(service.buscar(id));
  }

  @GetMapping
  Page<EstudianteResumen> listar(
      @RequestParam(defaultValue="0") int page,
      @RequestParam(defaultValue="20") int size) {
    return service.listar(PageRequest.of(page, size));
  }

  @PostMapping
  ResponseEntity<EstudianteResponse> crear(
      @Valid @RequestBody EstudianteRequest req) {
    var res = service.crear(req);
    var uri = URI.create("/api/v1/estudiantes/" + res.id());
    return ResponseEntity.created(uri).body(res);
  }

  @DeleteMapping("/{id}")
  ResponseEntity<Void> eliminar(@PathVariable Long id) {
    service.eliminar(id);
    return ResponseEntity.noContent().build();
  }
}
```

Semántica esperada

Método	Uso	Éxito	Idempotente
GET	Consultar	200	Sí
POST	Crear	201	No
PUT	Reemplazar	200/204	Sí
PATCH	Cambio parcial	200/204	Depende
DELETE	Eliminar	204	Sí

Decisiones del contrato

- **Creación:** devuelva **201 Created** y la cabecera `Location` del nuevo recurso.
- **Paginación:** use `Pageable`; limite `size` y declare ordenamiento estable.
- **PUT vs PATCH:** PUT reemplaza el recurso; PATCH modifica campos explícitos sin sobrescribir los omitidos.
- **Errores:** el controlador deja el formato homogéneo al manejador global, no captura excepciones genéricas.

ResponseEntity controla estado, cabeceras y cuerpo.

El ejemplo omite autenticación y documentación OpenAPI para concentrarse en el flujo.

FLUJO DE DATOS ENTRE CAPAS

01

Entrada y Validación

El cliente envía JSON, parámetros y cabeceras. El **Controller** los enlaza a un **Request DTO**, activa **@Valid**, delega al servicio y conserva el contexto HTTP.

02

Orquestación de Negocio

El **Service** procesa la lógica y gestiona la **Transacción**. Utiliza un mapeador para transformar el DTO en una **Entity**, asegurando que el dominio esté desacoplado de la API.

03

Acceso a Datos y Persistencia

El **Repository** recibe la entidad y gestiona la persistencia física en la **Base de Datos** mediante JPA/Hibernate, manteniendo la integridad referencial y el estado del objeto.

04

Transformación de Salida

La Entity recuperada se convierte en **Response DTO**. El Controller selecciona estado y cabeceras, serializa el DTO como JSON y evita exponer campos sensibles o asociaciones internas.

Regla de Oro: Las entidades son internas al dominio y la base de datos. Los DTOs son el contrato público. Nunca retorne entidades directamente en sus controladores REST para evitar fugas de información y acoplamiento excesivo.

LOMBOK

01 · CONFIGURACIÓN

Preparar el proyecto

Agregue Lombok como dependencia de compilación. La versión debe administrarse desde el proyecto; no se fija aquí.

POM.XML · FRAGMENTO

```xml
<dependency>
<groupId>org.projectlombok</groupId>
<artifactId>lombok</artifactId>
<scope>provided</scope>
</dependency>
```

IDE y compilación: active el procesamiento de anotaciones si el IDE lo requiere y confirme que el compilador reconoce el procesador.

USO SELECTIVO EN ENTIDAD Y SERVICIO

```java
@Entity @Getter
@NoArgsConstructor(access = PROTECTED)
@EqualsAndHashCode(onlyExplicitlyIncluded = true)
class Estudiante {
  @Id @GeneratedValue
@EqualsAndHashCode.Include private Long id;
  private String nombre;
}

@Service @RequiredArgsConstructor @Slf4j
class EstudianteService {
  private final EstudianteRepository repository;
}
```

02 · EXPLICACIÓN Y USO

Menos código, mismas decisiones

Lombok genera miembros durante la compilación; reduce código repetitivo, pero no sustituye el diseño del dominio ni las reglas de encapsulamiento.

`@Getter / @Setter`
Exponga solo los accesos necesarios.

`@NoArgsConstructor`
Constructor sin argumentos requerido por JPA.

`@RequiredArgsConstructor`
Inyección por constructor para campos *final*.

`@Builder`
Construcción legible, útil en DTO y pruebas.

`@Slf4j`
Logger tipado sin declaración manual.

`@EqualsAndHashCode`
Incluya únicamente campos estables y seguros.

- Prefiera **anotaciones selectivas** en lugar de generar todos los métodos automáticamente.
- Evite setters para identificadores, auditoría y relaciones que deban modificarse mediante métodos de dominio.
- Revise el código generado cuando cambien herencia, proxies de Hibernate o asociaciones bidireccionales.

Precaución con @Data en entidades JPA: combina getters, setters, `toString()`, `equals()` y `hashCode()`. Puede recorrer relaciones LAZY, producir ciclos, consultas inesperadas o una identidad inestable.

Criterio: use Lombok para eliminar boilerplate visible, no para ocultar invariantes, relaciones o decisiones de identidad de la entidad.

MODELMAPPER

01 · CONFIGURACIÓN

Mapeo estricto y controlado

Registre un único bean y haga explícitas las diferencias entre el contrato REST y el modelo persistente.

MODELMAPPERCONFIG.JAVA

```java
@Bean
ModelMapper modelMapper() {
  var mapper = new ModelMapper();
  mapper.getConfiguration()
      .setMatchingStrategy(MatchingStrategies.STRICT)
      .setSkipNullEnabled(true);

  mapper.typeMap(Estudiante.class,
                 EstudianteResponse.class)
      .addMapping(
          src -> src.getPrograma().getNombre(),
          EstudianteResponse::setPrograma);

  mapper.typeMap(EstudianteRequest.class,
                 Estudiante.class)
      .addMappings(m -> {
        m.skip(Estudiante::setId);
        m.skip(Estudiante::setCreadoEn);
      });
  return mapper;
}
```

- **IDs y auditoría:** no se copian desde la solicitud.
- **Asociaciones:** se resuelven en el servicio, no por coincidencia implícita.
- **Campos sensibles:** se excluyen mediante reglas o DTO separados.

02 · EXPLICACIÓN Y FLUJO

DTO ↔ Entity dentro del servicio

El controlador recibe y entrega DTO; el servicio mapea, aplica reglas y coordina la persistencia.

Request DTO datos validados › **Service** mapea y decide › **Entity** persistencia

EJEMPLO DE AMBOS SENTIDOS

```java
@Transactional
EstudianteResponse crear(EstudianteRequest request) {
  // DTO → Entity
var entity = mapper.map(
      request, Estudiante.class);
  entity.asignarPrograma(buscarPrograma(request.programaId()));

  var guardada = repository.save(entity);

  // Entity → Response DTO
return mapper.map(
      guardada, EstudianteResponse.class);
}
```

Pruebe el contrato de mapeo: valores, nulos, enums, fechas, colecciones y ambos sentidos. Verifique también que ID, auditoría y campos sensibles permanezcan intactos.

Evite confiar en mapeos implícitos no verificados. Para contratos críticos, transformaciones complejas o validación en compilación, prefiera **MapStruct** o mapeadores explícitos.

Criterio: ModelMapper reduce conversiones mecánicas; las asociaciones, reglas de negocio y decisiones de seguridad siguen perteneciendo al servicio.

MANEJO DE EXCEPCIONES

Flujo centralizado

Servicio lanza una **excepción de dominio** → **@RestControllerAdvice** la intercepta → **ProblemDetail** uniforme al cliente

Mapeo HTTP recomendado

Situación	Código
Datos inválidos / JSON incorrecto	400
Falta autenticación	401
Sin autorización	403
Recurso inexistente	404
Conflicto o regla de unicidad	409
Error no controlado	500

Jerarquía útil

RecursoNoEncontradoException, **ConflictoDominioException** y **ReglaNegocioException** expresan intención; las excepciones técnicas no se filtran al contrato REST.

APIEXCEPTIONHANDLER.JAVA

```java
@RestControllerAdvice
class ApiExceptionHandler {

  @ExceptionHandler(RecursoNoEncontradoException.class)
  ResponseEntity<ProblemDetail> noEncontrado(
      RecursoNoEncontradoException ex,
      HttpServletRequest request) {

    var problem = ProblemDetail.forStatusAndDetail(
        HttpStatus.NOT_FOUND, ex.getMessage());
    problem.setTitle("Recurso no encontrado");
    problem.setType(URI.create("https://api.ejemplo.edu/errors/not-found"));
    problem.setInstance(URI.create(request.getRequestURI()));
    problem.setProperty("timestamp", Instant.now());

    return ResponseEntity.status(404).body(problem);
  }

  @ExceptionHandler(MethodArgumentNotValidException.class)
  ResponseEntity<ProblemDetail> validacion(
      MethodArgumentNotValidException ex) {
    var problem = ProblemDetail.forStatus(400);
    problem.setTitle("Solicitud inválida");
    problem.setProperty("errors", camposInvalidos(ex));
    return ResponseEntity.badRequest().body(problem);
  }

  @ExceptionHandler(Exception.class)
  ResponseEntity<ProblemDetail> inesperado(Exception ex) {
    log.error("Error interno", ex); // solo en el servidor
var problem = ProblemDetail.forStatusAndDetail(
        HttpStatus.INTERNAL_SERVER_ERROR,
        "No fue posible procesar la solicitud");
    return ResponseEntity.status(500).body(problem);
  }
}
```

Seguridad: registre el detalle técnico con un identificador de correlación, pero no exponga trazas, SQL, nombres de tablas, credenciales, tokens ni mensajes internos en la respuesta.

PRUEBAS UNITARIAS Y SONARQUBE: INFORME ILUSTRATIVO

Validación Aislada (JUnit 5 + Mockito)

- Pruebas de unidad rápidas y deterministas.
- Mocking de dependencias (Repositories).
- Cobertura de caminos lógicos y excepciones.

```java
@ExtendWith(MockitoExtension.class)
class EstudianteServiceTest {
  @Mock private EstudianteRepository repo;
  @InjectMocks private EstudianteService service;

  @Test
  void crear_estudiante_valido() {
    // Arrange
    when(repo.save(any())).thenReturn(estudiante);
    // Act
    var res = service.crear(dto);
    // Assert
    assertNotNull(res.id());
    verify(repo, times(1)).save(any());
  }
}
```

COBERTURA DE CÓDIGO

84.2% A

DEUDA TÉCNICA

2d A

BUGS & VULNERABILIDADES

0 PASSED

CODE SMELLS

12 B

ID Caso	Objetivo de la Prueba	Resultado Esperado	Estado
UT-EST-001	Crear estudiante con programa válido	Persistencia exitosa y retorno ID	PASÓ
UT-EST-002	Error en programa inexistente	Lanza RecursoNoEncontradoException	PASÓ
UT-MAP-003	Mapeo de Entidad a DTO de salida	Excluye passwordHash del JSON	PASÓ
UT-VAL-004	Validación de nombre obligatorio	Falla si el nombre es nulo/vacío	PASÓ

PRUEBAS FUNCIONALES: CASOS E INFORME ILUSTRATIVO

Enfoque de Verificación

Valida el comportamiento del sistema desde el contrato HTTP. Evalúa la integración real entre controladores, servicios y base de datos (Testcontainers).

Herramientas Clave

Uso de **@SpringBootTest** para contexto completo y **MockMvc** para simular peticiones. Asegura fidelidad del entorno de ejecución.

ID	ENDPOINT / ACCIÓN	ENTRADA (PAYLOAD)	ESPERADO	OBTENIDO	ESTADO
FT-EST-001	`POST /api/v1/estudiantes`	JSON Estudiante Válido	201 Created + ID	201 Created + ID	PASÓ
FT-EST-002	`GET /api/v1/estudiantes/{id}`	ID Existente (ID: 1)	200 OK + Body JSON	200 OK + Body JSON	PASÓ
FT-EST-003	`GET /api/v1/estudiantes/{id}`	ID Inexistente (ID: 99)	404 Not Found	404 Not Found	PASÓ
FT-EST-004	`POST /api/v1/estudiantes`	Nombre: "" (Vacío)	400 Bad Request	400 Bad Request	PASÓ

Ambiente: Integración (H2/Testcontainers)

Resultados ilustrativos · sin ejecución real

Evidencia: Logs de consola y reportes Surefire

PRUEBAS DE CARGA: ESCENARIO E INFORME ILUSTRATIVO

Fundamentos y Herramientas

Objetivo: Evaluar estabilidad, throughput y latencia bajo perfiles de usuarios concurrentes.

Escenarios: Carga nominal, sostenida (soak), picos controlados (spike) y estrés.

k6 JMeter Gatling Prometheus

Métricas Clave

LATENCIA (P95/P99)
Tiempo de respuesta percentil

THROUGHPUT
Solicitudes por segundo (RPS)

TASA DE ERROR
% Fallos vs Exitosas

RECURSOS
CPU, Memoria y DB Pool

Informe de Escalabilidad (Ilustrativo)

ID ESCENARIO	USUARIOS (VUS)	THROUGHPUT	LATENCIA P95	ESTADO
LT-001: Carga Nominal	50	200 req/s	120 ms	**APROBADO**
LT-002: Sostenida	100	350 req/s	185 ms	**APROBADO**
LT-003: Pico Controlado	500	780 req/s	450 ms	**CONDICIONAL**
LT-004: Escritura Conc.	200	145 req/s	210 ms	**APROBADO**

** Los valores son ilustrativos. Un estado "Condicional" requiere análisis de cuellos de botella (N+1, índices o pool de conexiones).*

CHECKLIST Y ACTIVIDAD PRÁCTICA

BUENAS PRÁCTICAS

Principios de Diseño Sólido

Adoptar estas prácticas garantiza sistemas mantenibles, seguros y probables en cualquier equipo de desarrollo.

Separación estricta de capas
Controladores delegan; servicios orquestan; repositorios persisten.

DTO como contrato de API
Nunca exponer entidades; separar DTO de entrada y salida.

Fetch LAZY y control N+1
Usar JOIN FETCH o @EntityGraph solo cuando se necesita la asociación.

Transacciones explícitas
@Transactional en escritura; @Transactional(readOnly=true) en consultas.

Excepciones centralizadas
@RestControllerAdvice traduce a HTTP sin exponer trazas internas.

Credenciales externalizadas
Variables de entorno o vault; nunca tokens en código fuente ni slides.

Migraciones versionadas
Flyway o Liquibase; no usar ddl-auto=update en producción.

CHECKLIST DE ENTREGA

Verificación antes de liberar

- **Modelo relacional** e índices revisados
- **Entidades y relaciones** consistentes con mappedBy correcto
- **Fetch y consultas** revisados para eliminar N+1
- **DTO de entrada y salida** separados; campos sensibles excluidos
- **Reglas de negocio** y transacciones en capa de servicio
- **Controladores** con HTTP semántico y @Valid en entrada
- **Errores homogéneos** vía @RestControllerAdvice sin trazas
- **Pruebas unitarias** con Mockito y funcionales con MockMvc
- **Carga documentada** con p95, throughput y umbrales acordados
- **SonarQube integrado** en CI; cobertura y umbrales definidos

ACTIVIDAD PRÁCTICA

Modelar Estudiante, Programa, Asignatura | Repositorio con JPQL | DTO + Service + POST/GET | 2 pruebas unitarias | 2 pruebas funcionales | 1 escenario de carga