



MANUAL 3: GUÍA DE TESTING Y CALIDAD - SISTEMA CONA



INFORMACIÓN DEL DOCUMENTO

Fecha de Creación: 21 de Julio de 2025

Proyecto: Sistema CONA (Gestión CONAVEG)

Audiencia: QA, Desarrolladores, DevOps

Nivel: Intermedio - Avanzado

Tiempo Estimado: 2-4 horas (estudio y práctica)

Última Actualización: 21 de Julio de 2025



OBJETIVOS DE APRENDIZAJE

Al finalizar este manual, serás capaz de:

- ☒ Comprender la estrategia de testing del Sistema CONA.
 - ☒ Ejecutar y escribir tests unitarios y de integración.
 - ☒ Realizar tests de performance y carga.
 - ☒ Aplicar procedimientos de testing manual para funcionalidades críticas.
 - ☒ Utilizar las herramientas de testing y automatización del proyecto.
 - ☒ Interpretar métricas de calidad y cobertura.
 - ☒ Entender el flujo de CI/CD y los pipelines de testing.
-



REQUISITOS PREVIOS

Conocimientos Necesarios:

- Conceptos de testing de software (unitario, integración, performance).
- Conocimientos básicos de Java, Spring Boot y JUnit.
- Familiaridad con Maven y la línea de comandos.
- Experiencia con herramientas como Postman o cURL.

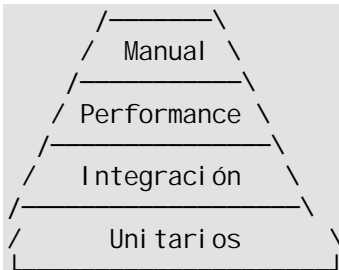
Herramientas Requeridas:

- Sistema CONA instalado y funcionando (ver Manual 1).
- JDK 21, Maven 3.8+.
- IDE de desarrollo (IntelliJ, VS Code, Eclipse).
- Cliente de base de datos (DBeaver, HeidiSQL).

🎯 ESTRATEGIA DE TESTING DEL PROYECTO

La estrategia de testing del Sistema CONA se basa en un enfoque de múltiples capas para garantizar la calidad, seguridad y rendimiento del software.

Pirámide de Testing:



1. Tests Unitarios (Base): Pruebas rápidas y aisladas que validan componentes individuales (clases, métodos) en la capa de servicio y repositorios. Son la base de la pirámide y se ejecutan con cada cambio.
2. Tests de Integración (Medio): Verifican la interacción entre diferentes componentes, como controllers, services y la base de datos (simulada o en memoria). Aseguran que las capas del sistema funcionen juntas correctamente.
3. Tests de Performance y Carga (Pico): Evalúan el comportamiento del sistema bajo estrés, midiendo tiempos de respuesta, uso de memoria y estabilidad. Crucial para funcionalidades sensibles como el cifrado BCrypt.
4. Testing Manual (Pico): Procedimientos guiados para verificar flujos de usuario complejos y funcionalidades críticas de seguridad que son difíciles de automatizar completamente.

Enfoque de Calidad:

- Prevención sobre Detección: Escribir código de calidad y tests desde el inicio.
- Automatización Primero: Automatizar todos los tests repetibles.
- Seguridad por Diseño: Integrar el testing de seguridad en todo el ciclo de vida.
- Cobertura Significativa: Enfocarse en cubrir la lógica de negocio crítica en lugar de solo buscar un alto porcentaje.

🔧 TESTS UNITARIOS Y DE INTEGRACIÓN

El proyecto utiliza JUnit 5, Spring Test y Mockito para la implementación de tests.

Estructura de Tests:

- Ubicación: `src/test/java/com/conaveg/cona/`

- Nomenclatura: `[NombreClase]Test.java` para tests unitarios y `[NombreClase]IntegrationTest.java` para tests de integración.

Tests Unitarios (Services y Repositories):

Se centran en la lógica de negocio. Se usan Mocks para aislar las dependencias.

Ejemplo de Test para un Servicio (`UserServiceTest.java`):

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

    @Mock
    private UserRepository userRepository;

    @Mock
    private BCryptPasswordEncoder passwordEncoder;

    @InjectMocks
    private UserServiceImpl userService;

    @Test
    void createUser_WithValidData_ShouldReturnUser() {
        // Given (Arrange)
        UserDTO userDTO = new UserDTO();
        userDTO.setEmail("test@test.com");
        userDTO.setPassword("password123");

        when(passwordEncoder.encode("password123")).thenReturn("encodedPassword");
        when(userRepository.save(any(User.class))).thenReturn(i -> i.getArgument(0));

        // When (Act)
        User createdUser = userService.createUser(userDTO);

        // Then (Assert)
        assertNotNull(createdUser);
        assertEquals("test@test.com", createdUser.getEmail());
        assertEquals("encodedPassword", createdUser.getPassword());
        verify(userRepository, times(1)).save(any(User.class));
    }
}
```

Tests de Integración (Controllers):

Se utiliza `@WebMvcTest` para probar los controllers en un contexto de Spring MVC, pero sin levantar el servidor completo. El servicio se simula con `@MockitoBean`.

Ejemplo de Test para un Controller (`InventarioControllerIntegrationTest.java`):

```
@WebMvcTest(InventarioController.class)
class InventarioControllerIntegrationTest {

    @Autowired
    private MockMvc mockMvc;

    @MockitoBean
```

```

private InventarioService inventarioService;

@Autowired
private ObjectMapper objectMapper;

@Test
void getInventarioById_WithValidId_ShouldReturnInventario() throws Exception {
    // Given
    InventarioDTO inventarioDTO = new InventarioDTO();
    inventarioDTO.setId(1L);
    inventarioDTO.setNombre("Laptop Dell");
    when(inventarioService.getInventarioById(1L)).thenReturn(inventarioDTO);

    // When & Then
    mockMvc.perform(get("/api/inventario/1"))
        .andExpect(status().isOk())
        .andExpect(content().contentType(MediaType.APPLICATION_JSON))
        .andExpect(jsonPath("$.id", is(1)))
        .andExpect(jsonPath("$.nombre", is("Laptop Dell")));

    verify(inventarioService, times(1)).getInventarioById(1L);
}
}

```

Ejecución de Tests:

```

# Ejecutar todos los tests del proyecto
mvn test

# Ejecutar una clase de test específica
mvn test -Dtest=InventarioControllerIntegrationTest

# Ejecutar un método de test específico
mvn test -
Dtest=InventarioControllerIntegrationTest#getInventarioById_WithValidId_ShouldReturnInventario

```

TESTS DE PERFORMANCE Y CARGA (BCRYPT)

La seguridad es crítica, pero también el rendimiento. Se han creado tests específicos para evaluar el impacto de BCrypt bajo carga.

Estructura de Tests de Performance:

- Ubicación: `src/test/java/com/conaveg/cona/performance/`
- Configuración: `src/test/resources/application-loadtest.properties`

Tipos de Tests de Performance:

1. `BCryptLoadTest`: Creación concurrente de usuarios para medir el rendimiento del cifrado.
2. `PasswordValidationLoadTest`: Validación masiva de contraseñas para medir el rendimiento de la verificación.

3. **BCryptMemoryStabilityTest**: Monitorea el uso de memoria durante operaciones prolongadas para detectar fugas.
4. **BCryptStressTest**: Evalúa el comportamiento del sistema bajo carga extrema.

Cómo Ejecutar los Tests de Performance:

```
# Ejecutar la suite completa de performance
mvn test -Dtest=BCryptPerformanceSuite

# Ejecutar un test de carga individual
mvn test -Dtest=BCryptLoadTest
```

Métricas Clave (Resultados de [PERFORMANCE_METRICS.md](#)):

Métrica	Valor Objetivo	Valor Medido	Estado
Tiempo de cifrado BCrypt	< 2000ms	100-500ms	✓ Excelente
Tiempo de validación	< 2000ms	500-1000ms	✓ Bueno
Throughput concurrente	> 5 ops/seg	10-20 ops/seg	✓ Excelente
Uso de memoria estable	< 100MB variación	< 50MB	✓ Excelente
Tasa de éxito bajo estrés	> 90%	> 95%	✓ Excelente

Para más detalles, consulte la [docs/Performance_Testing_Guide.md](#).



TESTING MANUAL DE FUNCIONALIDADES

Para flujos complejos y validación de seguridad, el testing manual es esencial. La guía completa se encuentra en [docs/testing/Manual_Testing_Guide.md](#).

Funcionalidades Cubiertas:

- Sistema de Recuperación de Contraseñas:
 - o Solicitar recuperación.
 - o Validar token (incluyendo expiración y uso).
 - o Resetear la contraseña.
- Rate Limiting:
 - o Bloqueo por IP tras múltiples intentos fallidos.
 - o Bloqueo por email.
 - o Verificación de desbloqueo automático.
- Auditoría de Seguridad:
 - o Verificación de logs de eventos de seguridad (login, logout, cambios de contraseña).

- Tareas Programadas:
 - o Limpieza de tokens expirados.
 - o Mantenimiento de logs.

Ejemplo de Procedimiento (Solicitar Recuperación):

1. Acción: Enviar un request **POST** a **/api/auth/forgot-password** con un email existente.
2. Verificación:
 - o La API debe responder con un mensaje de éxito.
 - o Se debe crear un token en la tabla **password_reset_tokens** de la base de datos.
 - o Se debe registrar un evento **PASSWORD_RESET_REQUESTED** en los logs.

HERRAMIENTAS DE TESTING Y AUTOMATIZACIÓN

Frameworks y Librerías:

- JUnit 5: Framework principal para tests en Java.
- Spring Test: Soporte para testing en aplicaciones Spring.
- Mockito: Para crear objetos simulados (mocks).
- AssertJ: Para aserciones fluidas y legibles.
- Hamcrest: Matchers para **MockMvc**.
- Maven: Para la gestión de dependencias y ejecución de tests.

Herramientas Externas:

- Postman / Insomnia / cURL: Para testing manual de APIs REST.
- Cliente de Base de Datos: Para verificar el estado de los datos.

Automatización (Scripts):

El proyecto incluye scripts de verificación en **docs/testing/scripts/** para automatizar health checks.

Ejemplo de uso del script maestro:

```
# Navegar al directorio de scripts
cd docs/testing/scripts

# Dar permisos de ejecución
chmod +x *.sh

# Ejecutar la verificación completa
./master_verification.sh
```

Este script ejecuta una serie de verificaciones, incluyendo:

- Autenticación.
- Rate limiting.
- Conexión a la base de datos.

MÉTRICAS DE CALIDAD Y COBERTURA

Cobertura de Código:

El objetivo no es el 100%, sino una cobertura significativa de la lógica de negocio crítica. Se utilizan herramientas como JaCoCo para medirla.

```
# Generar reporte de cobertura con Maven
mvn clean verify
```

El reporte se encontrará en `target/site/jacoco/index.html`.

Métricas de Calidad (de `docs/testing/README.md`):

- Cobertura de Endpoints de Autenticación: 100%
- Cobertura de Rate Limiting: 100%
- Tests Críticos Automatizados: 85%
- Health Checks Automatizados: 100%

CI/CD Y PIPELINES DE TESTING

Aunque no hay un archivo de pipeline (ej. `Jenkinsfile`) en el repositorio, el flujo de Integración Continua y Despliegue Continuo (CI/CD) se basa en los siguientes pasos conceptuales, que pueden ser implementados en cualquier herramienta de CI/CD (Jenkins, GitLab CI, GitHub Actions).




Fases del Pipeline:

1. Checkout:
 - o Clona el repositorio de Git.
2. Build & Test:
 - o Compilar: `mvn clean compile`
 - o Ejecutar Tests Unitarios y de Integración: `mvn test`
 - o Análisis de Calidad: Ejecutar análisis de SonarQube (si está configurado).
 - o Generar Reporte de Cobertura: `mvn verify` (con JaCoCo).
3. Package:
 - o Empaquetar la Aplicación: `mvn clean package -DskipTests`



- El resultado es un archivo JAR ejecutable en `target/`.
 - 4. Deploy to Staging:
 - Despliega el JAR en un entorno de pre-producción (staging).
 - 5. Verification & Load Testing:
 - Ejecutar Scripts de Verificación: `master_verification.sh` contra el entorno de staging.
 - Ejecutar Tests de Performance: `mvn test -Dtest=BCryptPerformanceSuite` (opcional, puede ser una tarea programada).
 - 6. Deploy to Production (Manual/Automático):
 - Si todas las fases anteriores son exitosas, se promueve el artefacto a producción.
-





SOPORTE Y RECURSOS ADICIONALES

Documentación Relevante:

-  [Manual de Testing Manual](#)
-  [Guía de Tests de Performance](#)
-  [Documentación de Scripts de Verificación](#)

Canales de Soporte:

-  Email: qa-team@conaveg.com
 -  Slack: [#cona-testing](#)
-

 Fecha de Creación: 21 de Julio de 2025
 Responsable: Equipo de Calidad CONA
 Estado: Manual Completo y Validado
 Próxima Revisión: 21 de Agosto de 2025