# **Alta de Propiedad — Paso a paso (Inyección de Dependencias + DAO + Factory Method)**

Te dejo un **resumen listo para pegar en un documento** (Drive). Va con explicación + estructura recomendada + snippets de código Java/Spring Boot que podés copiar.

## **Objetivo**

Implementar el caso de uso **Alta de Propiedad** en Spring Boot aplicando:

* **Inyección de Dependencias (IoC)** — Spring gestiona las dependencias.
* **DAO** — repositorios (JpaRepository) para persistencia.
* **Factory Method** — crear la entidad Propiedad (o su subclase) desde el DTO.

com.miapp.propiedades

├─ controller

├─ service

├─ repository

├─ model (entities)

├─ dto

└─ factory

## **Paso a paso**

### **1) Modelo / Entidades (JPA)**

Define Propiedad y Propietario (mínimo necesario para el alta).

// model/Propietario.java

@Entity

public class Propietario {

@Id @GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String nombre;

private String cbu;

// getters/setters, constructor vacío

}

// model/Propiedad.java

@Entity

public class Propiedad {

@Id @GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String direccion;

private Boolean enAlquiler;

private Double precio;

// otros campos...

@ManyToOne(fetch = FetchType.LAZY)

@JoinColumn(name = "propietario\_id")

private Propietario propietario;

// getters/setters, constructor vacío

}

### **2) DTO para la entrada (no exponer entidad directamente)**

// dto/PropiedadDTO.java

public class PropiedadDTO {

private String direccion;

private Boolean enAlquiler;

private Double precio;

private Long propietarioId; // vínculo por id

// getters/setters

}

### **3) DAO (Repositorios)**

Usá Spring Data JPA: JpaRepository implementa el patrón DAO.

// repository/PropietarioRepository.java

public interface PropietarioRepository extends JpaRepository<Propietario, Long> {}

// repository/PropiedadRepository.java

public interface PropiedadRepository extends JpaRepository<Propiedad, Long> {}

### **4) Factory Method (crear Propiedad desde DTO)**

Fábrica simple para encapsular la creación (si necesitás subclases, la fábrica devuelve la subclase correcta).

// factory/PropiedadFactory.java

public class PropiedadFactory {

public static Propiedad crearDesdeDTO(PropiedadDTO dto, Propietario propietario) {

Propiedad p = new Propiedad();

p.setDireccion(dto.getDireccion());

p.setEnAlquiler(dto.getEnAlquiler());

p.setPrecio(dto.getPrecio());

p.setPropietario(propietario);

// mapear otros campos según corresponda

return p;

}

}

Si tenés tipos (COMERCIAL / FAMILIAR) podés ampliar la fábrica para devolver new Comercial() o new Familiar() según dto.getTipo().

### **5) Service layer (inyección de dependencias + lógica de negocio)**

Usá constructor injection. Añadí @Transactional si corresponde.

// service/PropiedadService.java

@Service

public class PropiedadService {

private final PropiedadRepository propiedadRepo;

private final PropietarioRepository propietarioRepo;

public PropiedadService(PropiedadRepository propiedadRepo,

PropietarioRepository propietarioRepo) {

this.propiedadRepo = propiedadRepo;

this.propietarioRepo = propietarioRepo;

}

@Transactional

public Propiedad altaPropiedad(PropiedadDTO dto) {

Propietario prop = propietarioRepo.findById(dto.getPropietarioId())

.orElseThrow(() -> new EntityNotFoundException("Propietario no existe"));

Propiedad propiedad = PropiedadFactory.crearDesdeDTO(dto, prop);

return propiedadRepo.save(propiedad);

}

}

### **6) Controller (endpoint REST)**

Controller minimalista que delega al servicio.

// controller/PropiedadController.java

@RestController

@RequestMapping("/propiedades")

public class PropiedadController {

private final PropiedadService propiedadService;

public PropiedadController(PropiedadService propiedadService) {

this.propiedadService = propiedadService;

}

@PostMapping

public ResponseEntity<Propiedad> crear(@RequestBody @Valid PropiedadDTO dto) {

Propiedad creada = propiedadService.altaPropiedad(dto);

return ResponseEntity.status(HttpStatus.CREATED).body(creada);

}

}

## **Notas prácticas y buenas prácticas**

* **Validaciones:** Pon @Valid en el DTO y usa @NotNull, @Size, etc. para validar entrada. Centraliza validaciones en el *service* o en @ControllerAdvice.
* **Transaccionalidad:** Anotá métodos de servicio con @Transactional cuando el alta implique varias operaciones.
* **Errores:** Usa excepciones específicas y @ControllerAdvice para mapearlas a respuestas HTTP claras.
* **Factory ampliable:** Si implementás subtipos (Comercial/Familiar), que la fábrica encapsule la lógica de creación para no ensuciar el servicio.
* **Testabilidad:** Constructor injection facilita mocks en tests unitarios.
* **DTO ↔ Entity mapping:** Para proyectos más grandes, considerá usar MapStruct o un Mapper manual.
* **Logging:** Usa SLF4J para logs (LoggerFactory.getLogger(...)).

## **Resumen (1-2 líneas para la intro del documento)**

Aplicamos **Inyección de Dependencias** (Spring maneja el wiring), **DAO** (Spring Data JPA para acceder a la BD) y **Factory Method** (para encapsular la creación de la entidad Propiedad desde el DTO). La arquitectura resultante separa responsabilidades: **Controller → Service → Factory → Repository**.

## **Checklist para entregar**

* Entidades Propiedad y Propietario con mapeos JPA.
* PropiedadDTO con validaciones.
* PropietarioRepository y PropiedadRepository.
* PropiedadFactory implementado y probado.
* PropiedadService con constructor injection y @Transactional.
* PropiedadController con POST /propiedades.
* Tests unitarios para el Service (mockear repositorios).
* Documentación breve y diagrama de clases/secuencia (opcional).