Universidad Icesi

Computación en Internet 2 — Semana 6

Paginación JPA, Transacciones ACID & Testing con Mockito

Arquitectura Backend Empresarial con Spring Boot

Facultad de Ingeniería — Universidad Icesi

Universidad ICESI

Agenda y Tres Pilares de la Semana 6

1. Paginación y Ordenamiento

Spring Data JPA

- Evitar caídas de memoria por `findAll()` masivo.
- Interfaces `Pageable`, `PageRequest` y `Sort`.
- Traducción a `LIMIT` y `OFFSET` en SQL.

2. Gestión de Transacciones

@Transactional & Rollback

- Principios ACID y atomicidad "todo o nada".
- Interceptación con Spring AOP Proxy.
- Reglas de Rollback por defecto vs `rollbackFor`.

3. Pruebas Automatizadas

JUnit 5 & Mockito

- Pruebas unitarias ultrarrápidas con `@Mock`.
- Estructura con patrón AAA (Arrange-Act-Assert).
- Pruebas de integración con `@SpringBootTest` y H2.

Universidad ICESI

01. Paginación y Ordenamiento en Spring Data JPA

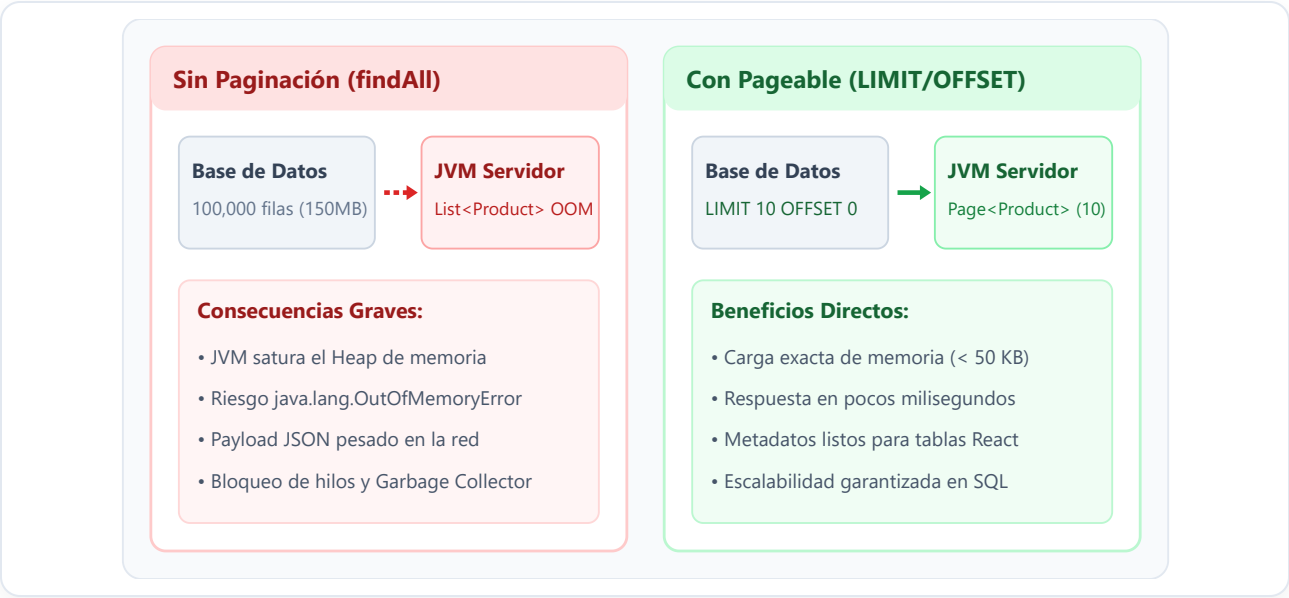
Segmentación eficiente de datos, consultas SQL escalables y rendimiento

El Problema de las Consultas Masivas: findAll()

¿Por qué no usar findAll()?

En aplicaciones reales con cientos de miles de registros, ejecutar `SELECT *` satura la infraestructura:

- **JVM Heap Exhaustion:** Se instancian miles de entidades en memoria, disparando `java.lang.OutOfMemoryError`.
- **Sobrecarga de Red:** Se transfieren decenas de megabytes JSON que el frontend ni siquiera puede pintar a la vez.
- **Solución (Paginación):** Segmentar datos en páginas discretas procesadas directamente por el motor SQL.



Arquitectura de Paginación en Spring Data

PAGEABLE (INTERFAZ)

Contrato que encapsula los parámetros de paginación:

- **page:** Índice de página (inicia en **0**).
- **size:** Cantidad de ítems solicitados.
- **sort:** Criterio de ordenación.

PAGEREQUEST (FÁBRICA)

Implementación concreta comúnmente utilizada:

```
Pageable p = PageRequest.of(page, size, sort);
```

Instanciación inmutable de la solicitud.

PAGE (CONTEO TOTAL)

Contenedor con datos y **metadatos calculados**:

- `getContent()` : Lista de entidades.
- `getTotalElements()` y `getTotalPages()` .
- Ejecuta un `COUNT(*)` adicional.

SLICE (LIGERO)

Alternativa eficiente a `Page<T>` :

- **NO** ejecuta `COUNT(*)` masivo.
- Sabe si hay siguiente página: `hasNext()` .
- Ideal para **scroll infinito** en móviles.

Universidad ICESI

- LIMIT & OFFSET
- Ejecución en Motor SQL
- Pageable a Dialecto

Traducción Nativa a Consultas SQL

Spring Data JPA no filtra en memoria: delega la segmentación directamente al optimizador del motor SQL mediante cláusulas nativas:

```
-- Consulta 1: Obtener la rebanada de 10 productos (Página 0)
SELECT * FROM products
ORDER BY price DESC
LIMIT 10 OFFSET 0;

-- Consulta 2: Calculada automáticamente por Page<T>
SELECT COUNT(*) FROM products;
```

Fórmula de OFFSET

`OFFSET = page * size` . Para la página 2 con tamaño 10: `LIMIT 10 OFFSET 20` .

Modelo JPA y Repositorio Paginado

Entidad JPA: `Product.java`

```java
@Entity
@Table(name = "products")
@Getter @Setter @NoArgsConstructor @AllArgsConstructor
public class Product {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  private String name;
  private String category;
  private Double price;
  private Integer stock;
}
```

Entidad estándar de persistencia mapeada a la tabla `products`.

Repositorio: `ProductRepository.java`

```java
@Repository
public interface ProductRepository
    extends JpaRepository<Product, Long> {

  // JpaRepository ya hereda Page<T> findAll(Pageable pageable)

  // Query method paginado por categoría
  Page<Product> findByCategory(
    String category,
    Pageable pageable
  );

  // Búsqueda insensible a mayúsculas paginada
  Page<Product> findByNameContainingIgnoreCase(
    String keyword,
    Pageable pageable
  );
}
```

Herencia Clave: `JpaRepository` hereda de `PagingAndSortingRepository`.

Capa de Servicio: Construcción de Pageable

Lógica de Negocio y Orden

En el servicio se reciben los parámetros desacoplados, se normaliza la dirección y se crea la petición de página:

- **Normalización de Sort:** Validación segura de `Sort.Direction.ASC` o `DESC`.
- **Inmutabilidad:** `PageRequest.of(...)` genera la instancia lista para enviar al repositorio.
- **Retorno con Metadatos:** Devuelve la colección empaquetada en `Page<Product>`.

ARCHIVO: `PRODUCTSERVICE.JAVA`

```java
@Service
public class ProductService {
  private final ProductRepository productRepository;

  @Autowired
  public ProductService(ProductRepository productRepository) {
    this.productRepository = productRepository;
  }

  public Page<Product> getProducts(
      int page, int size, String sortBy, String direction) {

    // 1. Determinar dirección de forma segura
    Sort.Direction sortDirection = direction.equalsIgnoreCase("desc")
        ? Sort.Direction.DESC
        : Sort.Direction.ASC;

    // 2. Configurar objeto Sort por propiedad
    Sort sort = Sort.by(sortDirection, sortBy);

    // 3. Instanciar Pageable con página, tamaño y orden
    Pageable pageable = PageRequest.of(page, size, sort);

    // 4. Ejecutar consulta SQL con LIMIT y OFFSET
    return productRepository.findAll(pageable);
  }
}
```

Capa Controlador REST: Exposición del Endpoint

Recepción de Parámetros HTTP

El endpoint REST expone parámetros de consulta en la URL mediante `@RequestParam` con valores por defecto seguros:

Parámetro	Default	Propósito
`page`	0	Página actual (índice 0)
`size`	10	Ítems por página
`sortBy`	"id"	Atributo de ordenamiento
`direction`	"asc"	Dirección (asc / desc)

Ejemplo URL:

```
GET /api/products?page=0&size=5&sortBy=price&direction=desc
```

ARCHIVO: `PRODUCTCONTROLLER.JAVA`

```java
@RestController
@RequestMapping("/api/products")
public class ProductController {

  private final ProductService productService;

  @Autowired
  public ProductController(ProductService productService) {
    this.productService = productService;
  }

  @GetMapping
  public ResponseEntity<Page<Product>> getAllProducts(
      @RequestParam(defaultValue = "0") int page,
      @RequestParam(defaultValue = "10") int size,
      @RequestParam(defaultValue = "id") String sortBy,
      @RequestParam(defaultValue = "asc") String direction) {

    Page<Product> productPage =
      productService.getProducts(page, size, sortBy, direction);

    return ResponseEntity.ok(productPage);
  }
}
```

Anatomía del Payload JSON: Page

Contenido + Metadatos

La serialización de `Page<T>` produce un JSON auto-descriptivo listo para componentes de paginación en React o Vue:

- ✓ `content` : Array con los registros específicos de la página.
- ✓ `totalElements` : Total de filas existentes en la tabla (150).
- ✓ `totalPages` : Cantidad de páginas posibles (`150 / 2 = 75`).
- ✓ `first` / `last` : Banderas booleanas para deshabilitar botones Anterior / Siguiente.

RESPUESTA HTTP 200 OK


```
{
 "content": [
  {"id": 84, "name": "Portátil Pro", "price": 4500000.0, "stock": 12},
  {"id": 12, "name": "Monitor 34"", "price": 2800000.0, "stock": 5}
 ],
 "pageable": {
  "pageNumber": 0,
  "pageSize": 2,
  "offset": 0,
  "paged": true
 },
 "totalElements": 150,
 "totalPages": 75,
 "last": false,
 "first": true,
 "size": 2,
 "number": 0,
 "numberOfElements": 2,
 "empty": false
}
```

Universidad ICESI

Buenas Prácticas en Paginación

LIMITAR EL SIZE MÁXIMO

Evitar ataques de denegación de servicio (DoS):

```
int safeSize = Math.min(size, 100);
```

Nunca permitir peticiones como `size=100000`.

INDEXACIÓN EN BASE DE DATOS

Optimizar la cláusula `ORDER BY`:

Crea índices B-Tree en SQL para columnas frecuentes como `created_at` o `price`.

EVALUAR SLICE

Evitar el costo del `COUNT(*)`:

En feeds y listas infinitas, `Slice` ahorra la segunda consulta masiva sobre millones de filas.

MAPEO A DTOS

Uso de `page.map()` nativo:

```
return page.map(ProductDTO::new);
```

Transforma entidades a DTOs conservando metadatos.

Universidad ICESI

02. Gestión de Transacciones con @Transactional

Principios ACID, interceptación Spring AOP y control de Rollback

Fundamentos: Las 4 Propiedades ACID

ATOMICIDAD (ATOMICITY)

Principio: Todo o Nada

La operación se completa íntegramente o se revierte por completo. Si se descuenta saldo de una cuenta pero falla el abono, no queda ningún cambio aplicado.

CONSISTENCIA (CONSISTENCY)

Estado Válido a Estado Válido

La base de datos respeta siempre todas las reglas de integridad referencial, checks, restricciones de no-nulo y llaves foráneas antes y después de la transacción.

AISLAMIENTO (ISOLATION)

Transacciones Concurrentes

Múltiples operaciones simultáneas no interfieren entre sí ni leen modificaciones parciales intermedias ("dirty reads") generadas por otros hilos.

DURABILIDAD (DURABILITY)

Persistencia Confirmada

Una vez que el motor ejecuta `COMMIT`, los datos quedan escritos permanentemente en el disco y sobreviven a apagones o caídas del servidor.

El Ciclo de Vida: Commit vs Rollback

Los Dos Caminos Transaccionales

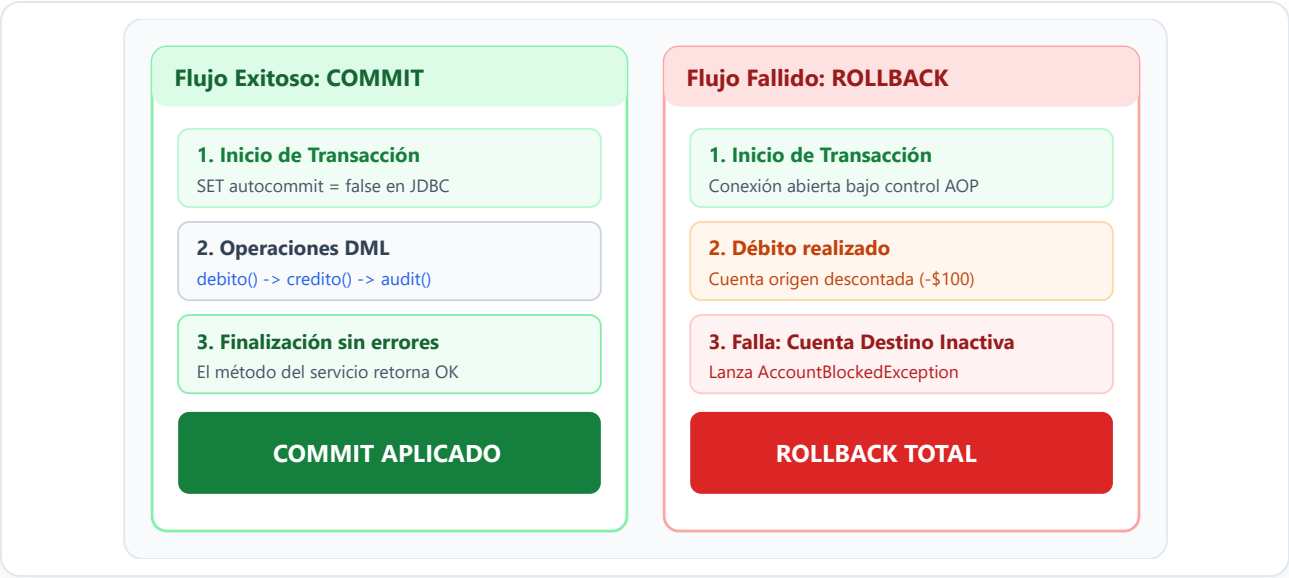
Toda transacción inicia abriendo una conexión JDBC con el flag `autocommit = false`:

Camino A: COMMIT (Éxito)

Si el método retorna sin excepciones, el proxy solicita confirmar todas las mutaciones en disco.

Camino B: ROLLBACK (Falla)

Si ocurre una excepción configurada, se descartan todos los cambios y la BD vuelve a su estado inicial intacto.

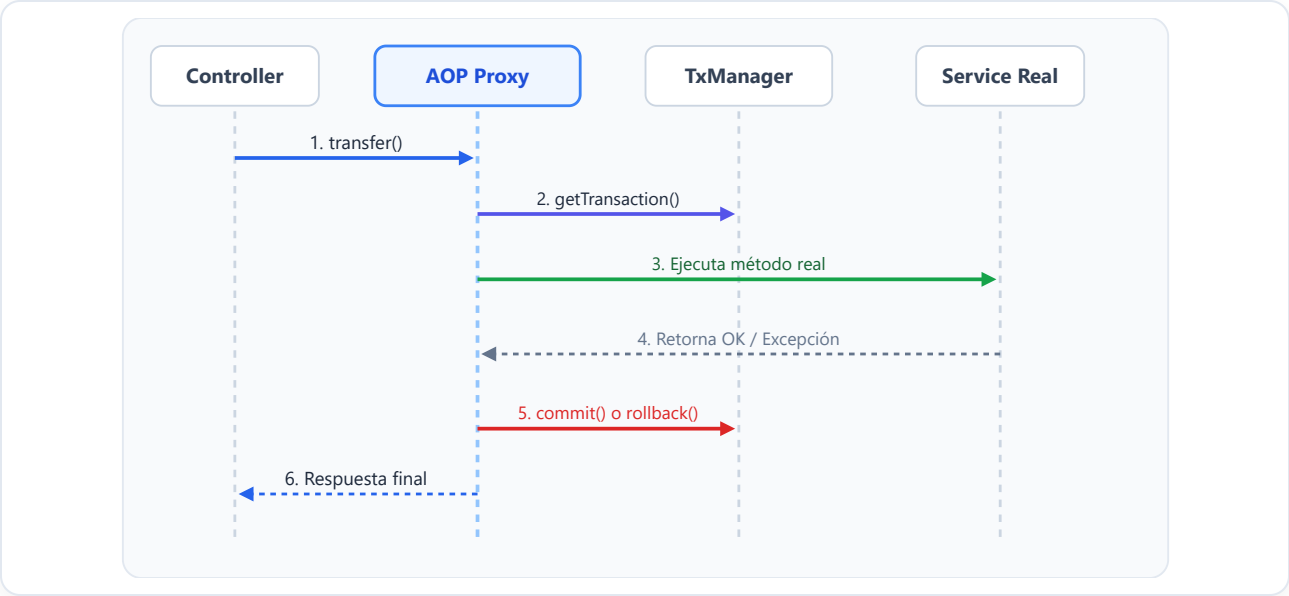


Arquitectura: El Proxy Transaccional de Spring

Patrón Proxy con Spring AOP

Cuando anotas una clase con `@Transactional`, Spring inyecta un **Proxy** intermedio en el ApplicationContext:

- El Controller interactúa con el Proxy sin saberlo.
- El Proxy solicita a `PlatformTransactionManager` una conexión transaccional.
- Se invoca el método real en el Service Bean.
- Si captura error aplica `ROLLBACK`; si no, `COMMIT`.



RuntimeException -> Rollback

Checked Ex -> No Rollback

rollbackFor = Exception

Reglas de Rollback: Por Defecto vs rollbackFor

¡Comportamiento Crítico por Defecto!

Por defecto, Spring **ÚNICAMENTE** aplica Rollback ante excepciones **no verificadas** (hijas de `RuntimeException` y `Error`).

Si un método lanza una excepción verificada (ej. `AccountBlockedException extends Exception`), Spring **NO** revertirá los cambios a menos que se declare explícitamente:

```
// Obliga a revertir ante cualquier excepción de negocio
@Transactional(rollbackFor = {AccountBlockedException.class, Exception.class})
```

Usa `noRollbackFor` cuando el fallo sea irrelevante (ej. error enviando correo de confirmación secundario).

Universidad ICESI

Parámetros Clave de @Transactional

ROLLBACKFOR

Fuerza la reversión ante excepciones verificadas:

```
rollbackFor = Exception.class
```

Garantiza consistencia absoluta en lógica bancaria o de pagos.

READONLY = TRUE

Optimización de consultas:

Informa a Hibernate que no habrá mutaciones. **Desactiva el dirty checking** en entidades y libera recursos de sesión JDBC más rápido.

PROPAGATION

Comportamiento con transacciones activas:

- `REQUIRED` (default): Se une o crea nueva.
- `REQUIRES_NEW`: Suspende la actual y crea una independiente.

ISOLATION

Control de concurrencia SQL:

Define niveles de aislamiento (`READ_COMMITTED` , `REPEATABLE_READ` , `SERIALIZABLE`) para evitar lecturas sucias o fantasmas.

Caso de Negocio: Transferencia Bancaria Atómica

Reglas del Dominio

- ✓ Validar que ambas cuentas existan y estén **activas**.
- ✓ Verificar que la cuenta de origen tenga saldo `>= monto`.
- ✓ Descontar saldo de origen (débito).
- ✓ Acreditar saldo en destino.
- ⓘ Si la cuenta destino está bloqueada, revertir todo inmediatamente.

Entidad `BankAccount.java`

```java
@Entity
@Table(name = "bank_accounts")
@Getter @Setter @NoArgsConstructor @AllArgsConstructor @Builder
public class BankAccount {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  private String accountNumber;
  private String ownerName;
  private Double balance;
  private Boolean active;
}
```

> Excepciones: `InsufficientFundsException` (Runtime) y `AccountBlockedException` (Checked).

Implementación de TransferService con Rollback

Paso a Paso de la Operación

- **1. Verificación:** Obtención de cuentas origen y destino de la BD.
- **2. Validación de saldo:** Lanza `InsufficientFundsException` si el saldo es menor.
- **3. Mutación 1:** Se aplica débito a la cuenta de origen.
- **4. Falla simulada:** Si la cuenta de destino está inactiva, se dispara `AccountBlockedException`.
- **5. Rollback:** `rollbackFor` garantiza que el débito sea descartado.

```java
@Transactional(rollbackFor = {AccountBlockedException.class, Exception.class})
public void transferMoney(Long srcId, Long dstId, Double amount)
    throws AccountBlockedException {

  BankAccount src = accountRepository.findById(srcId)
    .orElseThrow(() -> new IllegalArgumentException("Origen no existe"));
  BankAccount dst = accountRepository.findById(dstId)
    .orElseThrow(() -> new IllegalArgumentException("Destino no existe"));

  if (src.getBalance() < amount) {
    throw new InsufficientFundsException("Saldo insuficiente: $" + amount);
  }

  // Débito en origen
  src.setBalance(src.getBalance() - amount);
  accountRepository.save(src);

  // Validación de cuenta destino
  if (Boolean.FALSE.equals(dst.getActive())) {
    throw new AccountBlockedException("Cuenta destino inactiva");
  }

  // Crédito en destino
  dst.setBalance(dst.getBalance() + amount);
  accountRepository.save(dst);
}
```

Antipatrón Común: Auto-Invocación (Self-Invocation)

¿Por qué falla @Transactional?

Si un método no transaccional llama a otro método con `@Transactional` dentro del **mismo Bean** (`this.metodoB()`):

¡La transacción NUNCA se iniciará!

La llamada ocurre de forma directa dentro del objeto Java en memoria, saltándose el Proxy de Spring AOP.

- ✓ **Solución:** Trasladar la operación transaccional a un servicio independiente e inyectarlo.
- ✓ **Evitar I/O Lento:** No realizar llamadas HTTP a APIs externas dentro de un bloque `@Transactional`.

CÓDIGO ERRÓNEO (OMITE PROXY)

```
@Service
public class OrderService {

  public void procesarPedido() {
    // ❌ ERROR: Llamada directa a 'this'
    // El Proxy AOP nunca intercepta la llamada
    this.guardarEnBaseDeDatos();
  }

  @Transactional
  public void guardarEnBaseDeDatos() {
    // Esta anotación es ignorada en auto-invocación
    repository.save(order);
  }
}
```

Regla de Oro: Las anotaciones AOP de Spring sólo actúan cuando la invocación entra a través del Proxy externo.

Universidad ICESI

03. Pruebas Unitarias y de Integración con Mockito

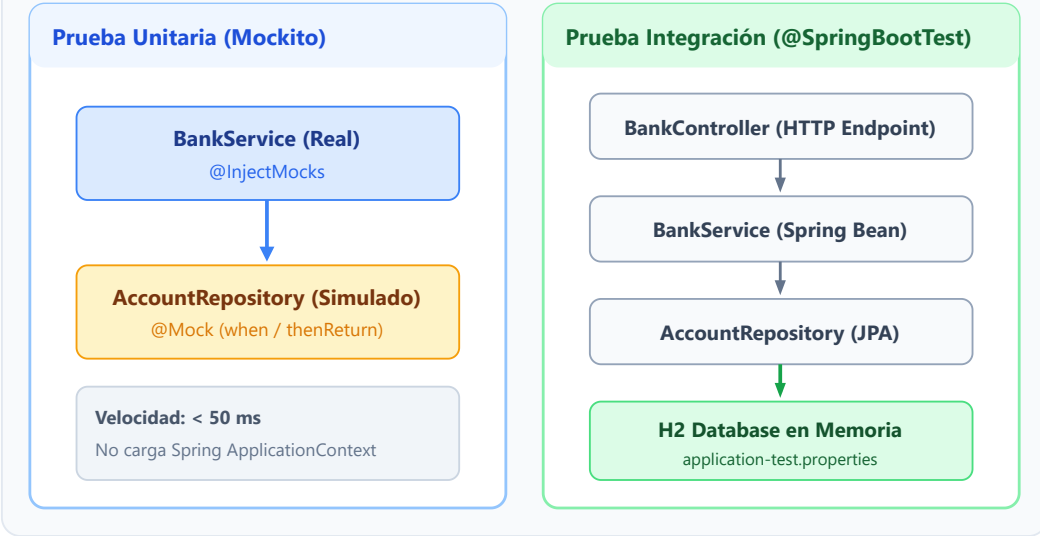
JUnit 5, simulación de dependencias, base de datos H2 y Maven Wrapper

Pruebas Unitarias vs. Pruebas de Integración

Diferencias Fundamentales

Criterio	Unitarias (Mockito)	Integración (Spring)
Alcance	1 clase aislada	Múltiples capas
Spring Context	No se carga	`@SpringBootTest`
Dependencias	Mocks simulados	Beans reales + H2
Velocidad	< 50 ms	Segundos por suite
Propósito	Lógica de negocio	SQL, JPA y Transacciones

Estrategia Piramidal: Muchas pruebas unitarias rápidas y una cantidad controlada de pruebas de integración críticas.



Universidad ICESI

@Mock simulado

@InjectMocks real

Sin Spring Context

Aislamiento con Mockito: @Mock vs @InjectMocks

Mockito permite probar el comportamiento de un servicio simulando sus dependencias sin necesidad de conectarse a una base de datos real:

@Mock (Dependencia Ficticia)

Crea un cascarón simulado de `AccountRepository`. Responde exclusivamente con lo que programes con `when(...).thenReturn(...)`.

@InjectMocks (Clase Bajo Prueba)

Crea la instancia real de `BankService` e inyecta automáticamente en ella todos los `@Mock` definidos.

Se activa con `@ExtendWith(MockitoExtension.class)` en JUnit 5.

El Patrón AAA: Arrange — Act — Assert

1. Arrange (Preparación)

Configurar datos de entrada y comportamiento

Se crean los objetos de prueba y se programa la respuesta esperada de los mocks:

```
Account src = new Account(1L, 100.0);
Account dst = new Account(2L, 50.0);

when(repo.findById(1L))
  .thenReturn(Optional.of(src));
when(repo.findById(2L))
  .thenReturn(Optional.of(dst));
```

2. Act (Ejecución)

Invocación del método bajo prueba

Se llama al método concreto que se está validando pasándole los parámetros correspondientes:

```
// Invocamos la unidad bajo prueba
boolean resultado =
  bankService.transfer(1L, 2L, 30.0);
```

Esta fase debe ser concisa, idealmente una sola línea de código.

3. Assert (Verificación)

Comprobación de estado e interacciones

Se valida el retorno y se comprueba que el repositorio fue llamado adecuadamente:

```
assertTrue(resultado);
assertEquals(70.0, src.getBalance());

// Verificar llamadas al mock
verify(repo, times(1)).save(src);
verify(repo, times(1)).save(dst);
```

Prueba Unitaria: Camino Exitoso con Mockito

Estructura de la Clase de Prueba

- Anotada con `@ExtendWith(MockitoExtension.class)`.
- Inyección de dependencias simuladas con `@Mock` y `@InjectMocks`.
- Uso de `@DisplayName` para describir la intención semántica del test.
- Verificación de que `accountRepository.save(...)` se ejecutó exactamente una vez por cuenta.

BANKSERVICETEST.JAVA (TEST EXITOSO)

```java
@Test
@DisplayName("Debe transferir exitosamente con saldo suficiente")
void shouldTransferFundsSuccessfully() {
  // 1. Arrange
  Account src = new Account(1L, "Carol", 100.0);
  Account dst = new Account(2L, "Juan", 50.0);

  when(accountRepository.findById(1L)).thenReturn(Optional.of(src));
  when(accountRepository.findById(2L)).thenReturn(Optional.of(dst));

  // 2. Act
  boolean result = bankService.transfer(1L, 2L, 30.0);

  // 3. Assert
  assertTrue(result);
  assertEquals(70.0, src.getBalance());
  assertEquals(80.0, dst.getBalance());

  // Verificación de interacciones con el mock
  verify(accountRepository, times(1)).save(src);
  verify(accountRepository, times(1)).save(dst);
}
```

Prueba Unitaria: Fallos y Excepciones

CASO 1: SALDO INSUFICIENTE

```java
@Test
void shouldReturnFalseWhenInsufficientBalance() {
  Account src = new Account(1L, "Carol", 20.0);
  Account dst = new Account(2L, "Juan", 50.0);

  when(accountRepository.findById(1L)).thenReturn(Optional.of(src));
  when(accountRepository.findById(2L)).thenReturn(Optional.of(dst));

  boolean result = bankService.transfer(1L, 2L, 100.0);

  assertFalse(result);
  assertEquals(20.0, src.getBalance());
  // Garantiza que NUNCA se guardaron cambios
  verify(accountRepository, never()).save(any(Account.class));
}
```

CASO 2: MONTO NEGATIVO (ASSERTTHROWS)

```java
@Test
void shouldThrowExceptionWhenAmountIsNegative() {
  // Valida que el método lance la excepción esperada
  assertThrows(IllegalArgumentException.class, () -> {
    bankService.transfer(1L, 2L, -10.0);
  });

  // Garantiza que ni siquiera se consultó la BD
  verifyNoInteractions(accountRepository);
}
```

`verifyNoInteractions(mock)` valida que ninguna operación tocó el repositorio ante argumentos inválidos.

Pruebas de Integración con @SpringBootTest y H2

APPLICATION-TEST.PROPERTIES

```
# Base de datos en memoria H2 aislada
spring.datasource.url=jdbc:h2:mem:testdb;DB_CLOSE_DELAY=-1
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=

# Regenera el esquema en cada ejecución
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=true
```

@Transactional en Tests: Cada inserción realizada durante la prueba se revierte al terminar.

ACCOUNTREPOSITORYINTEGRATIONTEST.JAVA

```java
@SpringBootTest
@ActiveProfiles("test")
@Transactional // Rollback automático tras cada test
class AccountRepositoryIntegrationTest {

  @Autowired
  private AccountRepository accountRepository;

  @Test
  @DisplayName("Debe persistir y recuperar una cuenta en H2")
  void shouldPersistAndRetrieveAccount() {
    Account account = new Account(null, "Diana", 450.0);

    Account saved = accountRepository.save(account);
    assertNotNull(saved.getId());

    Optional<Account> retrieved =
      accountRepository.findById(saved.getId());

    assertTrue(retrieved.isPresent());
    assertEquals("Diana", retrieved.get().getOwnerName());
  }
}
```

Ejecución y Automatización con Maven Wrapper

¿Por qué usar el Maven Wrapper (mvnw)?

El wrapper asegura que todos los desarrolladores y los pipelines de CI/CD ejecuten exactamente la misma versión estandarizada de Maven sin instalación previa.

- Cero diferencias de versión entre entornos Windows, Mac o Linux.
- Generación de reportes automáticos en `target/surefire-reports/`.
- Integración nativa con pipelines de GitHub Actions y GitLab CI.

COMANDOS DE TERMINAL

```
# 1. Ejecutar toda la suite de pruebas
.mvnw.cmd test      # Windows
./mvnw test         # Linux / macOS

# 2. Ejecutar una clase de prueba específica
.mvnw.cmd test -Dtest=BankServiceTest

# 3. Ejecutar un único método puntual
.mvnw.cmd test -Dtest=BankServiceTest#shouldTransferFundsSuccessfully

# 4. Empaquetar omitiendo pruebas
.mvnw.cmd package -DskipTests
```

Universidad ICESI

Cheatsheet de JUnit 5 & Mockito

ASERCIONES DE VALOR

- `assertEquals(exp, act)`
- `assertTrue(cond)`
- `assertFalse(cond)`
- `assertNotNull(obj)`

ASERCIONES DE ERROR

- `assertThrows(Type.class, () -> op)`
- `assertAll(exec1, exec2)`
- Valida excepciones y agrupa comprobaciones en un solo reporte.

PROGRAMACIÓN DE MOCKS

- `when(m.fn()).thenReturn(val)`
- `when(m.fn()).thenThrow(ex)`
- `any(Class.class)`
- Configura respuestas del cascarón simulado.

VERIFICACIÓN DE LLAMADAS

- `verify(m, times(n)).fn()`
- `verify(m, never()).fn()`
- `verifyNoInteractions(m)`
- Audita el contrato de llamadas al mock.