

Spring: aspectos no funcionales

Integrando requisitos no funcionales en nuestros backends



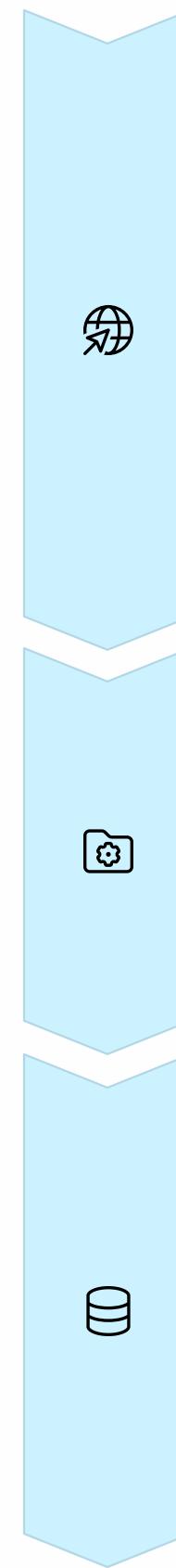
por **Ivan Ruiz Rube**



Arquitectura basada en capas



Spring Boot



Capa de presentación

Los **controladores** REST reciben peticiones HTTP y devuelven respuestas. Esta capa también puede estar formada por controladores gRPC, GraphQL, aplicaciones de escritorio, aplicaciones de consola (CLI) o aplicaciones web (HTML, plantillas JSP, etc.)

En esta capa podemos implementar patrones como **DTO** y **DataMappers** para gestionar la transferencia de datos.

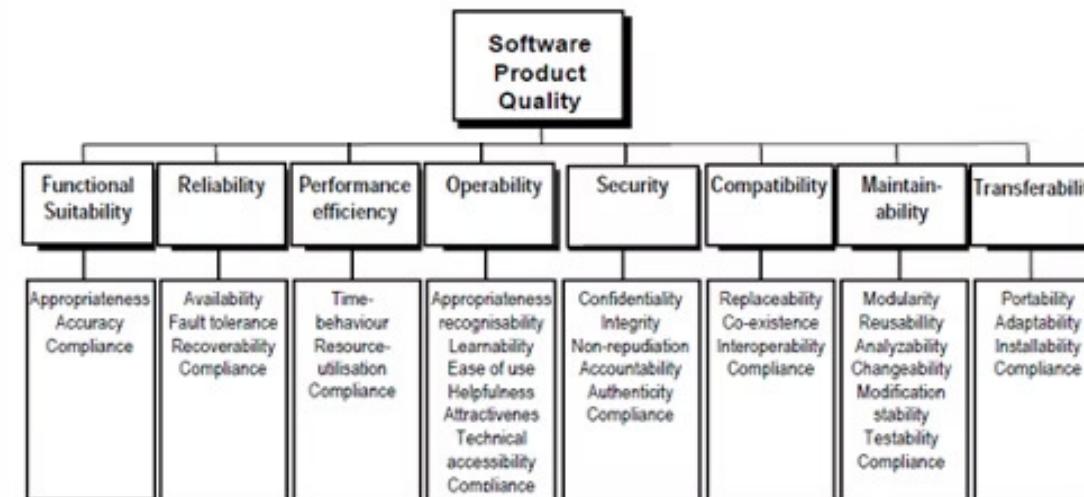
Capa de negocio

Los **servicios** implementan los casos de uso, garantizando el cumplimiento de las reglas de negocio, mientras que las **entidades** representan los elementos de información fundamentales de nuestro modelo de dominio.

Capa de datos/infraestructura

Los **repositorios** permiten acceder a la base de datos. Esta capa también incluye otros componentes como: APIs de terceros, servicios de notificaciones (push, SMS, emails), servicios de mapas, sistemas de almacenamiento masivo, pasarelas de pago, colas de mensajería y motores de búsqueda.

Atributos de calidad del software



ISO/IEC CD 25010 Software engineering -- Software product Quality Requirements and Evaluation (SQuaRE)

Agenda

1. Intro
2. Docs
3. Seguridad
4. Validaciones
5. Pruebas
6. Transacciones
7. Caché
8. Clientes HTTP
9. Perfiles
10. Observabilidad
11. Otros

Docs

La documentación clara y completa constituye un pilar fundamental para el éxito y la sostenibilidad de cualquier proyecto de software empresarial. Podemos distinguir varios tipos de documentación esenciales:

Documentación de requisitos

Define las necesidades y expectativas del cliente, estableciendo el alcance del proyecto y sirviendo como contrato entre los stakeholders y el equipo de desarrollo.

Documentación técnica

Describe la arquitectura, patrones de diseño, decisiones técnicas y componentes del sistema para facilitar el mantenimiento y evolución del software por parte del equipo técnico.

Documentación de usuario

Proporciona guías, manuales y tutoriales que permiten a los usuarios finales entender y aprovechar todas las funcionalidades del sistema de manera efectiva.

Documentación de API

Permite a equipos externos entender cómo consumir e interactuar con nuestras APIs locales o remotas, incluyendo endpoints disponibles, formatos de datos, autenticación y ejemplos de uso.

Docs: API



Facilita el consumo

Una API bien documentada permite a otros desarrolladores entender cómo utilizarla correctamente

The screenshot shows the Swagger UI interface with the following details:

- Header:** Shows the Swagger logo and the URL `/api-docs`.
- Section:** **OpenAPI definition** (version 3.0.1) is selected.
- Servers:** A dropdown menu shows the selected server as `http://localhost:8080 - Generated server url`.
- Default Route:** `/api/book/{id}` is highlighted in blue.
- Operations:** A list of methods for the `/api/book/{id}` endpoint:
 - PUT /api/book/{id}
 - DELETE /api/book/{id}
 - PATCH /api/book/{id}
 - GET /api/book/
 - POST /api/book/



Estándar de facto

OpenAPI (anteriormente Swagger) se ha convertido en el estándar para describir API REST



Audiencia dual

Documentación legible tanto para humanos como para máquinas

Docs: beneficios de Swagger UI / OpenAPI



Documentación viva

Siempre actualizada con el código actual



Pruebas interactivas

Permite probar la API directamente desde el navegador



Generación de clientes

La especificación OpenAPI puede usarse para generar clientes (SDK) automáticamente



API Management

Integración con servicios de gestión de APIs

Docs: OpenAPI Specification

Formato YAML/JSON

Describe de forma estructurada:

- Endpoints disponibles
- Métodos HTTP soportados
- Parámetros de entrada
- Esquemas de datos
- Códigos de respuesta
- Ejemplos de uso

Docs: Uso de springdoc-openapi

Agregar dependencia

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.8.8</version>
</dependency>
```

Anotaciones

Enriquecer el código de los controladores con anotaciones @Operation, etc. Si no se agregan las anotaciones, springdoc tratará de inferir la mayor parte de información de los tipos de datos y las convenciones

Documentación automática

Al ejecutar la aplicación se genera:

- Documento OpenAPI en JSON (/v3/api-docs)
- Interfaz web interactiva (/swagger-ui.html)

Docs: ejemplo

Identificación del API

```
@Bean
public OpenAPI customOpenAPI() {

    return new OpenAPI().info(new Info()
        .title("API de la Banca Online UCA")
        .version("v1")
        .termsOfService("https://www.uca.es/aviso-legal/")
        .description("API REST para la gestión de clientes, cuentas y movimientos en una aplicación de banca
online.")
    );
}
```

Docs: ejemplo

Describiendo operaciones

```
@RestController
@RequestMapping("/api/cuentas")
@Tag(name = "Cuentas", description = "Operaciones relacionadas con cuentas")
public class CuentaController {

    ...

    @PostMapping("/")
    @ResponseStatus(org.springframework.http.HttpStatus.CREATED)
    @Operation(summary = "Crear cuenta", description = "Crea una nueva cuenta para el cliente especificado")
    public CuentaDTO post(@RequestBody Long clientelid) {
        ...
    }

    ...
}
```

Docs: ejemplo (II)

Describiendo esquemas de datos

```
@Schema(description = "Representa una cuenta del cliente")
public record CuentaDTO(
    @Schema(description = "Número de cuenta", example = "123")
    String numero,
    @Schema(description = "Disponible", example = "1000.50")
    BigDecimal saldo,
    @Schema(description = "Identificador del cliente", example = "112233")
    Long clientelid)
{}
```

Seguridad

Top 10 Controles Proactivos 2018

Controles de seguridad esenciales recomendados por OWASP para que los desarrolladores los integren en cada proyecto de software para mejorar la protección contra vulnerabilidades comunes.

Definir los requisitos de seguridad

- 1 Establecer objetivos de seguridad claros al inicio del desarrollo. Incluir tanto las características de seguridad funcionales como los requisitos no funcionales como los estándares de cumplimiento.

Aprovechar los frameworks y bibliotecas de seguridad

- 2 Utilizar marcos de seguridad establecidos y actualizados en lugar de implementar mecanismos de seguridad desde cero para evitar errores de implementación comunes.

Asegurar el acceso a la base de datos

- 3 Implementar una autenticación adecuada, conexiones cifradas y consultas parametrizadas para proteger contra la inyección SQL y el acceso no autorizado.

Codificar y escapar datos

- 4 Aplicar la codificación y el escape de salida adecuados para evitar ataques de inyección como XSS, asegurándose de que la entrada del usuario no se interprete como código ejecutable.

Validar todas las entradas

- 5 Verificar que todos los datos de fuentes externas cumplan con el formato, la longitud y los criterios de rango esperados antes de procesarlos para evitar ataques.

Implementar identidad digital

- 6 Crear controles robustos de autenticación y gestión de sesiones. Utilizar autenticación de varios factores y políticas de contraseñas seguras.

Hacer cumplir los controles de acceso

- 7 Aplicar el principio del privilegio mínimo, implementar el acceso basado en roles y verificar la autorización en el servidor para cada solicitud.

Proteger los datos en todas partes

- 8 Implementar cifrado para los datos en reposo y en tránsito. Clasificar los datos por sensibilidad y aplicar las medidas de protección apropiadas.

Implementar registro y monitoreo de seguridad

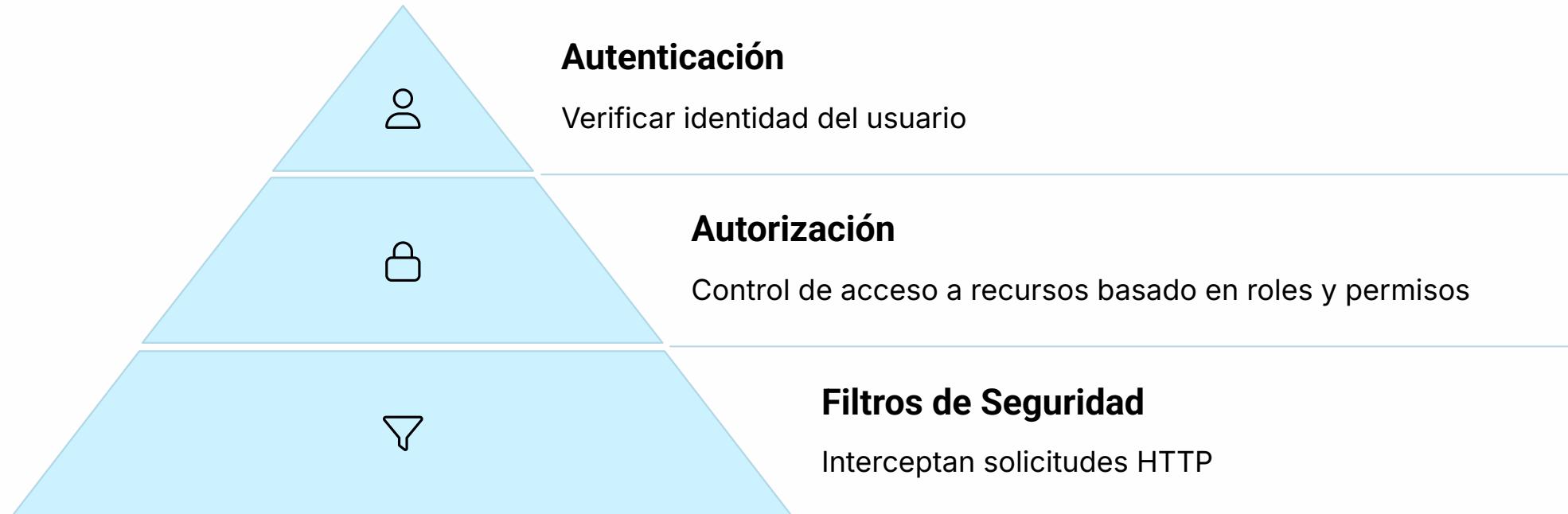
- 9 Registrar los eventos relevantes para la seguridad y establecer un monitoreo para detectar actividades sospechosas y apoyar la respuesta a incidentes.

Manejar todos los errores y excepciones

- 10 Implementar un manejo adecuado de excepciones para evitar fugas de información. Crear mensajes de error significativos para los usuarios sin exponer detalles confidenciales.



Seguridad: Spring Security



Con solo incluir la dependencia **spring-boot-starter-security**, nuestra aplicación Spring Boot queda protegida por defecto.

Seguridad: características de Spring Security

Spring Security proporciona una capa robusta de seguridad para aplicaciones empresariales

- Autenticación flexible (LDAP, JDBC, OAuth2, etc.)
- Autorización a nivel de método y URL
- Protección contra ataques comunes (CSRF, XSS)
- Integración con frameworks web
- Compatible con el cifrado de comunicaciones (HTTPS)
- Soporta CORS para configurar para APIs consumidas desde otros dominios

Seguridad: autenticación en Spring Security



Verificación de Identidad

Spring Security abstrae este proceso mediante la interfaz *UserDetailsService*, que actúa como puente entre tu sistema de autenticación y Spring. Esta interfaz define el método *loadUserByUsername* que transforma un identificador de usuario en un objeto *UserDetails* completo con sus roles y permisos asociados.



Almacenamiento de Usuarios

Ofrece múltiples mecanismos de almacenamiento: en memoria para entornos de desarrollo (*InMemoryUserDetailsManager*), en base de datos relacional con JDBC utilizando contraseñas cifradas. Puede extenderse completamente implementando la interfaz *UserDetailsService* para integrarse con cualquier modelo de datos o servicio de autenticación existente.



Autenticación local

Soporta diversos métodos nativos como autenticación básica HTTP, formularios personalizados de login y tokens JWT para APIs, permitiendo elegir el enfoque más adecuado según los requisitos de la aplicación.



Autenticación Externa

Integra OAuth2/OIDC para implementar login social o SSO corporativo, y soporta JWT para APIs sin estado (stateless). Facilita la conexión con proveedores como Google, GitHub o Facebook, así como con servidores de identidad empresariales mediante configuraciones declarativas simplificadas.



Personalización del Proceso

Permite configurar cada aspecto del flujo de autenticación: desde formularios de login a medida, gestión de eventos de autenticación exitosa o fallida, hasta reglas sofisticadas de expiración de sesiones y aplicación de políticas robustas de seguridad de contraseñas.

Seguridad: autorización en Spring Security

Control de Acceso

Determinar qué recursos o acciones puede realizar un usuario autenticado. Se basa en roles o privilegios (authorities).

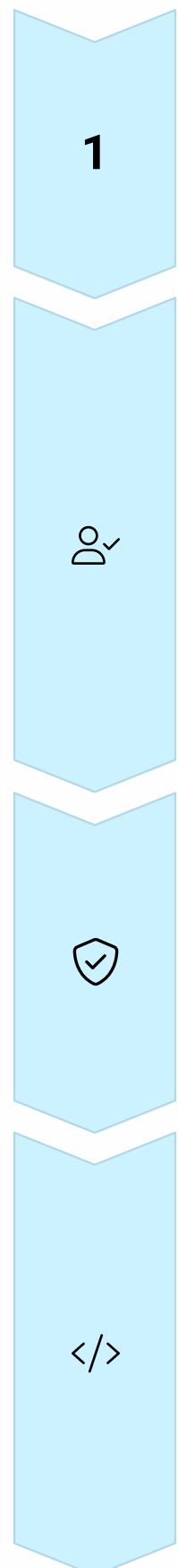
URL protegidas

Spring Security permite definir reglas de acceso a URL.

Restricción por Método

Es posible restringir ciertos métodos de negocio a roles específicos, complementando la seguridad declarada en los endpoints. Con `@EnableMethodSecurity` (en Spring Security 6) se activan anotaciones como `@PreAuthorize` en métodos de servicio/controlador para autorización a nivel de método.

Seguridad: filtros en Spring Security



El servidor HTTP interceptan las peticiones y las derivan a una cadena de filtros secuenciales de Spring Security para su procesamiento.

Filtro de Autenticación

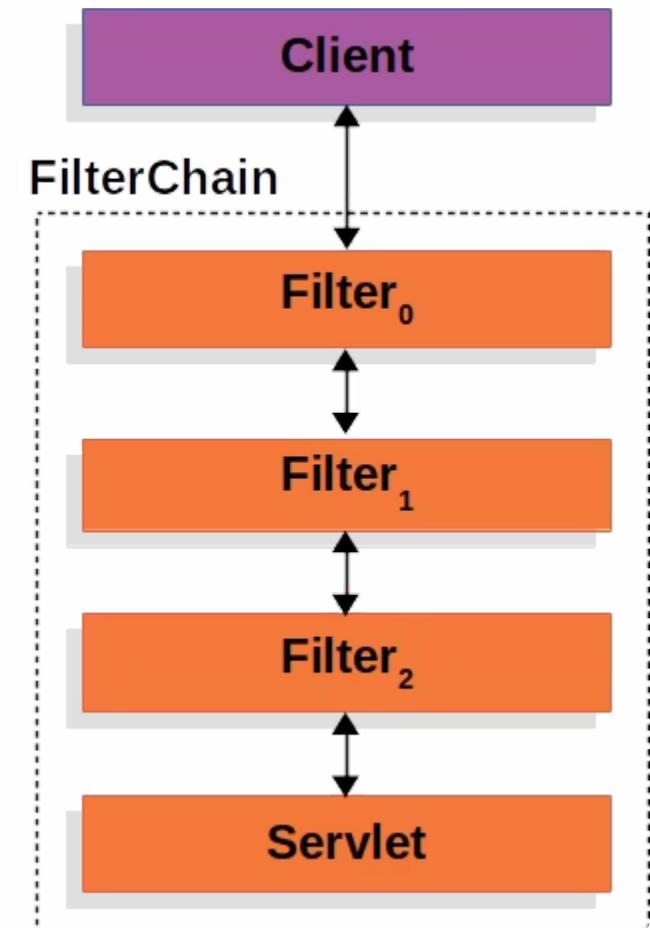
El proceso inicia cuando un AuthenticationFilter captura las credenciales (como tokens JWT en cabeceras Authorization), genera un token de autenticación y lo transmite al AuthenticationManager, que delega en los AuthenticationProviders configurados para validar la identidad del usuario y establecer el contexto de seguridad cuando las credenciales son válidas.

Filtro de Autorización

Una vez autenticado, este filtro evalúa si el usuario posee los permisos necesarios para acceder al recurso solicitado, verificando sus roles y privilegios contra las políticas de seguridad definidas.

Controlador

Todo este proceso de verificación y autorización se completa antes de que la petición alcance nuestro controlador (internamente implementados como Java Servlets), garantizando que solo las solicitudes debidamente autenticadas y autorizadas sean procesadas por la lógica de negocio.



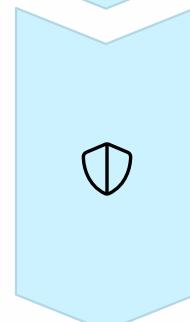
Seguridad: configuración de Spring Security



Configurable

1

Configuración mediante código Java usando la API de HttpSecurity para definir un bean SecurityFilterChain en una clase @Configuration. También admite propiedades externas (application.properties)



Seguridad por Defecto



Sin ninguna configuración adicional, Spring Boot arranca con seguridad básica: crea un usuario predeterminado (user) con contraseña aleatoria (imprimida en el log al inicio) y protege todas las rutas con autenticación HTTP Basic por defecto.



Personalización



Permite especificar qué rutas exigirán autenticación, cuáles estarán abiertas, el método de autenticación, personalización de la página/formulario de login, el manejo de logout, la estrategia de gestión de sesiones, políticas de CSRF (deshabilitadas en API REST porque usualmente usan tokens y no sesiones web tradicionales) y de CORS (para APIs)

Seguridad: ejemplo de autenticación basic

La autenticación Basic es un mecanismo simple de seguridad HTTP donde las credenciales (usuario y contraseña) se envían codificadas en Base64 en el encabezado de la petición.

En este ejemplo configuramos Spring Security para utilizar autenticación Basic, definimos un usuario en memoria para pruebas. En producción, se recomienda utilizar un UserDetailsService personalizado conectado a una base de datos.

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
        .csrf(csrf -> csrf.disable())
        .httpBasic(Customizer.withDefaults());
    return http.build();
}

@Bean
public UserDetailsService userDetailsService() {
    // Crear un usuario con rol USER para pruebas
    UserDetails user = User.builder()
        .username("usuario")
        .password(passwordEncoder().encode("usuario"))
        .roles("USER")
        .build();

    // Añadir un usuario con rol ADMIN para pruebas
    UserDetails admin = User.builder()
        .username("admin")
        .password(passwordEncoder().encode("admin"))
        .roles("ADMIN")
        .build();

    // Creamos un InMemoryUserDetailsManager con los usuarios
    InMemoryUserDetailsManager userDetailsManager = new InMemoryUserDetailsManager();
    userDetailsManager.createUser(user);
    userDetailsManager.createUser(admin);
    return userDetailsManager;
}

@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```

Seguridad: ejemplo de control de acceso a URL

```
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {  
    http  
        .csrf(csrf -> csrf.disable())  
        .authorizeHttpRequests(auth -> auth  
            .requestMatchers("/swagger-ui.html",  
                "/swagger-ui/**",  
                "/v3/api-docs/**").permitAll()  
            .requestMatchers("/api/v1/clientes/**").hasRole("ADMIN")  
            .anyRequest().authenticated()  
        )  
        .httpBasic(Customizer.withDefaults());  
  
    return http.build();  
}
```

Seguridad: ejemplo de control de acceso a método

```
// añadir @EnableMethodSecurity en la clase @SpringBootApplication

@Service
public class CuentaService {

    ...
    @PreAuthorize("hasRole('ADMIN')")
    public void eliminarCuenta(Long id) {
        Cuenta cuenta = obtenerCuenta(id);
        cuentaRepo.delete(cuenta);
    }

    ...
}

}
```

Validaciones

Las validaciones garantizan la integridad y coherencia de los datos que fluyen entre las capas de nuestra aplicación.

Capa de Presentación

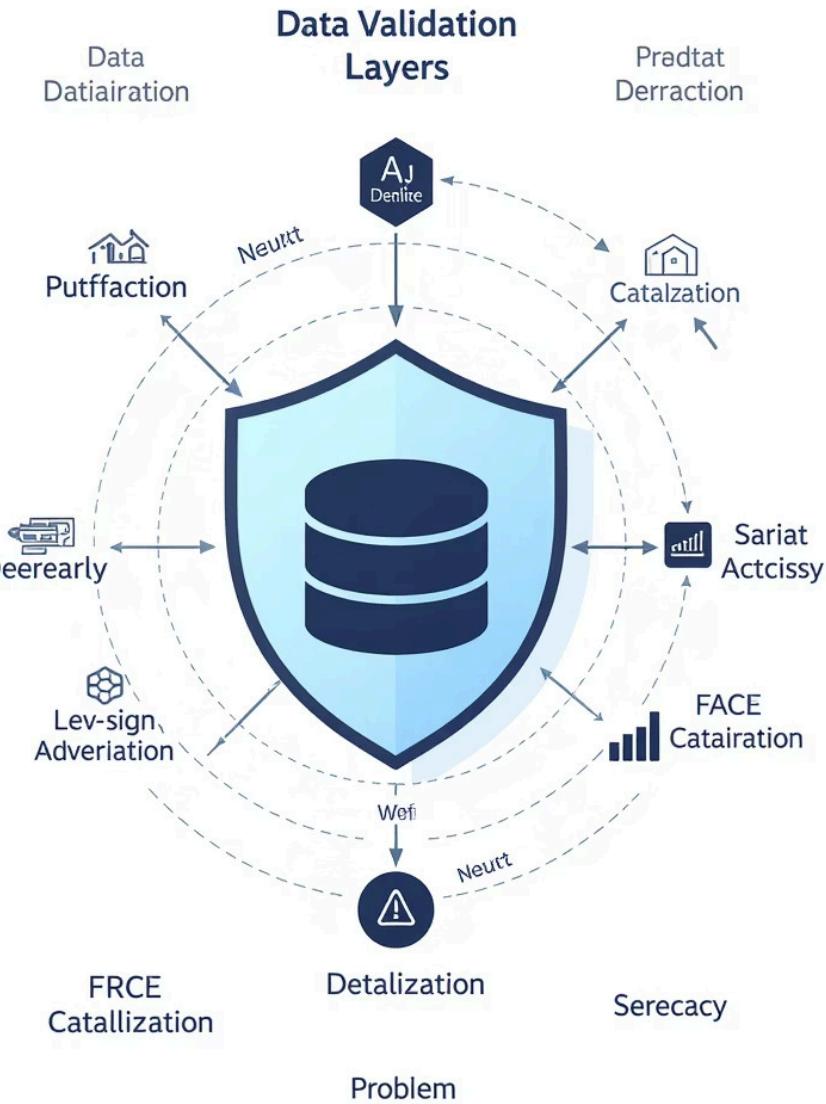
Validación temprana en formularios y API endpoints. Previene envío de datos incorrectos al servidor.

Capa de Negocio

Implementa reglas complejas y validaciones contextuales.
Garantiza coherencia entre entidades relacionadas.

Capa de Persistencia

Restricciones a nivel de base de datos como claves únicas e integridad referencial.



Validaciones: JSR-380

Bean Validation

JSR-380 es la especificación de Bean Validation 2.0 en Java

Anotaciones

Define anotaciones para validar propiedades de los **beans** o **DTO** (objetos de datos)

Errores

Mecanismo para reportar violaciones de validación

Validaciones: anotaciones comunes

- * **@NotNull / @NotBlank / @NotEmpty**

Que un valor esté presente

-  **@Size(min=, max=)**

Longitud de cadenas o tamaño de colecciones dentro de rangos

-  **@Min / @Max / @DecimalMin / @DecimalMax**

Valores numéricos dentro de límites

-  **@Email**

Formato de correo válido

-  **@Past / @Future**

Fechas en el pasado o futuro

-  **@Pattern(regexp="")**

Validación mediante expresiones regulares

-  **@URL**

Formato de URL válido

-  **@Positive / @Negative**

Valores numéricos positivos o negativos

Estas anotaciones llevan un mensaje de error por defecto personalizable.

Validaciones: Integración con Spring Boot

1

Integración en Spring Boot

Incorpora Hibernate Validator automáticamente.

Spring valida los datos de entrada al aplicar anotaciones en DTOs o entidades.

Lanza MethodArgumentNotValidException cuando detecta errores de validación.

2

Validación en Controladores

Usa @Valid en parámetros del controlador:

Rechaza automáticamente peticiones inválidas con estado 400 Bad Request.

3

Validación en Servicios

Usa @Validated a nivel de clase para habilitar validación en servicios y @Valid en los parámetros de los métodos públicos.

Permite validaciones complejas mediante grupos de validación. Ideal para reglas de negocio que involucran múltiples campos o entidades.

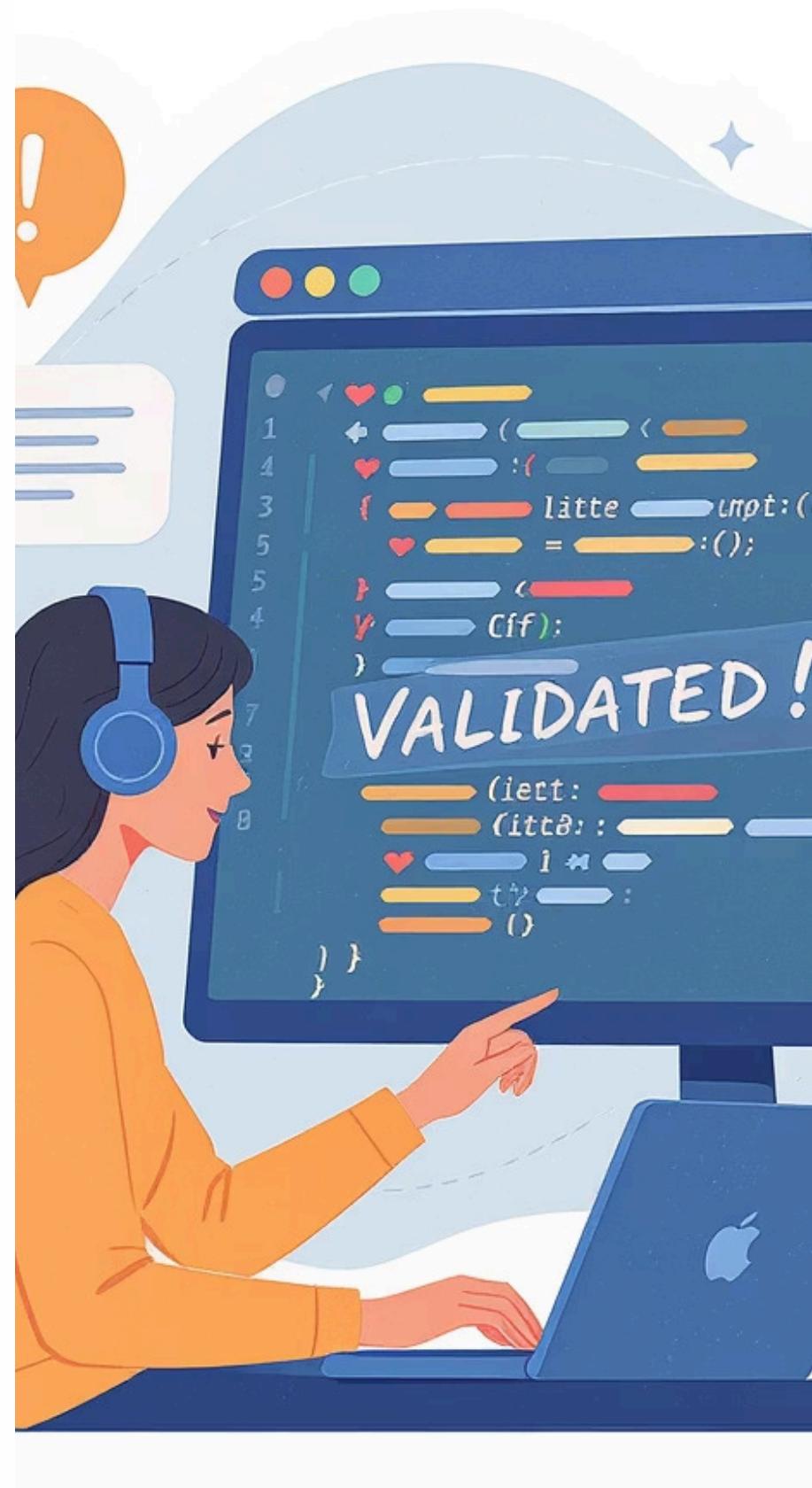
4

Validación en Repositorios

Spring Data JPA aplica validaciones de entidades en operaciones de persistencia.

Verifica anotaciones de validación antes de ejecutar operaciones en el repositorio.

Asegura la integridad de datos.



Validaciones: Uso de ProblemDetail

Manejo estandarizado de errores de validación

Spring Framework 6 introduce **ProblemDetail**, una implementación de la especificación RFC 7807 (Problem Details for HTTP APIs) que revoluciona el manejo de errores en APIs REST.

- Proporciona una estructura estandarizada para representar errores con claridad
- Mejora la comunicación entre cliente y servidor durante situaciones de error
- Define campos informativos como *type*, *title*, *status*, *detail* y *instance*

Cuando ocurren errores de validación, Spring lanza *MethodArgumentNotValidException*, que puede capturarse mediante un *@ExceptionHandler* para personalizar las respuestas de error.

Validaciones: ejemplo

```
// DTO con validaciones de Bean Validation
public record ClienteDTO(
    Long id,
    @NotBlank(message = "El NIF del cliente no puede estar vacío")
    String nif,
    @NotBlank(message = "El nombre no puede estar vacío")
    String nombre,
    @Email(message = "El email debe ser válido")
    String email
) {
}
```

```
// En el controlador Spring Boot
@PostMapping
public ResponseEntity<ClienteDTO> post(@RequestBody @Valid ClienteDTO dto) {
    Cliente cliente = clienteService.registrarCliente(mapper.toEntity(dto));
    ClienteDTO result = mapper.toDTO(cliente);
    return ResponseEntity.created(URI.create("/api/v1/clientes/" + result.id())).body(result);
}
```

Pruebas



Verificación

Comprobamos que el software cumple con sus requisitos funcionales y no funcionales.



Detección temprana

Identificamos errores en etapas iniciales del desarrollo.



Confianza

Aseguramos la estabilidad del sistema ante cambios y nuevas funcionalidades.



Automatización

Implementamos procesos repetibles que facilitan la integración continua.



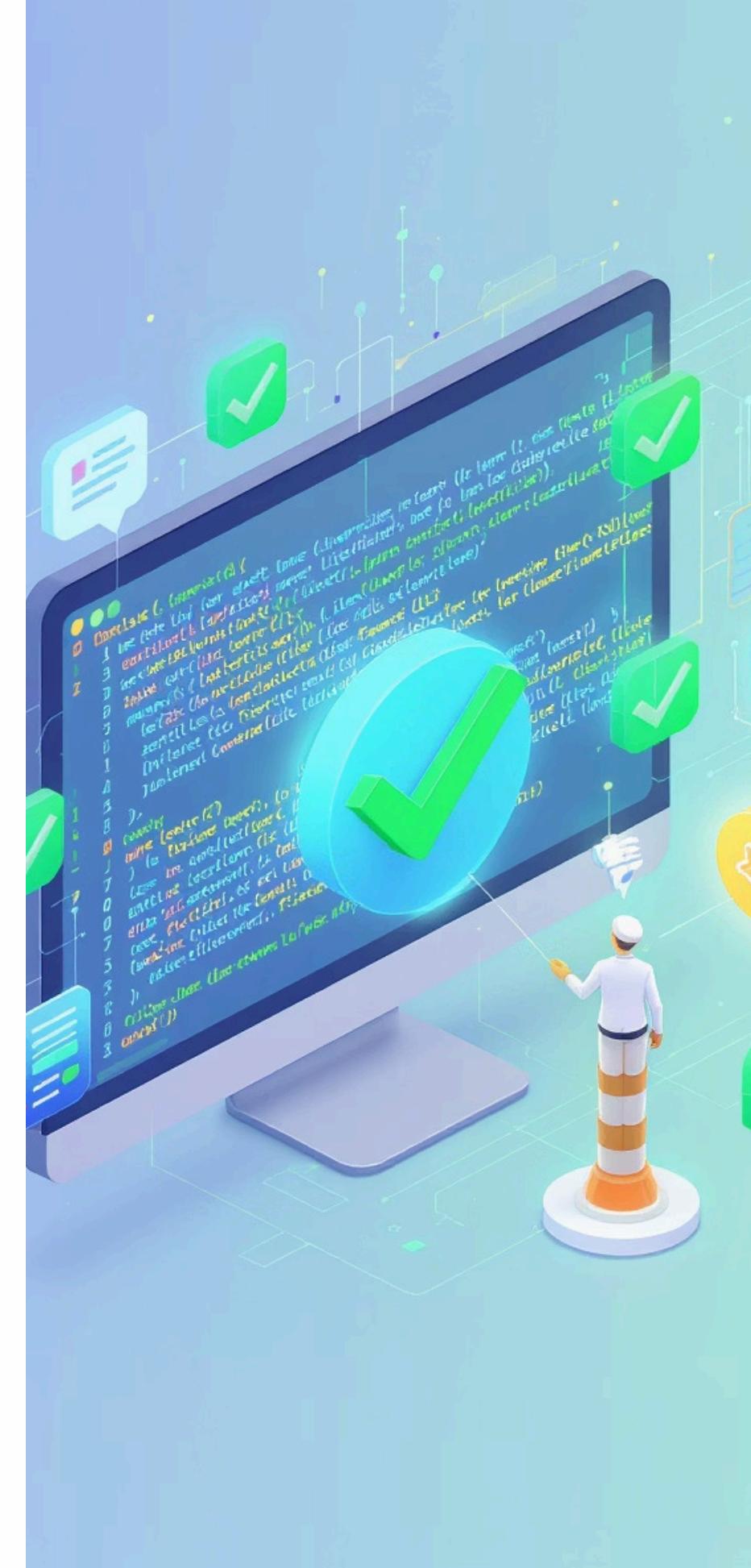
Esfuerzo adicional

Las pruebas requieren tiempo de programación que debe ser considerado en la planificación del proyecto.



Falsa seguridad

Pueden generar una sensación de seguridad engañosa si no cubren todos los escenarios posibles.



Pruebas: framework JUnit



JUnit 5 (Jupiter)

Versión moderna del popular framework de pruebas unitarias en Java



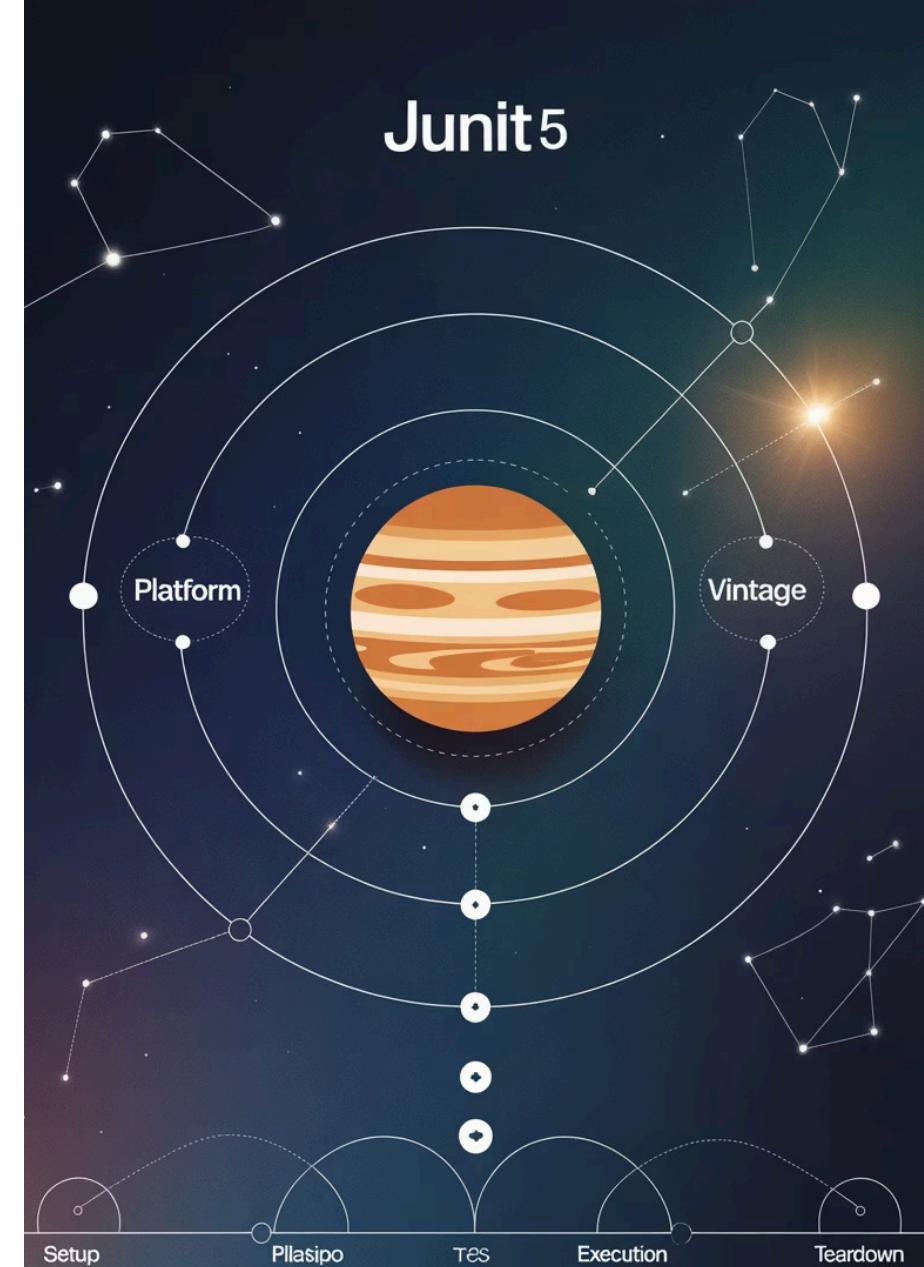
Arquitectura modular

Separado en módulos: Jupiter API, Platform, Vintage



Integración con build tools

Maven (plugin Surefire) o Gradle ejecutan tests automáticamente durante el build



Pruebas: JUnit

Estructura básica

- @Test - Marca métodos de prueba
- @BeforeEach / @AfterEach - Antes/después de cada test
- @BeforeAll / @AfterAll - Antes/después de todos los tests
- @DisplayName - Nombre descriptivo para el test

Aserciones

- assertEquals - Verifica igualdad
- assertTrue / assertFalse - Condiciones booleanas
- assertNotNull / assertNull - Verificar nulidad
- assertThrows - Verificar excepciones

Pruebas: ejemplo

Junit 5 Test Example Code

```
CalculatorTest.java
public class CalculatorTest {
    @Test
    void sumarDosNumeros_retornaResultadoCorrecto() {
        Calculator calc = new Calculator();
        int resultado = calc.sumar(2, 3);
        assertEquals(5, resultado);
    }

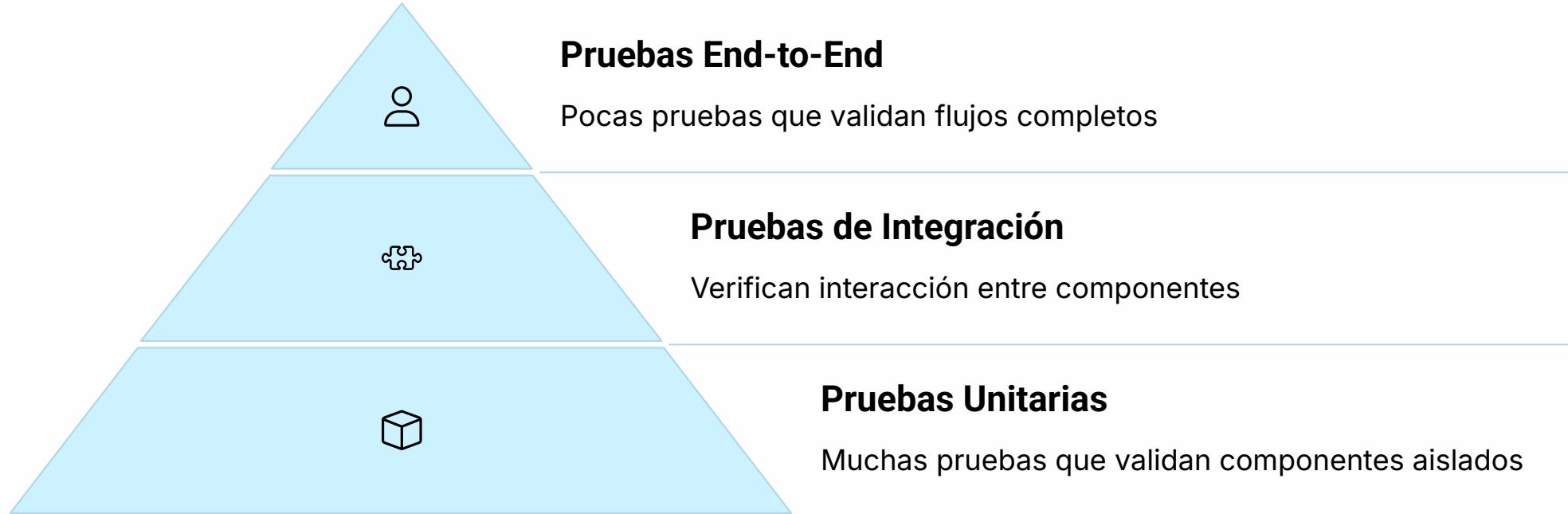
    @Test
    void dividir_divisionPorCero_lanzaExpcion() {
        Calculator calc = new Calculator();
        assertThrows(ArithmeticException.class,
            () -> calc.dividir(10, 0));
    }
}
```

```
class CalculadoraTest {

    @Test
    void sumarDosNumeros_retornaResultadoCorrecto() {
        Calculadora calc = new Calculadora();
        int resultado = calc.sumar(2, 3);
        assertEquals(5, resultado);
    }

    @Test
    void dividir_divisionPorCero_lanzaExpcion() {
        Calculadora calc = new Calculadora();
        assertThrows(ArithmeticException.class,
            () -> calc.dividir(10, 0));
    }
}
```

Pruebas: niveles de prueba



Una estrategia de pruebas equilibrada sigue la pirámide de pruebas: muchas pruebas unitarias rápidas en la base, pruebas de integración en el medio y algunas pruebas end-to-end más lentas en la cima, asegurando calidad a diferentes niveles.

Pruebas: principios FIRST

F - Fast (Rápido)

Las pruebas deben ejecutarse rápidamente

- Minimizar tiempos de ejecución
- Permitir retroalimentación frecuente

I - Independent (Independiente)

Cada prueba debe ser independiente del resto

- No depender del estado de otras pruebas
- Evitar dependencias entre tests

R - Repeatable (Repetible)

Las pruebas deben ser repetibles

- Mismo resultado en cada ejecución
- Evitar dependencias externas no controladas

S - Self-validating (Auto-validante)

Las pruebas deben validarse a sí mismas

- Usar aserciones para verificar resultados
- No requerir verificación manual

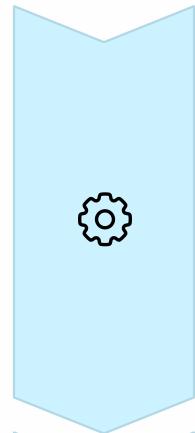
T - Timely (Oportuno)

Escribir pruebas en el momento adecuado

- Idealmente antes o junto con el código
- TDD: Test-Driven Development

Pruebas: estructura recomendada

Es útil seguir la estructura GIVEN-WHEN-THEN:



Preparación (Given)

Configurar el escenario de prueba



- Crear objetos necesarios
- Configurar mocks
- Preparar datos de entrada



Ejecución (When)

Invocar el método o funcionalidad a probar



- Llamar al método bajo prueba
- Capturar el resultado o excepción



Verificación (Then)

Comprobar que el resultado es el esperado



- Usar aserciones (assertEquals, assertTrue, etc.)
- Verificar interacciones con mocks



Pruebas: unitarias (por capas)

Un enfoque sistemático para garantizar la calidad del software en arquitectura por capas

Pruebas de controladores

Verifican la capa de presentación con servicios simulados

- Validan el correcto mapeo de peticiones HTTP
- Comprueban la gestión de errores y excepciones
- Aseguran respuestas con formatos y códigos correctos

Pruebas de servicios

Comprueban la lógica de negocio con repositorios simulados

- Verifican las reglas y algoritmos del dominio
- Evalúan el manejo de casos límite y excepcionales
- Aseguran la correcta aplicación de transacciones

Pruebas de repositorios

Validan el acceso a datos con bases de datos en memoria

- Comprueban las consultas y operaciones CRUD
- Verifican el mapeo entre objetos y estructuras de datos
- Evalúan el rendimiento de las operaciones de datos

Pruebas: mocks

Problema en pruebas unitarias

Las clases frecuentemente presentan dependencias complejas: servicios, repositorios, clientes HTTP, sistemas de bases de datos y APIs externas.

En el contexto de pruebas unitarias, cuando evaluamos una clase con dependencias, no deseamos invocar las implementaciones reales de estas dependencias. Nuestro objetivo es aislar y probar únicamente el comportamiento de la clase bajo prueba, no de sus componentes asociados.

Objetos simulados al rescate!

Implementaremos objetos simulados o "dobles de prueba" (mocks) para:

- Aislar completamente la unidad bajo prueba, garantizando pruebas verdaderamente unitarias
- Controlar con precisión el comportamiento de las dependencias en diferentes escenarios
- Simular condiciones excepcionales, situaciones de error y casos límite difíciles de reproducir
- Evita incurrir en costes innecesarios
- Ejecutar pruebas de manera más rápida, predecible y consistente



Pruebas: mocks con Mockito

Mockito es un framework de simulación que permite crear sustitutos de objetos para pruebas unitarias efectivas. Facilita la configuración de comportamientos predefinidos, aísla completamente el código bajo prueba, y ofrece mecanismos robustos para verificar interacciones, excepciones y argumentos capturados durante la ejecución.

Creación de mocks

Instanciar objetos simulados mediante anotaciones o el método Mockito.mock() para reemplazar dependencias reales

Programación de comportamiento

Configurar respuestas específicas con when().thenReturn() o doReturn().when() para simular distintos escenarios de ejecución

Ejecución del código a probar

Invocar el método de la clase bajo prueba que interactúa con las dependencias simuladas para evaluar su comportamiento aislado

Verificación de interacciones

Utilizar verify() para confirmar que los métodos de los mocks fueron invocados con los parámetros esperados y en el orden correcto

Pruebas: inyección de Mocks

Inyección automática con anotaciones

```
@ExtendWith(MockitoExtension.class)
class ClienteServiceTest {

    @Mock
    private ClienteRepository clienteRepo;

    @Mock
    private NotificationService notificationService;

    @InjectMocks
    private ClienteService clienteService;

    ....
}
```

Pruebas: ejemplo test unitario

```
@Test
void shouldRegistrarCliente() {
    when(clienteRepo.save(cliente)).thenReturn(cliente);

    Cliente result = clienteService.registrarCliente(cliente);

    assertEquals(cliente, result);

    verify(clienteRepo).save(cliente);
}

@Test
void shouldNotObtenerClienteNoExistente() {
    when(clienteRepo.findById(99L)).thenReturn(Optional.empty());

    assertThrows(ResourceNotFoundException.class, () -> clienteService.obtenerCliente(99L));

    verify(clienteRepo).findById(99L);
}
```

Pruebas: Integración

INTEGRATION TESTING

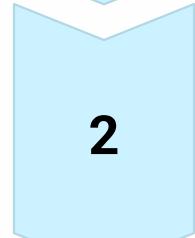
Verifican la correcta interacción entre componentes y subsistemas de nuestra aplicación.

Podemos seguir un enfoque top-down o bottom-up o mezclado



1 Cliente - API

El *cliente* accediendo al API a través del protocolo HTTP. Sirven para comprobar mapeos de URL, conversión de datos JSON/Java, manejo de errores estándar, seguridad, etc.



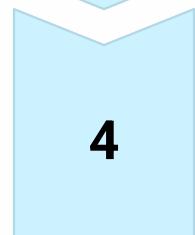
2 Controladores - Servicios

Los controladores invocando a las clases de negocio (servicios)



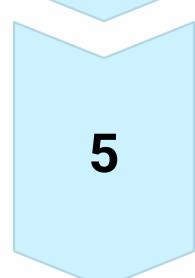
3 Servicios - Servicios Externos

Las clases de servicios accediendo a *servicios externos*



4 Servicios - Repositorios

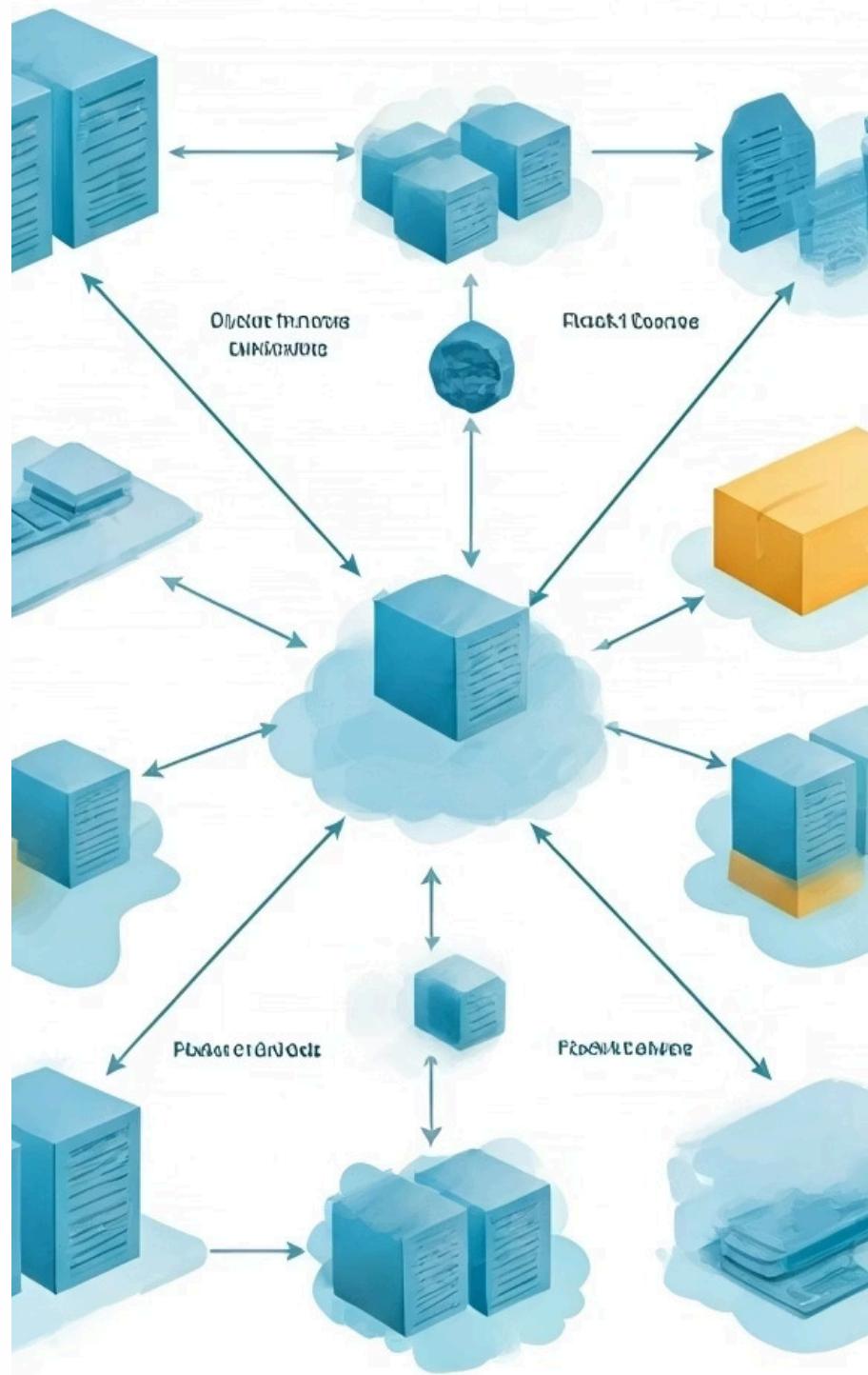
Las clases de servicio utilizan las clases de acceso a datos (repositorios)



5 Repositorios - Base de Datos

Las clases de acceso a datos (repositorios) se comunican con la *base de datos*

ARCHITECTURE



Pruebas: ejemplo test integración

```

@SpringBootTest
@AutoConfigureMockMvc
@AutoConfigureTestDatabase // H2 embebida
public class CuentaControllerIntegrationTest {

    @Autowired
    private MockMvc mockMvc;

    @Autowired
    private ClienteRepository clienteRepository;

    @Test
    public void shouldCrearCuentaParaClienteExistente() throws Exception {

        // Given: hay un cliente existente
        Cliente cliente = clienteRepository.save(new Cliente("123", "Juan", "ivan@uca.es"));

        // When: se hace la petición para crear una cuenta
        String nuevaCuentaJson = """
            {
                "clientelId": %s
            }
            """.formatted(cliente.getId());

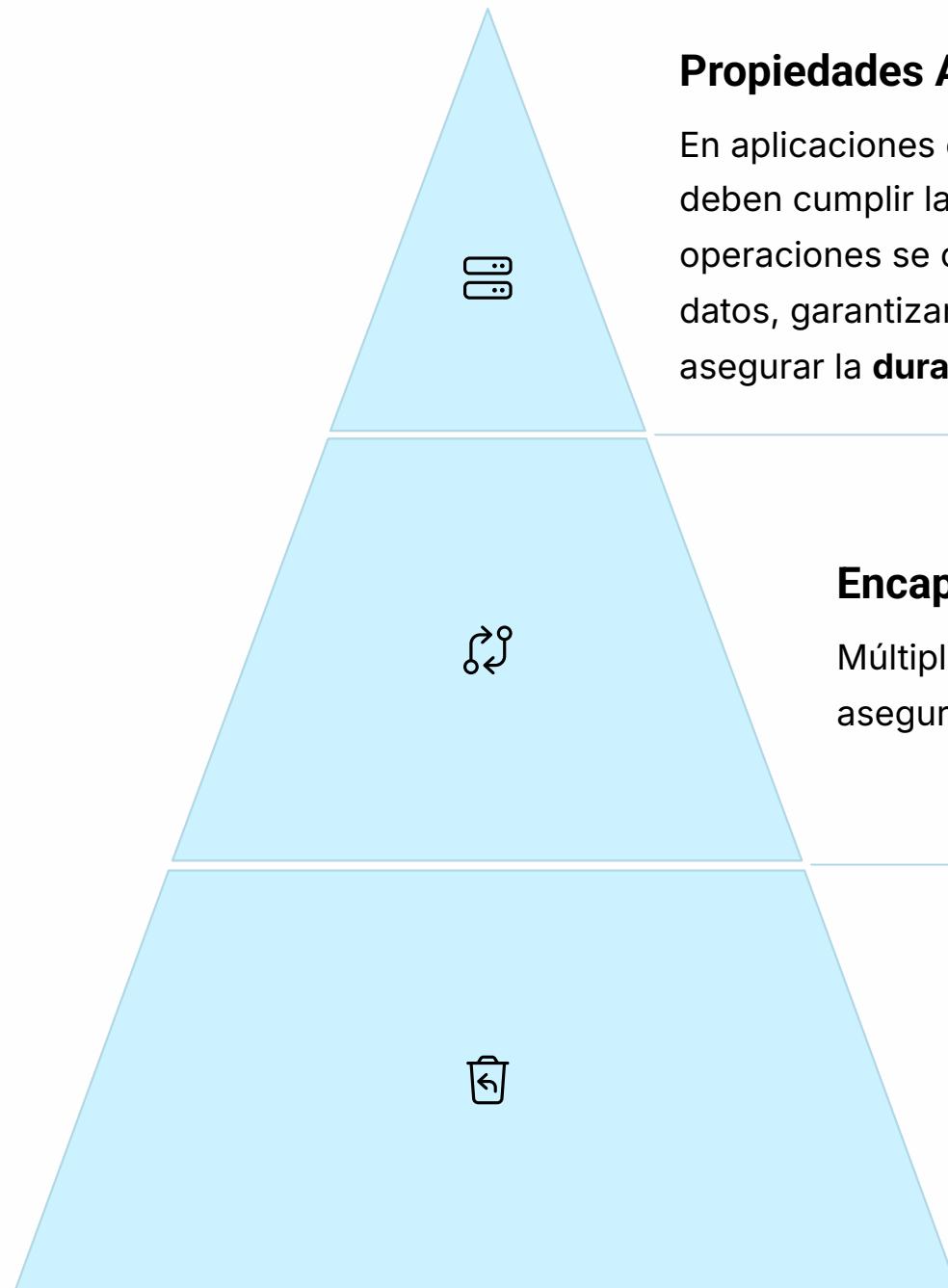
        mockMvc.perform(post("/api/v1/cuentas")
                .with(httpBasic("usuario", "usuario")) // Basic Auth credentials
                .contentType(MediaType.APPLICATION_JSON)
                .content(nuevaCuentaJson))
                // Then: se espera que la cuenta se cree correctamente
                .andExpect(status().isCreated())
                .andExpect(header().exists("Location"))
                .andExpect(header().string("Location", matchesPattern("/api/v1/cuentas/\d+")));

    }

}

```

Transacciones



Propiedades ACID

En aplicaciones empresariales, las operaciones en la base de datos deben cumplir las propiedades **ACID**: ser **atómicas** (todas las operaciones se completan o ninguna), mantener la **consistencia** de los datos, garantizar el **aislamiento** entre transacciones concurrentes y asegurar la **durabilidad** de los cambios.

Encapsulación

Múltiples operaciones como una unidad lógica de trabajo, asegurando la integridad de los datos incluso ante fallos.

Rollback Automático

Reversión en caso de excepciones

Transacciones: uso en Spring

Soporte en Spring

Spring permite manejo declarativo de transacciones mediante `@Transactional`.

Los métodos de los repositorios Spring Data JPA son transaccionales por defecto

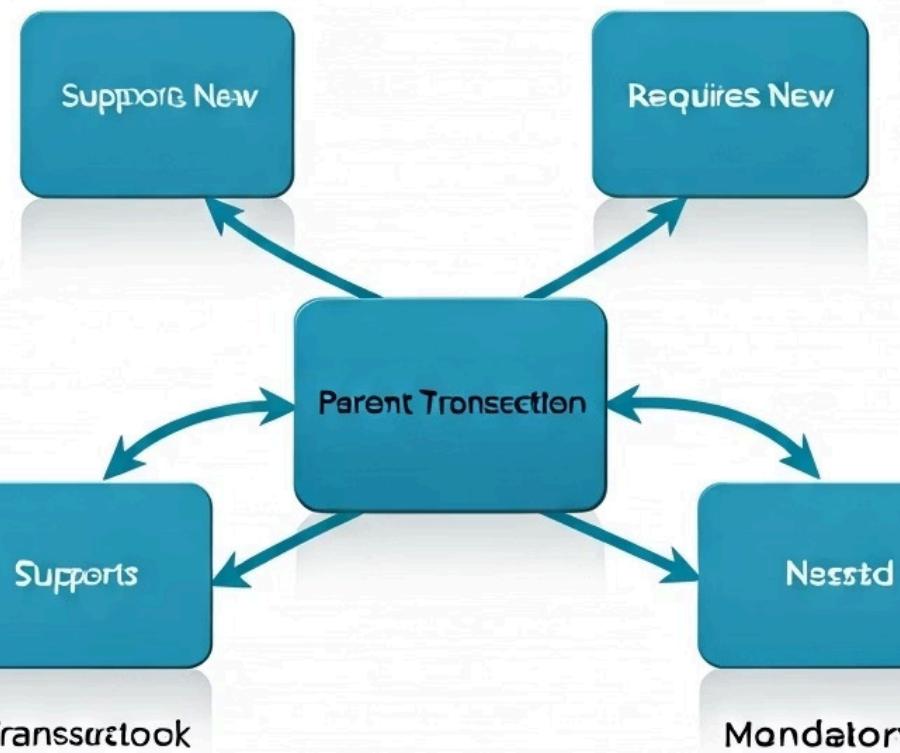
Al anotar un método de servicio con `@Transactional`, Spring abre automáticamente una transacción al inicio del método y la cierra al final (haciendo **commit** si todo va bien, o **rollback** si ocurre una excepción no controlada).

Rollback Automático

Por defecto, Spring solo marca la transacción para rollback si la excepción lanzada es **unchecked** (`RuntimeException`).

Las excepciones checked (`IOException`, `SQLException`, etc.) no provocan rollback automático a menos que se indique (`rollbackFor`). Esto se puede personalizar en la anotación.

Transacciones: propagación



La propagación en `@Transactional` define cómo se comporta una transacción cuando un método anotado es invocado desde otro método que ya tiene una transacción activa.



REQUIRED (por defecto)

Utiliza la transacción existente si hay una activa, o crea una nueva transacción si no existe ninguna.



REQUIRES_NEW

Siempre crea una nueva transacción independiente, suspendiendo temporalmente la transacción actual si existe. Útil cuando necesitamos operaciones aisladas (v.g. audit_log).



Otros modos

NESTED (procesos por lotes que toleran fallos parciales), SUPPORTS (participa en transacción si existe, sin transacción en caso contrario), MANDATORY (requiere transacción existente), NEVER (error si existe transacción), NOT_SUPPORTED (suspende transacción existente).



The screenshot shows a Java IDE interface with three tabs open. The left tab contains a list of icons and their corresponding names: Cuenta Banco, Cuenta Baja, Cuenta Corriente, Cuenta Negocios, Cuenta Operativa, Cuenta Personal, Cuenta Plataforma, Cuenta Salvo, Cuenta Simple, Cuenta Transferencia, Cuenta Virtual, and Movimiento. The middle tab displays a Java code snippet for a 'transferir' method. The right tab shows a detailed log or audit trail for a transaction, listing various steps and their timestamps.

```
public void transferir(Long origenId, Long destinoid, BigDecimal cantidad, String descripcion) {
    Cuenta origen = obtenerCuenta(origenId);
    origen.validarOperacionRetiro(cantidad);
    origen.decrementarSaldo(cantidad);
    cuentaRepo.save(origen);

    Cuenta destino = obtenerCuenta(destinoid);
    destino.incrementarSaldo(cantidad);
    cuentaRepo.save(destino);

    Movimiento movimiento = Movimiento.transferencia(origen, destino, cantidad, descripcion);
    movimientoRepo.save(movimiento);
}
```

Log details from the right tab:

- 00:00:00.000 - 00:00:00.000: Inicio de transacción
- 00:00:00.000 - 00:00:00.000: Cuenta origen: Cuenta Banco
- 00:00:00.000 - 00:00:00.000: Cuenta destino: Cuenta Banco
- 00:00:00.000 - 00:00:00.000: Cantidad: 100.00
- 00:00:00.000 - 00:00:00.000: Descripción: Transferencia entre cuentas
- 00:00:00.000 - 00:00:00.000: Validación de saldo en cuenta origen
- 00:00:00.000 - 00:00:00.000: Decremento de saldo en cuenta origen
- 00:00:00.000 - 00:00:00.000: Guardado de movimiento en repositorio
- 00:00:00.000 - 00:00:00.000: Cuenta destino: Cuenta Banco
- 00:00:00.000 - 00:00:00.000: Cuenta origen: Cuenta Banco
- 00:00:00.000 - 00:00:00.000: Cantidad: 100.00
- 00:00:00.000 - 00:00:00.000: Descripción: Transferencia entre cuentas
- 00:00:00.000 - 00:00:00.000: Incremento de saldo en cuenta destino
- 00:00:00.000 - 00:00:00.000: Guardado de movimiento en repositorio
- 00:00:00.000 - 00:00:00.000: Finalización de transacción

Transacciones: ejemplo

@Transactional

```
public void transferir(Long origenId, Long destinoid, BigDecimal cantidad, String descripcion) {
    Cuenta origen = obtenerCuenta(origenId);
    origen.validarOperacionRetiro(cantidad);
    origen.decrementarSaldo(cantidad);
    cuentaRepo.save(origen);

    Cuenta destino = obtenerCuenta(destinoid);
    destino.incrementarSaldo(cantidad);
    cuentaRepo.save(destino);

    Movimiento movimiento = Movimiento.transferencia(origen, destino, cantidad, descripcion);
    movimientoRepo.save(movimiento);
}
```

Caché

Mecanismo esencial para mejorar el rendimiento de aplicaciones Spring Boot mediante el almacenamiento temporal de datos frecuentemente utilizados.



Rendimiento Mejorado

Reduce drásticamente los tiempos de respuesta al evitar operaciones costosas (consultas a BD, cálculos complejos) o repetitivas.



Menor Carga

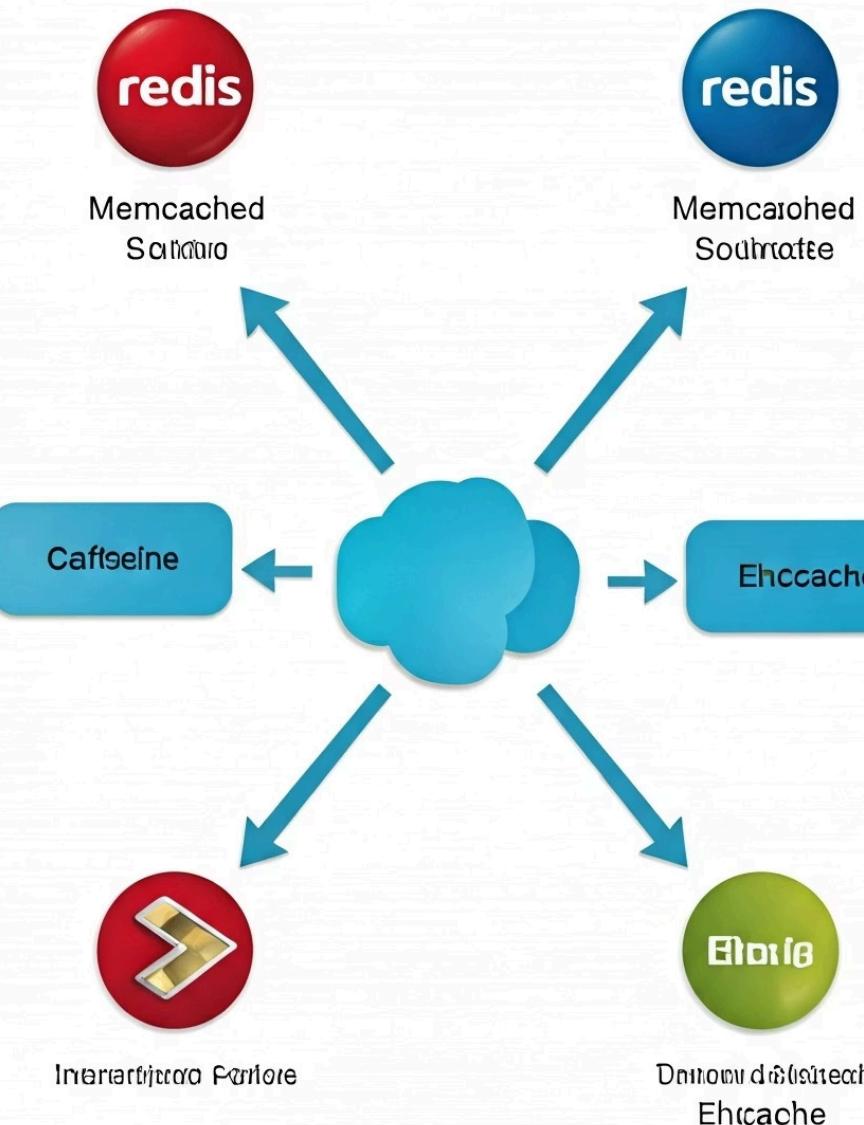
Disminuye la presión sobre bases de datos y servicios externos.



Configuración Sencilla

Implementación mediante anotaciones sin necesidad de código complejo.





Caché: proveedores



ConcurrentHashMap

Proveedor por defecto, en memoria de la JVM.



Ehcache

Caché en memoria con opciones avanzadas.



Caffeine

Caché de alto rendimiento para Java.



Redis

Almacén de datos en memoria distribuido.

```
cacheable - cacheable=true (defecto); cacheable=false (no cache);
cacheable(key).key=expresión SpEL;
cacheable(true);
cacheable(false);
cacheable(expresión SpEL);
```

Caché: anotaciones en Spring



@Cacheable(value="nombreCache", key="#parametro")

Almacena el resultado del método en caché. La clave se define con SpEL: "#parametro", "#root.methodName" o "#p0". En futuras invocaciones con la misma clave, Spring retorna el resultado desde caché sin ejecutar el método.



@CacheEvict(value="nombreCache", key="#id", allEntries=true)

Elimina entradas específicas (mediante key) o toda la caché (allEntries=true). Útil después de operaciones de modificación para evitar que lecturas posteriores obtengan datos obsoletos.



@CachePut(value="nombreCache", key="#result.id")

Ejecuta el método y actualiza la caché con su resultado. A diferencia de @Cacheable, siempre ejecuta el método. Puede referenciar parámetros o el resultado (#result). Ideal para mantener la caché sincronizada tras actualizaciones.

Caché: ejemplo

```
@SpringBootApplication
@EnableCaching // Habilita la funcionalidad de caché de Spring
public class Aplicacion {
    /* ... */
}

@Cacheable(value = "bancosCache", key = "#id")
public Banco obtenerBanco(Long id) {
    System.out.println("Leyendo banco de la base de datos con ID: " + id);
    return repo.findById(id).orElseThrow(() -> new ResourceNotFoundException("Banco", id));
}
```

En la primera invocación de `obtenerBanco()`, se mostrará el mensaje y se consultará la BD. En llamadas posteriores, mientras la caché `bancosCache` tenga datos válidos, no se volverá a ejecutar el método ni consultar la BD.

Cliente HTTP: servicios web

Los servicios web permiten la comunicación entre aplicaciones a través de la red utilizando protocolos estándar como REST y SOAP, facilitando la integración entre sistemas heterogéneos.

Al consumir servicios web, debemos gestionar aspectos críticos como el manejo de errores, tiempos de espera (timeout), políticas de reintentos, serialización/deserialización de datos y mecanismos de autenticación y autorización.

Spring Framework proporciona herramientas robustas que permiten a nuestras aplicaciones consumir servicios de terceras partes de manera eficiente, segura y con código limpio y mantenible.

Cliente HTTP: RestTemplate



Simplificación de APIs REST

Clase utilitaria para el consumo de APIs REST.



Cliente HTTP Síncrono

Cliente HTTP síncrono (bloqueante).



API Simple

API simple y directa.



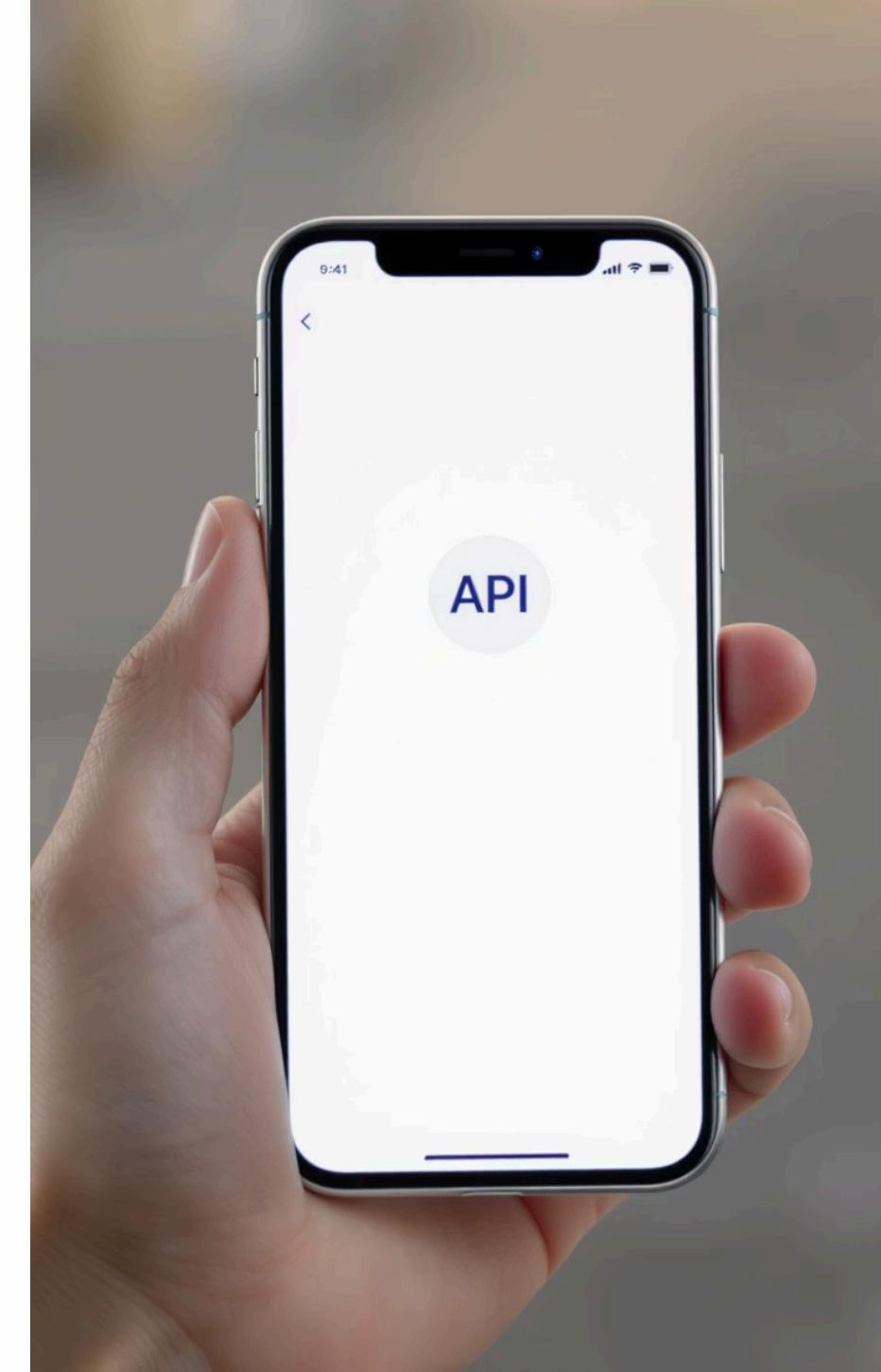
Aplicaciones Tradicionales

Ideal para aplicaciones tradicionales, no para entornos de alta concurrencia.



Análisis de JSON

Analiza respuestas JSON y devuelve objetos Java.



Cliente HTTP: configuraciones adicionales

Excepciones por Defecto

Por defecto, RestTemplate lanza excepciones específicas (`HttpClientErrorException`, `HttpServerErrorException`) cuando la respuesta tiene códigos de estado 4xx o 5xx. Estas excepciones pueden capturarse para implementar un manejo de errores personalizado.

Personalización

Es posible configurar un `ResponseErrorHandler` personalizado para modificar la lógica de gestión de errores según las necesidades específicas de la aplicación, permitiendo mayor control sobre las respuestas HTTP.

Timeout

Se puede establecer un *Timeout* para limitar el tiempo máximo de espera en las peticiones, evitando bloqueos indefinidos cuando un servicio no responde.

Interceptors

Se pueden incorporar **interceptors** para intervenir en el ciclo de vida de las peticiones, permitiendo registrar trazas, añadir cabeceras o modificar el contenido de todas las peticiones de forma centralizada.



Cliente HTTP: uso de RestTemplate

+ Creación

Se puede crear una instancia con RestTemplate rest = new RestTemplate();

getForObject

`getForObject(url, Clase.class)` realiza un HTTP GET y convierte la respuesta JSON en una instancia de Clase

postForEntity

`postForEntity(url, objetoRequest, ClaseResp.class)` realiza un HTTP POST enviando un objeto (como JSON) en el cuerpo, y devuelve una `ResponseEntity` con la respuesta convertida a `ClaseResp`

Otros Métodos

Métodos similares para PUT (put), DELETE (delete), etc.

Cliente HTTP: ejemplo

Respuesta del Servicio web <https://apisandbox.openbankproject.com/obp/v4.0.0/banks>

```
{
  "banks": [
    {
      "id": "rbs",
      "short_name": "The Royal Bank of Scotland",
      "full_name": "The Royal Bank of Scotland",
      "logo": "http://www.red-bank-shoreditch.com/logo.gif",
      "website": "http://www.red-bank-shoreditch.com",
      "bank_routings": [
        {
          "scheme": "OBP",
          "address": "rbs"
        }
      ],
      "attributes": []
    },
    {
      ...
    }
  ]
}
```

Mapeo Java-JSON

```
public class BankAPIResponse {
  private List<Bank> banks;

  public List<Bank> getBanks() {
    return banks;
  }

  public void setBanks(List<Bank> banks) {
    this.banks = banks;
  }
}
```

Uso de RestTemplate

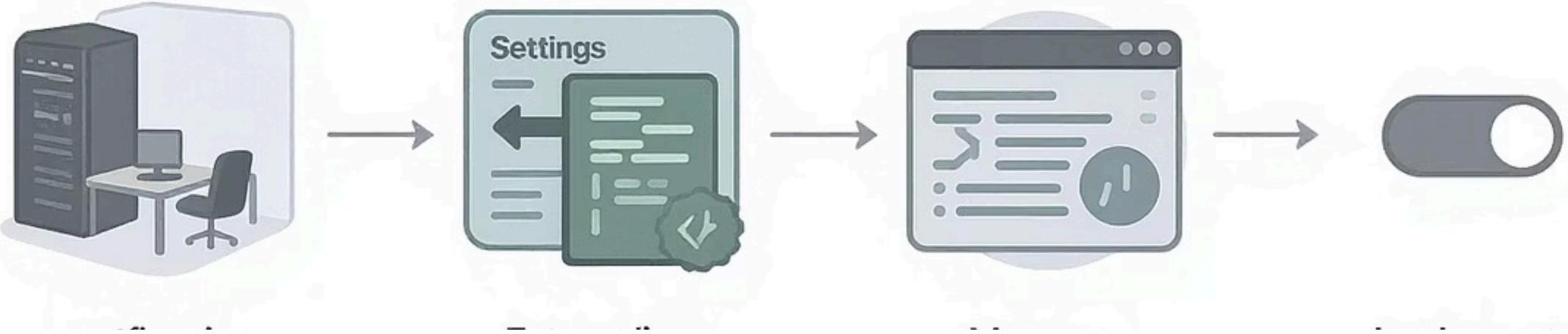
```
public void fetchAll() {
  String url = "https://apisandbox.openbankproject.com/obp/v4.0.0/banks";
  System.out.println("Descargando datos de bancos...");
  RestTemplate restTemplate = new RestTemplate();
  BankAPIResponse response = restTemplate.getForObject(url, BankAPIResponse.class);
  syncBancos(response.getBanks());
  System.out.println("Datos actualizados.");
}
```

Perfiles

Las aplicaciones suelen tener requisitos distintos en función de su ámbito de ejecución. Por ejemplo, necesitamos configuraciones específicas para entornos de desarrollo, pruebas o producción.

Los perfiles permiten adaptar la configuración de la aplicación según diferentes contextos como:

- Entornos de ejecución (desarrollo, QA, producción)
- Múltiples tenants en aplicaciones multi-inquilino
- Diferentes plataformas hardware/software
- Configuraciones específicas por región o cliente



Perfiles: en Spring Boot

Perfiles

Los perfiles en Spring Boot permiten configurar diferentes entornos, adaptando la aplicación **en tiempo de ejecución** según el contexto operativo sin necesidad de recompilar o modificar el código fuente.

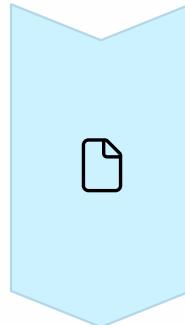
Gestión de Configuración Base

Define valores predeterminados en **application.properties** y sobrescribe únicamente los parámetros específicos que requieran personalización en cada perfil, manteniendo una estructura organizada y reduciendo la duplicación.

Externalización de Configuraciones Variables

Separa elementos sensibles como credenciales de bases de datos, tokens API y endpoints de servicios en archivos de perfil independientes para mejorar la seguridad y facilitar auditorías. Adicionalmente, permite ajustar configuraciones de logging, puertos del servidor, políticas CORS y otros parámetros según el entorno.

Perfiles: en Spring Boot (II)



application-{profile}.properties

Spring Boot soporta perfiles de ejecución para cargar configuraciones diferentes según el entorno activo. Por convención, se crean archivos de propiedades o YAML nombrados como application-dev.properties, application-prod.properties, etc., que contienen overrides específicos para ese perfil.



@Profile en Beans

Además de propiedades, Spring permite anotar beans con `@Profile("nombrePerfil")` para que solo se carguen en ciertos perfiles. Por ejemplo, usar un servicio de pago online real para prod o uno simulado para dev y test.

Perfiles: en Maven

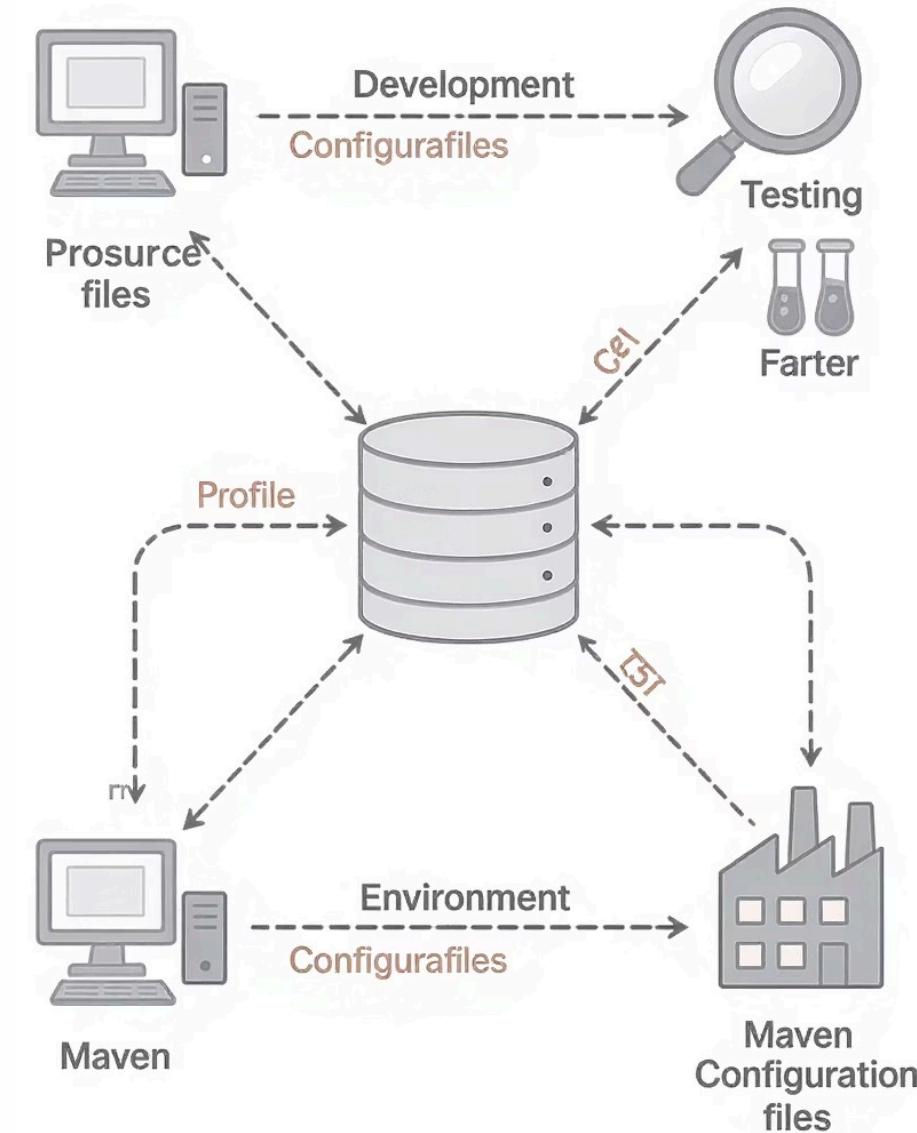
Configuraciones Múltiples

Los **perfiles** en Maven también permiten facilitar la adaptación de la aplicación a diferentes entornos pero **en tiempo de construcción**.

Se pueden definir variaciones de configuración en el pom.xml para diferentes entornos. Cada perfil puede cambiar dependencias, plugins de Maven, propiedades, etc.

¿Qué pueden cambiar?

Por ejemplo, un perfil dev podría incluir la dependencia de una base de datos MySQL y otro prod para Oracle. Usar diferentes versiones de la JDK según el entorno. Generar diferentes componentes nativos para sistemas operativos distintos (GraalVM).



Perfiles: activación contextual

Integración Maven + Spring

Configurar mecanismos para activar perfiles que aseguren despliegues consistentes y reproducibles en distintos entornos.

Los perfiles de Maven (build) y Spring (runtime) pueden (y deben) complementarse. Por ejemplo, el perfil Maven prod activa también el perfil Spring prod.

Activación de Perfiles en Maven

Aunque se puede hacer de forma **automática**, en función de condiciones definidas en POM.xml sobre propiedades del sistema o usuario, lo habitual es hacerlo de forma..

Manual: Uso del flag -P. Ejemplo: `mvn clean package -Pprod` activa el perfil prod.

Activación de Perfiles en Spring

Se usa la propiedad `spring.profiles.active` y es posible activar múltiples perfiles simultáneamente. Ej:
`spring.profiles.active=dev,debug`

Formas de activación:

- Variable de entorno del sistema operativo
- Argumento JVM: `-Dspring.profiles.active=dev`
- Parámetro de línea de comandos: `--spring.profiles.active=dev`
- Establecido mediante una **propiedad** definida en un perfil de Maven en el pom.xml

Carga de Configuración Externa

Es posible mantener la configuración específica por entorno separada del binario. Para leer configuración desde fuera del JAR, usar:

```
java -jar app.jar --spring.config.additional-location=file:./config/
```

Perfiles: ejemplo de Spring

application.properties

```
# Profile selected from a Maven property  
spring.profiles.active=@spring.profiles.active@  
  
# Enable SSL support. If not set, the server will run on HTTP.  
server.ssl.enabled=true  
  
# The format used for the keystore  
server.ssl.key-store-type=PKCS12
```

application-dev.properties

```
spring.security.oauth2.client.provider.demoapp.token-uri=https://prauthserver.es/OAUTH2/authserver.php  
  
spring.security.oauth2.client.provider.demoapp.issuer-uri=https://prauthserver.es
```

Inyección de propiedades en objetos

```
@Value("${ucabank.api.url}")  
private String url;
```

Perfiles: ejemplo de Maven

```
<profiles>
  <profile>
    <id>dev</id>
    <activation>
      <activeByDefault>true</activeByDefault>
    </activation>
    <properties>
      <spring.profiles.active>dev</spring.profiles.active>
    </properties>
    <dependencies>
      <dependency>
        <groupId>com.mysql</groupId>
        <artifactId>mysql-connector-j</artifactId>
      </dependency>
    </dependencies>
  </profile>

  <profile>
    <!-- Production mode is activated using -Pproduction-->
    <id>prod</id>
    <properties>
      <spring.profiles.active>prod</spring.profiles.active>
    </properties>
    <dependencies>
      <dependency>
        <groupId>com.oracle.database.jdbc</groupId>
        <artifactId>ojdbc11</artifactId>
        <version>23.5.0.24.07</version>
      </dependency>

      <!-- Exclude development dependencies from prod -->
      <dependency>
        <groupId>..</groupId>
        <artifactId>..</artifactId>
        <exclusions>
          <exclusion>
            ...
          </exclusion>
        </exclusions>
      </dependency>
    </dependencies>
    <build>
      <plugins>
        <plugin>
          <!-- Plug-ins para optimizar buids -->
          ...
        </plugin>
      </plugins>
    </build>
  </profile>
</profiles>
```

Perfiles: diferencias entre Maven y Spring Boot

Perfiles de Maven

- Definidos en pom.xml
- Afectan al proceso de build
- Cambian dependencias, plugins, propiedades
- Activación con -P o automática

Perfiles de Spring Boot

- Definidos en ficheros properties
- Afectan al comportamiento en runtime
- Cambian configuraciones de la aplicación
- Activación con spring.profiles.active



Observabilidad

La observabilidad permite comprender el estado interno de un sistema a través de sus salidas externas. Entender qué sucede internamente en la aplicación en tiempo real para diagnosticar problemas complejos en sistemas distribuidos.



Logs

Registros detallados de eventos ocurridos durante la ejecución de la aplicación.

Se recomienda usar formatos de log estructurados (JSON) para facilitar el análisis y búsqueda.

¡Cuidado con registrar información sensible en los logs!



Métricas

Mediciones numéricas del comportamiento del sistema a lo largo del tiempo.

Identificar y monitorizar las métricas clave para el negocio, no solo las técnicas.



Trazas

Seguimiento del flujo de una solicitud a través de diferentes componentes.

En ocasiones será necesario implementar IDs de correlación para seguir peticiones a través de múltiples servicios.



Alertas Accionables

Configurar alertas solo para situaciones que requieran intervención humana inmediata.

Observabilidad: logs



Configuración

Spring Boot integra Logback como implementación predeterminada de SLF4J, permitiendo una gestión eficiente de logs.

Fácilmente configurable mediante application.properties o con mayor flexibilidad a través de logback-spring.xml para escenarios complejos.



Niveles

Jerarquía de niveles TRACE, DEBUG, INFO, WARN y ERROR que permite clasificar mensajes según su importancia y propósito.

Configurable de manera granular por paquete o clase específica, optimizando el volumen de información según el entorno.



Rotación

Implementa soporte avanzado para rotación de archivos basada en tamaño máximo o intervalos temporales definidos.

Fundamental para prevenir el crecimiento descontrolado de logs en entornos de producción y facilitar la gestión del ciclo de vida de los registros.

Observabilidad: ejemplo de logs

```

public void fetchAll() {
    logger.info("Descargando datos de bancos desde la URL: {}", url);

    RestTemplate restTemplate = new RestTemplate();

    try {
        restTemplate.getForObject(url, BankAPIResponse.class);
    } catch (Exception e) {
        logger.error("Error al descargar datos desde la API: {}", e.getMessage());
        return;
    }

    BankAPIResponse response = restTemplate.getForObject(url, BankAPIResponse.class);
    logger.info("Se han recibido {} bancos...", response.getBanks().size());
    syncBancos(response.getBanks());
}

private void syncBancos(List<Bank> banks) {
    for (Bank bank : banks) {

        Optional<Banco> optBanco = bancoRepo.findById(bank.id());

        if (optBanco.isPresent()) {
            bancoRepo.save(bancoMapper.updateBean(optBanco.get(), bank));
            logger.debug("Banco actualizado: {}", optBanco.get().getFullName());
        } else {
            // Crear un nuevo banco
            bancoRepo.save(bancoMapper.toBean(bank));
            logger.debug("Nuevo banco creado con nombre: {}", bank.id());
        }
    }
}

```

Observabilidad: plataformas de gestión de logs

Spring Boot ofrece integración flexible con diversas plataformas de gestión de logs como **GrayLog**, ELK Stack (Elasticsearch, Logstash, Kibana), Splunk, Papertrail y otras soluciones que facilitan la centralización y análisis de registros.

La configuración se realiza típicamente a través de appenders específicos en el archivo logback-spring.xml, permitiendo el envío de logs estructurados a estos sistemas mediante formatos como GELF, JSON o protocolos como Syslog.

Beneficios principales:

- Centralización de logs de múltiples instancias y microservicios en un único punto
- Capacidades avanzadas de búsqueda, filtrado y análisis de patrones
- Visualización y monitoreo a través de dashboards personalizados
- Configuración de alertas basadas en eventos específicos o umbrales predefinidos
- Retención configurable y gestión del ciclo de vida de los logs

Observabilidad: métricas



Métricas Técnicas

Monitoriza el consumo de recursos críticos como CPU, memoria, conexiones de base de datos, latencia de red, consumo de *tokens*, etc. para identificar cuellos de botella y anticipar problemas de rendimiento antes de que afecten a los usuarios finales.



Métricas del Dominio

Diseña métricas personalizadas enfocadas en conceptos específicos del negocio como usuarios activos, transacciones completadas o tiempo de procesamiento de pedidos, proporcionando visibilidad sobre el comportamiento real de la aplicación.



Estrategias de Muestreo

Implementa técnicas de muestreo adaptativo para métricas de alta frecuencia, reduciendo la sobrecarga en producción mientras mantienes la precisión estadística necesaria para análisis significativos del comportamiento del sistema.



Observabilidad: Micrometer

Micrometer es una fachada para sistemas de observabilidad que instrumenta aplicaciones JVM sin acoplarse a proveedores específicos, similar a SLF4J para métricas.



Integración nativa con Spring Boot Actuator para métricas estandarizadas.



Captura métricas dimensionales con etiquetas personalizables.



Abstrae sistemas como Prometheus, Datadog y New Relic con API unificada.



Registros modulares con conectores para diversos backends sin cambios de código.



Permite visualizaciones en Grafana y soporta métricas de negocio personalizadas.

Instrumenta aplicaciones con mínimo impacto en rendimiento, siendo esencial para implementar observabilidad en entornos productivos.

Observabilidad: Spring Boot Actuator

Habilitación

Agregar la dependencia **spring-boot-starter-actuator**.

En dev mode los endpoints suelen estar abiertos, mientras que en producción requieren autenticación.

Configuración

Configure endpoints expuestos con propiedades (management.endpoints.web.exposure.include=*). Es posible modificar la ruta base para mejorar la seguridad y restringir acceso externo.

/actuator/health

Muestra el estado de salud de la aplicación. Por defecto indica "UP" si funciona correctamente. Puede incluir verificaciones de bases de datos, colas y servicios externos.

/actuator/info

Expone información configurable: versión, descripción, entorno y otros datos relevantes definidos en las propiedades del proyecto.

/actuator/metrics

Proporciona métricas de rendimiento: uso de CPU, memoria, estadísticas HTTP, tiempos de respuesta y métricas personalizadas definidas programáticamente.

Otros endpoints

Incluye /env (variables de entorno), /threaddump (análisis de hilos), /loggers (configuración de logging) y /httptrace (seguimiento de peticiones). Muchos desactivados por defecto en producción.

Gestión de Logs

Permite modificar niveles de log en tiempo de ejecución mediante /actuator/loggers/{nombreLogger}, facilitando diagnósticos sin reiniciar la aplicación.

Observabilidad: configuración de Actuator

Habilitación

Para usar Actuator, agregar la dependencia **spring-boot-starter-actuator**. Por defecto, en Spring Boot dev mode muchos endpoints están abiertos, pero en producción la mayoría requieren autenticación.

Configuración

Se puede configurar qué endpoints exponer mediante propiedades (management.endpoints.web.exposure.include=). También es posible cambiar la ruta base (por seguridad, muchas veces se restringe el acceso externo a estos endpoints).

Observabilidad: tracing

Tracing Distribuido

Spring Boot se integra con herramientas de **tracing** distribuido para seguir peticiones a través de múltiples microservicios, facilitando la detección de problemas en arquitecturas complejas.

Spring Cloud Sleuth asigna identificadores únicos (trace ID y span ID) a cada solicitud, correlacionando eventos entre servicios. Zipkin complementa esta funcionalidad con almacenamiento y visualización de datos de tracing, esencial para identificar cuellos de botella.

Este enfoque mejora significativamente el diagnóstico en sistemas complejos y proporciona insights valiosos sobre el rendimiento entre servicios.

Otros: Spring Scheduling



Spring Scheduling

Permite programar tareas para ejecutarse periódicamente o en momentos específicos.

Ideal para procesos batch, informes y mantenimiento automatizado.



@Scheduled

Anotación principal para configurar la ejecución programada de métodos.

Soporta expresiones cron, intervalos fijos y retrasos configurables.



Configuración

Requiere habilitar scheduling con `@EnableScheduling` en una clase de configuración.

Permite definir pools de hilos dedicados para tareas programadas.

```
@SpringBootApplication  
@EnableScheduling  
public class SpringavanzadoApplication {  
}
```

```
@Service  
public class BancoService {  
...  
    @Scheduled(cron = "0 * * * *") // Cada minuto  
    public void scheduledFetching() {  
        fetchAll();  
    }  
...  
}
```

Otros: Spring Mail

1 Integración Simple

Spring Mail facilita el envío de correos electrónicos desde aplicaciones Java.

Se integra perfectamente con cualquier aplicación Spring Boot.

2 Características Principales

Soporta correos con HTML, adjuntos y plantillas.

Permite configurar servidores SMTP con autenticación y TLS/SSL.

3 Implementación

Utiliza JavaMailSender para enviar mensajes de forma programática.

Ofrece plantillas con Thymeleaf o FreeMarker para correos dinámicos.

application.properties

```
spring.mail.properties.mail.smtp.timeout=2000  
spring.mail.host=smtp.gmail.com  
spring.mail.port=587  
spring.mail.username=username  
spring.mail.password=password
```

Uso

```
@Autowired  
private JavaMailSender mailSender;  
  
public void enviarCorreo() {  
    SimpleMailMessage mensaje = new SimpleMailMessage();  
    mensaje.setTo("destinatario@example.com");  
    mensaje.setSubject("Asunto del correo");  
    mensaje.setText("Contenido del mensaje");  
    mailSender.send(mensaje);  
}
```

Otros: Spring Async Methods



Operaciones No Bloqueantes

Spring permite ejecutar métodos de forma asíncrona sin bloquear el hilo principal de ejecución.



@Async

Con solo añadir esta anotación, los métodos se ejecutarán en hilos separados del pool de Spring.



CompletableFuture

Permite encadenar acciones asíncronas y manejar resultados cuando estén disponibles.

Configuración

```
@Configuration
@EnableAsync
public class AsyncConfig {
    @Bean
    public Executor taskExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(5);
        executor.setMaxPoolSize(10);
        return executor;
    }
}
```

Uso

```
@Service
public class EmailService {
    @Async
    public CompletableFuture enviarEmail(String destino) {
        // lógica de envío
        return CompletableFuture.completedFuture(true);
    }
}
```

Otros: Spring AI

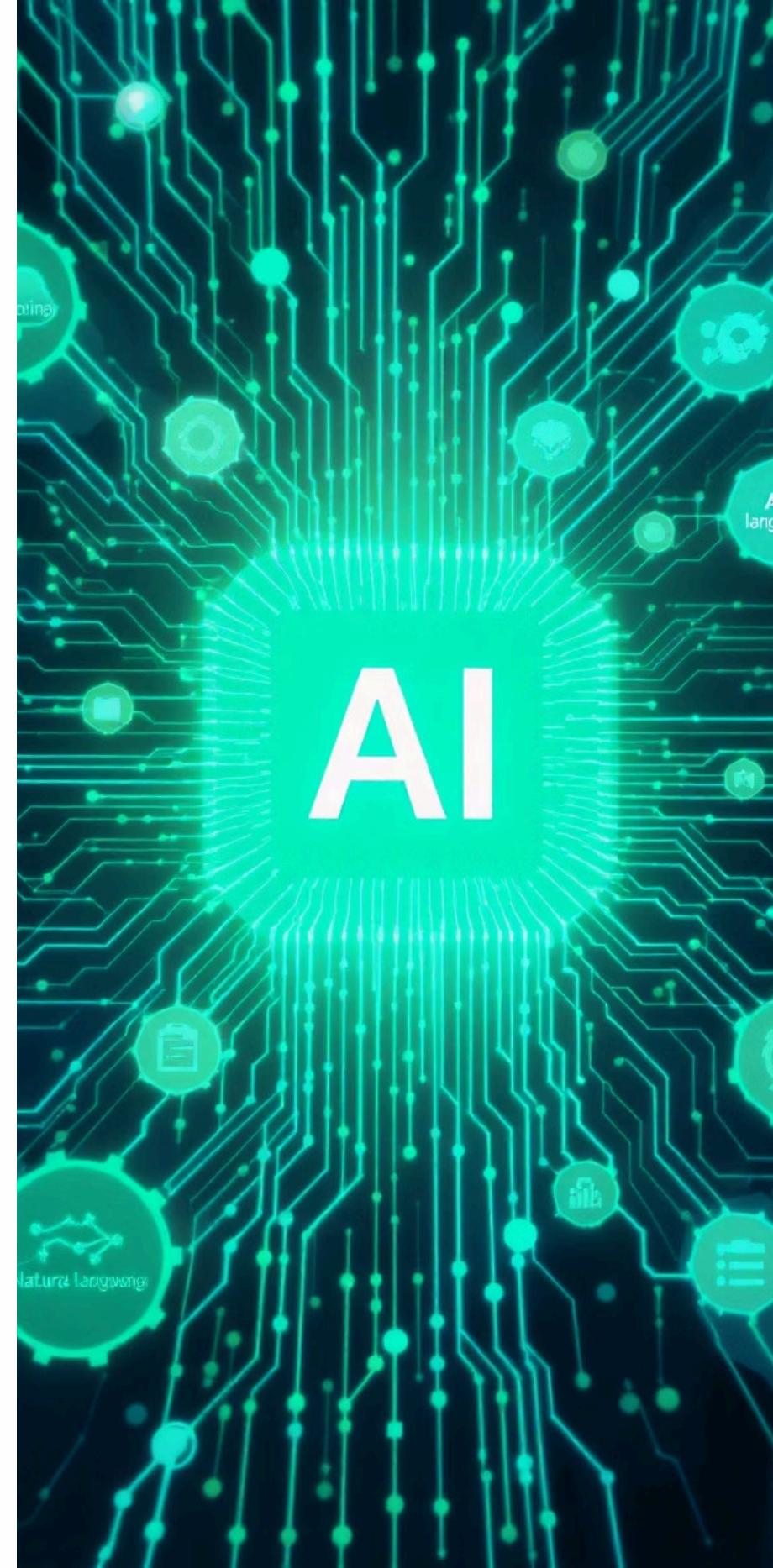
Framework que simplifica la integración de modelos de lenguaje (LLMs).

Es un módulo experimental en constante evolución.

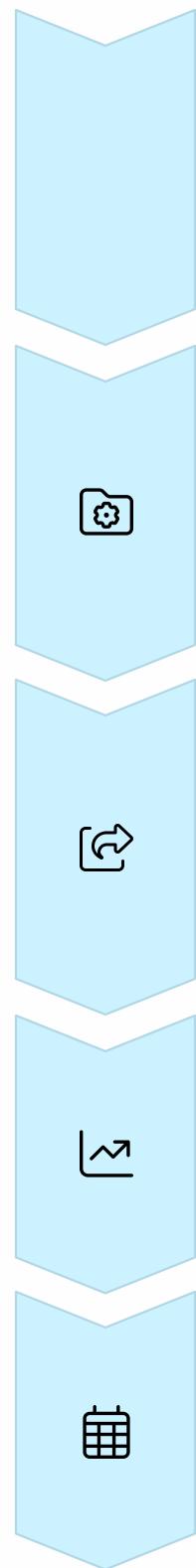
Permite desarrollar fácilmente soluciones como chatbots, generadores de contenido, analizadores de sentimientos, clasificadores de documentos, resumidores de texto, etc..

Características principales:

- Integración nativa con múltiples proveedores: OpenAI, Azure OpenAI, Amazon Bedrock, Google, Anthropic y otros servicios.
- API unificada que garantiza consistencia entre diferentes proveedores sin necesidad de modificar código
- Capacidades generativas implementadas mediante anotaciones declarativa.
- Sistema avanzado de procesamiento de prompts con plantillas parametrizadas para interacciones más precisas.
- Conversión automática de respuestas de IA a objetos Java tipados para integración fluida con el dominio de aplicación.
- Soporte para embeddings que habilita búsquedas semánticas y comparación contextual de contenidos.
- Herramientas de gestión de tokens que optimizan costes y rendimiento en entornos de producción.
- Compatibilidad con ModelContext Protocol para la creación de agentes que se conecten con herramientas externas.



Otros: Spring Batch



Lectura de Datos

Extracción desde múltiples fuentes (BBDD, archivos, XML, JSON) con paginación eficiente.

Procesamiento

Transformación con manejo de errores, procesamiento paralelo y políticas de reintento.

Escritura

Almacenamiento con transacciones, reinicio y escritura por lotes para maximizar rendimiento.

Monitorización

Registro con métricas de rendimiento, visibilidad de progreso y estadísticas.

Programación

Integración con Spring Scheduler o Quartz para ejecución según calendario.

Framework para procesamiento masivo de datos con registro, reinicio y gestión de transacciones.

Ideal para procesos nocturnos, migraciones e informes. Resuelve:

- Procesamiento de millones de registros
- Recuperación tras fallos
- Particionamiento para paralelismo
- Seguimiento de trabajos

Componentes clave:

- **Job**: Unidad de trabajo con pasos secuenciales
- **Step**: Fase de lectura, procesamiento y escritura
- **JobRepository**: Almacén de metadatos
- **ItemReader/ItemWriter**: Interfaces para fuentes y destinos

Escala desde procesamiento simple hasta distribución en múltiples nodos.

Otros: Spring Cloud

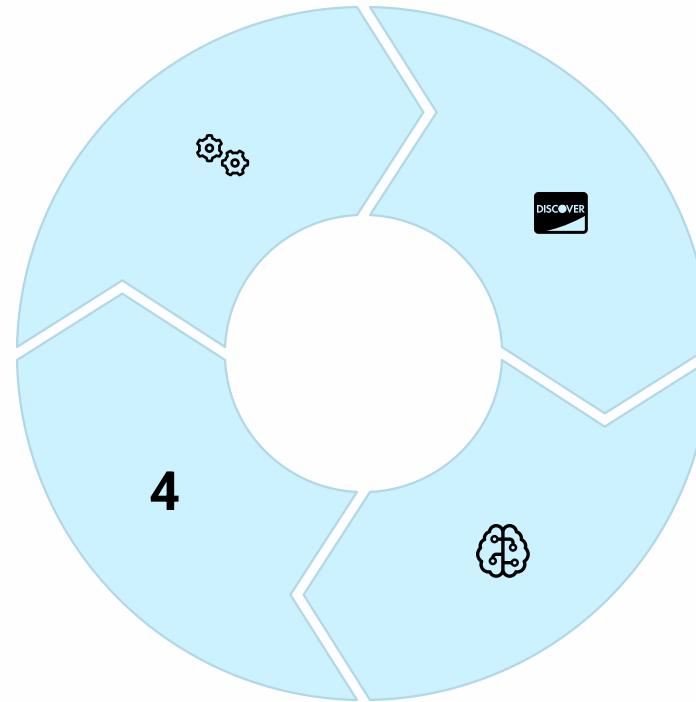
Spring Cloud ofrece herramientas para construir sistemas distribuidos basados en patrones como:

Spring Cloud Config

Gestión centralizada de configuraciones externas con soporte para versionado, cifrado y actualización en tiempo real.

API Gateway

Punto de entrada único para enrutamiento y filtrado de peticiones, proporcionando autenticación, autorización, limitación de tasa y balanceo de carga centralizado.



Eureka

Servicio de descubrimiento que permite el registro y localización dinámica entre microservicios, facilitando el balanceo de carga sin configuraciones estáticas.

Circuit Breakers

Implementa patrones de tolerancia a fallos con Resilience4j, previniendo fallos en cascada mediante circuit breaker, rate limiter y otros mecanismos de resiliencia.

Estos componentes facilitan la creación de arquitecturas de microservicios resilientes y escalables en la nube.

Otros: Spring Reactive

La programación reactiva permite mejorar la utilización de recursos, mayor capacidad de respuesta y escalabilidad mejorada, mediante el uso de flujos de datos asíncronos



WebFlux

Alternativa no bloqueante a Spring MVC. Utiliza Project Reactor con tipos Flux y Mono.

WebClient

Cliente HTTP reactivo que reemplaza a RestTemplate. Ofrece operaciones sincrónicas y asíncronas.

Otros: lo que no te cuentan en los tutoriales de JPA

| | | |
|---|---|--|
|  Caché de Primer Nivel <p>El contexto de persistencia (EntityManager) mantiene una caché interna de entidades cargadas durante la transacción activa.</p> <p>Reduce consultas redundantes y mejora el rendimiento significativamente sin requerir configuración adicional.</p> |  Pool de Conexiones <p>HikariCP optimiza el rendimiento mediante la gestión avanzada del ciclo de vida de conexiones a la base de datos.</p> <p>Parámetros críticos como maximumPoolSize, connectionTimeout y leakDetectionThreshold impactan directamente en la escalabilidad y estabilidad del sistema.</p> |  Concurrencia Optimista <p>El control de versiones mediante @Version detecta modificaciones simultáneas sin necesidad de implementar bloqueos de base de datos.</p> <p>Garantiza la integridad de datos en entornos multiusuario lanzando excepciones controladas cuando se detectan conflictos de modificación concurrente.</p> |
|  Consultas Optimizadas <p>Implementa proyecciones (DTO, interfaces) y consultas nativas (SQL/JPQL) para minimizar la transferencia de datos entre la base de datos y la aplicación.</p> <p>Permite reducir el tiempo de respuesta y mejorar la eficiencia de la aplicación.</p> |  Carga Perezosa <p>La estrategia FetchType.LAZY optimiza el uso de memoria al cargar relaciones únicamente cuando se accede explícitamente a ellas.</p> <p>Fundamental para aplicaciones con modelos de datos complejos y múltiples relaciones anidadas.</p> |  N+1 Queries <p>Este problema ocurre cuando se ejecuta una consulta principal seguida de múltiples consultas adicionales para cargar entidades relacionadas.</p> <p>Uso de JOIN FETCH en JPQL o la anotación @EntityGraph permite cargar relaciones en una sola consulta eficiente.</p> <p>Los @EntityGraph de entidad permiten definir estrategias de carga granulares por consulta en vez de a nivel de entidad.</p> |

Resumen

Seguridad

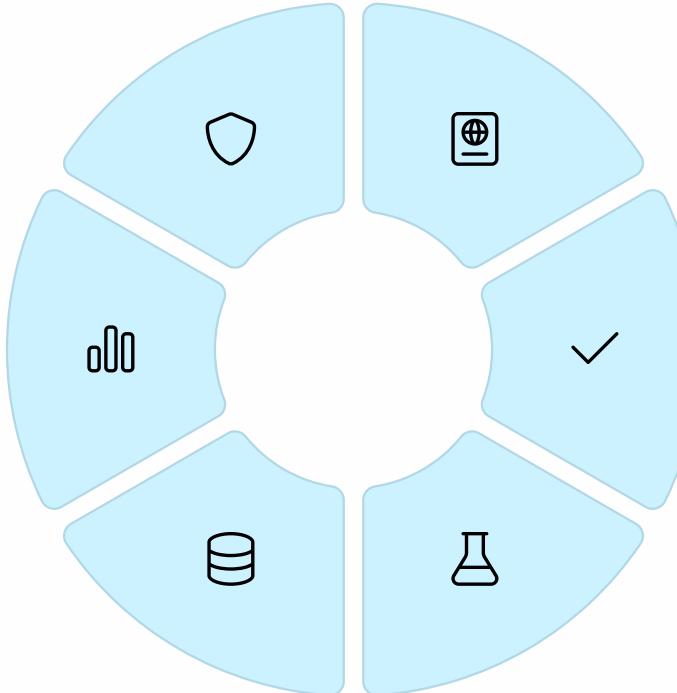
Spring Security proporciona autenticación y autorización robustas mediante filtros configurables.

Observabilidad

Logs, métricas y tracing permiten monitorizar aplicaciones en producción.

Transacciones

Gestión declarativa con `@Transactional` y propagación configurable entre métodos.



Documentación

OpenAPI/Swagger facilita la creación de documentación API interactiva y mantenible.

Validaciones

JSR-380 permite validar datos de entrada con anotaciones declarativas.

Pruebas

JUnit con principios FIRST facilita tests unitarios y de integración.

Estos componentes no funcionales son esenciales para crear aplicaciones Spring robustas y productivas. Complementan la lógica de negocio y garantizan calidad, seguridad y mantenibilidad.