

Guía Completa de Arquitectura Spring Boot - ApiNoe










Índice

1. Visión General
 2. Estructura del Proyecto
 3. Capas de la Aplicación
 4. Arquitectura Base (Genérica)
 5. Flujo de Datos
 6. Configuraciones
 7. Patrones Implementados
 8. Entidades y Relaciones
 9. Endpoints y API
 10. Manejo de Errores
 11. Seguridad
 12. Testing
-

1. Visión General

ApiNoe es una API REST desarrollada con Spring Boot que implementa una arquitectura escalable basada en capas, utilizando patrones genéricos reutilizables para minimizar la duplicación de código y facilitar el mantenimiento.

Características Principales:

-  **Arquitectura en capas** (Controller → Service → Repository)
 -  **Clases base genéricas** para operaciones CRUD
 -  **DTOs** para transferencia de datos
 -  **Mappers** para conversión entre entidades y DTOs
 -  **Manejo centralizado de errores**
 -  **Validaciones automáticas**
 -  **Seguridad con OAuth2/JWT**
 -  **Configuración CORS**
 -  **Transacciones automáticas**
-

2. Estructura del Proyecto

```
src/main/java/com/noe/apinoe/
├── config/                # Configuraciones
│   ├── CorsConfig.java
│   ├── DatabaseConfig.java
│   ├── SecurityConfig.java
│   └── GlobalExceptionHandler.java
├── controller/           # Controladores REST
│   ├── base/
│   │   └── BaseController.java
│   ├── UsuarioController.java
│   ├── ProductoController.java
│   └── ...
├── dto/                  # Data Transfer Objects
│   ├── ApiResponse.java
│   ├── UsuarioDto.java
│   └── ...
├── mapper/               # Conversores Entity ↔ DTO
│   ├── BaseMapper.java
│   ├── UsuarioMapper.java
│   └── ...
├── model/                # Entidades JPA
│   ├── Usuario.java
│   ├── Producto.java
│   └── ...
├── repository/           # Repositorios JPA
│   ├── UsuarioRepository.java
│   ├── ProductoRepository.java
│   └── ...
├── service/              # Lógica de negocio
│   ├── BaseService.java
│   ├── UsuarioService.java
│   ├── impl/
│   │   ├── UsuarioServiceImpl.java
│   │   └── ...
│   └── ...
└── ApinoeApplication.java # Clase principal
```

3. Capas de la Aplicación

3.1 Capa de Presentación (Controllers)

Responsabilidad: Recibir peticiones HTTP, validar entrada, delegar a servicios y devolver respuestas.

java

```
@RestController
@RequestMapping("/api/usuarios")
public class UsuarioController extends BaseController<Usuario, UsuarioDto, Integer>
```

Características:

- Extienden `BaseController` para operaciones CRUD básicas
- Implementan endpoints específicos de cada entidad
- Manejan validaciones con `@Valid`
- Devuelven respuestas consistentes con `ApiResponse<T>`

3.2 Capa de Servicio (Services)

Responsabilidad: Contener la lógica de negocio, validaciones complejas y orquestar operaciones.

```
java

public interface UsuarioService extends BaseService<Usuario, Integer>
public class UsuarioServiceImpl implements UsuarioService
```

Características:

- Interfaces que extienden `BaseService`
- Implementaciones con `@Transactional`
- Validaciones de negocio
- Manejo de excepciones específicas

3.3 Capa de Persistencia (Repositories)

Responsabilidad: Acceso a datos, consultas personalizadas.

```
java

public interface UsuarioRepository extends JpaRepository<Usuario, Integer>
```

Características:

- Extienden `JpaRepository<Entity, ID>`
- Métodos de consulta derivados
- Consultas personalizadas con `@Query`

3.4 Capa de Transferencia (DTOs)

Responsabilidad: Transferir datos entre capas y hacia el cliente.

```
java
```

```
public class UsuarioDto {  
    @NotBlank private String nombre;  
    @Email private String email;  
}
```

3.5 Capa de Mapeo (Mappers)

Responsabilidad: Convertir entre entidades y DTOs.

```
java
```

```
public class UsuarioMapper implements BaseMapper<Usuario, UsuarioDto>
```

4. Arquitectura Base (Genérica)

4.1 BaseService<E, ID>

Interface genérica que define operaciones CRUD básicas:

```
java
```

```
public interface BaseService<E, ID> {  
    List<E> findAll();  
    Optional<E> findById(ID id);  
    E save(E entity);  
    E update(ID id, E entity);  
    void deleteById(ID id);  
    boolean existsById(ID id);  
}
```

4.2 BaseMapper<E, D>

Interface genérica para conversiones:

```
java
```

```
public interface BaseMapper<E, D> {  
    D toDto(E entity);  
    E toEntity(D dto);  
    void updateEntityFromDto(E entity, D dto);  
}
```

4.3 BaseController<E, D, ID>

Controlador genérico con operaciones CRUD:

```
java

public abstract class BaseController<E, D, ID> {
    // GET /api/entidades
    public ResponseEntity<ApiResponse<List<D>>> getAll()

    // GET /api/entidades/{id}
    public ResponseEntity<ApiResponse<D>> getById(@PathVariable ID id)

    // POST /api/entidades
    public ResponseEntity<ApiResponse<D>> create(@Valid @RequestBody D dto)

    // PUT /api/entidades/{id}
    public ResponseEntity<ApiResponse<D>> update(@PathVariable ID id, @Valid @RequestBody D dto)

    // DELETE /api/entidades/{id}
    public ResponseEntity<ApiResponse<String>> delete(@PathVariable ID id)
}
```

4.4 ApiResponse<T>

Wrapper genérico para respuestas consistentes:

```
java

public class ApiResponse<T> {
    private boolean success;
    private String message;
    private T data;
    private LocalDateTime timestamp;
}
```

5. Flujo de Datos

5.1 Flujo de Lectura (GET)

```
Cliente HTTP Request → Controller → Service → Repository → Base de Datos
                        ↓
Cliente HTTP Response ← DTO ← Mapper ← Entity ← Repository
```

5.2 Flujo de Escritura (POST/PUT)

Cliente HTTP Request → Controller → Validación → DTO → Mapper → Entity



Cliente HTTP Response ← DTO ← Mapper ← Entity ← Repository ← Base de Datos

5.3 Ejemplo Completo - Crear Usuario

1. **Cliente** envía POST `/api/usuarios` con JSON
2. **UsuarioController** recibe `UsuarioDto` y valida con `@Valid`
3. **BaseController.create()** llama `validateBeforeCreate()`
4. **UsuarioMapper.toEntity()** convierte DTO → Usuario
5. **UsuarioService.save()** aplica lógica de negocio
6. **UsuarioRepository.save()** persiste en BD
7. **UsuarioMapper.toDto()** convierte Usuario → DTO
8. **BaseController** devuelve `ApiResponse<UsuarioDto>`

6. Configuraciones

6.1 CorsConfig.java

Configuración para permitir peticiones desde frontend:

```
java

@Configuration
public class CorsConfig implements WebMvcConfigurer {
    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/api/**")
            .allowedOrigins("*")
            .allowedMethods("GET", "POST", "PUT", "DELETE");
    }
}
```

6.2 DatabaseConfig.java

Configuración de base de datos y JPA:

```
@Configuration
@EnableJpaRepositories
@EnableTransactionManagement
public class DatabaseConfig {
    // Configuración de DataSource, EntityManager, etc.
}
```

6.3 SecurityConfig.java

Configuración de seguridad (OAuth2/JWT):

```
java

@Configuration
@EnableWebSecurity
public class SecurityConfig {
    // Configuración de autenticación y autorización
}
```

6.4 GlobalExceptionHandler.java

Manejo centralizado de excepciones:

```
java

@ControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(ValidationException.class)
    public ResponseEntity<ApiResponse<String>> handleValidation(ValidationException e);
}
```

7. Patrones Implementados

7.1 Repository Pattern

- Abstrae el acceso a datos
- Métodos derivados de Spring Data JPA
- Consultas personalizadas

7.2 Service Layer Pattern

- Encapsula lógica de negocio
- Maneja transacciones
- Valida reglas de negocio

7.3 DTO Pattern

- Transfiere datos entre capas
- Evita exponer entidades directamente
- Permite validaciones específicas

7.4 Mapper Pattern

- Convierte entre diferentes representaciones
- Centraliza lógica de conversión
- Facilita mantenimiento

7.5 Template Method Pattern

- `BaseController` define estructura común
 - Controladores específicos implementan detalles
 - Métodos hook para personalización
-

8. Entidades y Relaciones

8.1 Estructura Base de Entidades

```
java
@Entity
public class BaseEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private LocalDateTime fechaCreacion;
    private LocalDateTime fechaActualizacion;
    private Boolean activo;
}
```

8.2 Entidades del Dominio

- **Usuario:** Gestión de usuarios
- **Producto:** Catálogo de productos
- **Proyecto:** Gestión de proyectos
- **Tecnología:** Tecnologías utilizadas
- **ProyectoTecnología:** Relación muchos a muchos
- **Almacen:** Gestión de inventario

8.3 Tipos de Relaciones

- **OneToMany**: Usuario → Proyectos
 - **ManyToOne**: Proyecto → Usuario
 - **ManyToMany**: Proyecto ↔ Tecnología
 - **OneToOne**: Usuario → Perfil (si aplica)
-

9. Endpoints y API

9.1 Estructura de URLs

```
/api/{entidad}          # CRUD básico
/api/{entidad}/{id}     # Por ID
/api/{entidad}/activos  # Filtros específicos
/api/{entidad}/buscar   # Búsquedas
/api/{entidad}/{id}/activar # Acciones específicas
```

9.2 Códigos de Respuesta HTTP

- **200 OK**: Operación exitosa
- **201 Created**: Recurso creado
- **400 Bad Request**: Error de validación
- **404 Not Found**: Recurso no encontrado
- **500 Internal Server Error**: Error del servidor

9.3 Formato de Respuesta

```
json

{
  "success": true,
  "message": "Usuario creado exitosamente",
  "data": { "id": 1, "nombre": "Juan" },
  "timestamp": "2024-01-15T10:30:00"
}
```

10. Manejo de Errores

10.1 Tipos de Excepciones

- **ValidationException**: Errores de validación
- **ResourceNotFoundException**: Recurso no encontrado

- **BusinessRuleException:** Violación de reglas de negocio
- **DataIntegrityException:** Violación de integridad

10.2 Respuestas de Error

```
json
{
  "success": false,
  "message": "Usuario no encontrado con id: 123",
  "data": null,
  "timestamp": "2024-01-15T10:30:00"
}
```

11. Seguridad

11.1 Autenticación

- **OAuth2** con Google
- **JWT** para sesiones
- **Refresh tokens**

11.2 Autorización

- Roles de usuario (ADMIN, USER)
- Endpoints protegidos
- Validación de permisos

11.3 Configuración CORS

- Orígenes permitidos
- Métodos HTTP habilitados
- Headers personalizados

12. Testing

12.1 Tipos de Tests

- **Unit Tests:** Servicios y mappers
- **Integration Tests:** Controladores
- **Repository Tests:** Acceso a datos

12.2 Herramientas

- **JUnit 5:** Framework de testing
 - **Mockito:** Mocking
 - **TestContainers:** Tests de integración
 - **WebMvcTest:** Tests de controladores
-

13. Ventajas de esta Arquitectura

13.1 Escalabilidad

- Fácil agregar nuevas entidades
- Reutilización de código base
- Estructura predecible

13.2 Mantenibilidad

- Separación clara de responsabilidades
- Código DRY (Don't Repeat Yourself)
- Patrones consistentes

13.3 Testabilidad

- Inyección de dependencias
- Interfaces bien definidas
- Mocking sencillo

13.4 Flexibilidad

- Personalización por entidad
 - Métodos hook para lógica específica
 - Configuración centralizada
-

14. Próximos Pasos

Para implementar esta arquitectura:

1. **Definir entidades** con sus columnas y relaciones
2. **Crear interfaces base** (BaseService, BaseMapper)
3. **Implementar clase BaseController**
4. **Configurar base de datos y JPA**
5. **Crear servicios específicos** para cada entidad
6. **Implementar mappers** para conversiones

7. **Configurar seguridad** y CORS
 8. **Agregar manejo de errores**
 9. **Escribir tests** para cada capa
 10. **Documentar API** con Swagger
-

15. Comandos para Empezar

```
bash
```

```
# 1. Crear proyecto Spring Boot
```

```
spring init --dependencies=web,data-jpa,mysql,security apinoe
```

```
# 2. Estructura de directorios
```

```
mkdir -p src/main/java/com/noe/apinoe/{config,controller/base,dto,mapper,model,repository,service/impl}
```

```
# 3. Configurar application.yml
```

```
# 4. Crear clases base
```

```
# 5. Implementar primera entidad (Usuario)
```

```
# 6. Testear endpoints básicos
```

Esta guía te servirá como referencia completa. **¿Por dónde quieres que empecemos?** ¿Defines primero las entidades con sus columnas y relaciones, o prefieres que creamos las clases base genéricas?