



UNIVERSIDAD
NACIONAL
DE COLOMBIA

Universidad Nacional de Colombia - sede Bogotá
Facultad de Ingeniería
Departamento de Ingeniería de Sistemas e Industrial
Curso: Ingeniería de Software 1

Integrantes:

- Santiago Alejandro Rojas Feo
 - Juan Diego Rozo Álvarez
 - Felipe Rojas Marín
 - Miguel Ángel Citarella Camargo
-



Módulo de Testing - Proyecto SpyNet

Este documento describe las pruebas unitarias implementadas por nosotros, los miembros del equipo para garantizar el correcto funcionamiento de funcionalidades importantes en el sistema SpyNet. Cada integrante ha contribuido con al menos 3 pruebas unitarias significativas como parte del desarrollo del módulo de testing.



Herramientas utilizadas

Dado que en este proyecto estamos haciendo uso de Java, para introducir los test incluiremos el marco de pruebas unitarias JUnit 5.

Este módulo se incluye tanto en el pom.xml como en nuestra clase de module-info.java.

Los test podrán ser hallados en la siguiente ruta del proyecto:

Proyecto\SpyNet\src\test\java\org\catatunbo\spynet\tests

Y estarán separados en dos clases distintas, las cuales son; CreateUserTests.java y SpyNetCoreTest.java. Esta decisión fue tomada arbitrariamente para separar las funcionalidades que son propias de la aplicación, como las que son parte aislada que no son fundamentalmente incluyentes en la lógica del producto.

Integrante 1: Juan Diego Rozo Álvarez

1. Generación de PDF

Clase: **ReporteAuditoriaPDF**

Test: **testPDFGenerationCreatesFile**

Creamos un archivo PDF de auditoría utilizando datos ficticios. Esta prueba verifica que el archivo se genere correctamente en el sistema de archivos, que tenga contenido y que cumpla su propósito de almacenar los resultados de auditoría en un formato estructurado y legible para su posterior exportación.

Java

```
assertTrue(file.exists() && file.length() > 0);
```

2. Ejecución de comando Nmap

Clase: **nmapCommand**

Test: **testNmapCommandReturnsOutput**

Ejecutamos un escaneo de red mediante Nmap sobre la IP **127.0.0.1** utilizando el parámetro **-sn** (ping scan). Evaluamos si la salida del proceso contiene información relevante para determinar si el sistema reconoce la herramienta externa y su ejecución correcta.

Java

```
assertTrue(output.toLowerCase().contains("nmap"));
```

3. Verificación de conexión a base de datos

Test: **testDatabaseConnection**

Abrimos una conexión a la base de datos MySQL y comprobamos que no sea nula ni esté cerrada. Esto garantiza que la configuración de conexión de la aplicación esté funcionando correctamente.

Java

```
assertNotNull(conn);  
assertFalse(conn.isClosed());
```

Integrante 2: Santiago Alejandro Rojas Feo

1. Patrón Singleton y validación de roles

Clase: **Session**

Test: **testSessionSingletonAndRoleValidation**

Validamos que la clase **Session** esté diseñada siguiendo el patrón Singleton, es decir, que todas las llamadas a `getInstance()` retornen la misma instancia. Además, evaluamos si el sistema reconoce correctamente el rol del usuario en sesión y permite acceso según privilegios.

Java

```
assertSame(session1, session2);  
assertTrue(session1.hasRole("Admin"));
```

2. Validación de estados de usuario

Clase: **User**

Test: **testUserStateValidationAndBusinessLogic**

Creamos dos instancias de usuario con estados distintos (activo e inactivo) y roles diferentes. Confirmamos que el método `toString()` refleje esa información, y que el sistema maneje correctamente la lógica de negocio entre usuarios con distintos permisos o fechas de última sesión.

Java

```
assertTrue(activeToString.contains("ACTIVE"));
```

3. Creación de nuevo usuario sin excepción

Clase: **CreateUserController**

Test: **testAddNewUserNotThrowException**

Creamos un nuevo usuario desde el controlador **CreateUserController** y lo insertamos en la base de datos. Este test garantiza que el proceso no lanza excepciones inesperadas, asegurando la estabilidad del flujo de registro.

Java

```
assertDoesNotThrow(() -> userDao.addToDataBase(newUser));
```

Integrante 3: Felipe Rojas Marín

1. Consulta de auditorías

Clase: **AuditoryDAO**

Test: **getAllAuditories**

Llamamos al método `getAllAuditories()` que consulta la base de datos y retorna todas las auditorías registradas. Verificamos que la lista retornada no sea nula ni vacía, lo cual confirma que la persistencia y recuperación de datos funciona adecuadamente.

Java

```
assertFalse(auditories.isEmpty());
```

2. Inserción de hallazgos

Clase: **AuditoryDAO**

Test: **insertFinding**

Probamos la funcionalidad de registrar un hallazgo en una auditoría específica. Insertamos manualmente los datos y comprobamos que el método devuelva `true`, lo cual asegura que el sistema está permitiendo almacenar hallazgos correctamente.

Java

```
assertTrue(result);
```

3. Actualización del auditor asignado !

Clase: **AuditoryDAO**

Test: **updateAuditoryAssignedUser**

Este test fue propuesto por IA para cubrir la lógica de modificación de asignación de auditores. Simula el cambio del usuario encargado de una auditoría específica, verificando que dicho cambio se realice correctamente.

Java

```
assertTrue(result);
```

Integrante 4: Miguel Ángel Citarella Camargo

1. Seguridad: Hash de contraseñas

Clase: PasswordHasher

Test: testPasswordHashingConsistencyAndSecurity

Evaluamos la robustez y consistencia del algoritmo de hash de contraseñas. Verificamos que con el mismo `salt` se obtenga siempre el mismo resultado, y que al cambiar el `salt`, el hash resultante sea distinto. Así garantizamos que el algoritmo evita colisiones y protege las credenciales.

Java

```
assertEquals(hash1, hash2);
assertNotEquals(hash1, differentHash);
```

2. Creación de nueva auditoría

Clase: AuditoryDAO

Test: createAuditory

Simulamos la creación de una nueva auditoría en la base de datos para un cliente específico. Verificamos que el método retorna un identificador mayor a cero, lo que confirma que la inserción fue exitosa y que la auditoría está correctamente registrada.

Java

```
assertTrue(newAuditoryId > 0);
```

3. Validación de API Key de OpenAI

Clase: OpenAIConfig

Test: testOpenAIApiKeyIsPresent

Comprobamos que la API Key de OpenAI esté debidamente configurada y accesible desde el archivo `openai.secret.properties`. Esto permite garantizar que los módulos que dependen de la IA funcionarán correctamente al realizar solicitudes.

Java

```
assertNotNull(OpenAIConfig.getApiKey());
```



Conclusiones del módulo de pruebas



- Hemos implementado 3 pruebas unitarias por cada uno de los 4 integrantes, cubriendo las áreas más importantes de nuestra aplicación.
 - Se verificaron componentes fundamentales como la conexión a la base de datos, generación de archivos, validación de roles, y APIs externas. Todo esto hace parte de las funcionalidades añadidas a forma de módulos para SpyNet.
 - Las pruebas fueron ejecutadas exitosamente mediante `mvn test`, sobre el PATH en el que se encuentra el POM, asegurando calidad antes del despliegue final.
 - Nota: los test que verifican la conexión a la base de datos, junto a la identificación de la API_KEY dentro del proyecto, son altamente propensos a fallar dada la alta sensibilidad de la configuración del entorno, junto las versiones usadas de Java. (En este caso, se usa Java 21, u OpenJDK 21)
-

Análisis estático del código con Checkstyle

Como parte del módulo de pruebas, también realizamos un análisis estático del código fuente utilizando la herramienta Checkstyle, con el conjunto de reglas `google_checks.xml`. Este paso fue clave para asegurar que el proyecto SpyNet no solo funcione correctamente, sino que también mantenga un estilo de codificación claro, consistente y mantenible.



Resumen del análisis

- Total de archivos analizados: 27
- Errores críticos encontrados: 0 
- Advertencias (warnings): 2094 
- Severidad de los problemas: Todos los hallazgos fueron advertencias de estilo, sin errores funcionales graves.



Principales categorías de advertencias detectadas

- Indentación incorrecta (Indentation)
Más de 1600 advertencias están relacionadas con niveles de sangría no consistentes, lo cual afecta la legibilidad del código.
- Orden y estilo de importaciones (CustomImportOrder, AvoidStarImport)
Se detectaron 115 advertencias relacionadas con importaciones mal organizadas o agrupadas.

- Faltan comentarios Javadoc (MissingJavadocType, MissingJavadocMethod)
Más de 60 advertencias indican que falta documentación en clases o métodos públicos.
- Longitud de línea excesiva (LineLength)
Se reportaron 83 líneas que superan el límite recomendado de 100 caracteres.
- Nombres no convencionales (AbbreviationAsWordInName, LocalVariableName)
Algunos nombres de clases o variables no siguen la convención camelCase.
- Espaciado y separación visual (WhitespaceAround, EmptyLineSeparator)
Más de 100 advertencias por espacios faltantes o líneas vacías incorrectas.

Conclusiones del análisis

- No se encontraron errores críticos en el código.
- La mayoría de advertencias están relacionadas con estética, organización y documentación.
- Corregir estos puntos mejorará la legibilidad, mantenibilidad y profesionalismo del proyecto.
- Checkstyle nos ayudó a mantener coherencia entre estilos de codificación entre todos los integrantes.

En la carpeta donde se encuentra este documento, adjuntamos el reporte directamente obtenido de Checkstyle. Este archivo se llamará checkstyle_informe.pdf