UNIVERSIDAD DE LAS FUERZAS ARMADAS ESPE

Departamento de Ciencias de la Computación

Pruebas Unitarias

Pruebas del Sistema

Sangolquí, Ecuador

Índice

Índice de figuras

Índice de cuadros

1 Introducción

El presente informe documenta el proceso de diseño, ejecución y análisis de las pruebas unitarias aplicadas al sistema **El Granito**, una plataforma financiera orientada a la gestión de créditos, pagos electrónicos y generación de reportes.

Las pruebas unitarias se basan en los flujos de actividades definidos para cada rol del sistema y tienen como objetivo validar el correcto funcionamiento de los componentes individuales antes de su integración con otros módulos.

Adicionalmente, como parte de la estrategia de aseguramiento de la calidad del software, se considera la futura ejecución de pruebas de carga, rendimiento y validación de servicios mediante herramientas especializadas como **Apache JMeter**, **Postman** y **k6**, las cuales complementan el proceso de pruebas unitarias.

2 Objetivo de las Pruebas Unitarias

El objetivo principal de las pruebas unitarias es:

- Verificar que cada función o método del sistema cumpla con los requisitos definidos.
- Detectar errores de forma temprana en la lógica del negocio.
- Asegurar la correcta validación de datos de entrada y salida.
- Reducir el costo de corrección de defectos en fases posteriores.
- Facilitar el mantenimiento y evolución del sistema.

3 Alcance

Las pruebas unitarias cubren los siguientes módulos del sistema:

- Autenticación y recuperación de contraseña
- Gestión de usuarios
- Gestión de créditos
- Procesamiento de pagos electrónicos
- Generación de certificados y reportes financieros

No se incluyen en este informe pruebas de integración, pruebas de rendimiento ni pruebas de aceptación, las cuales se documentarán en fases posteriores del proyecto.

4 Herramientas de Pruebas Utilizadas

4.1 Frameworks de Pruebas Unitarias

Las pruebas unitarias se ejecutaron de forma automatizada utilizando frameworks compatibles con la arquitectura del backend, permitiendo validar métodos y servicios de manera aislada.

4.2 Herramientas Complementarias

- **Apache JMeter:** Se utilizará para pruebas de carga y rendimiento de los servicios REST.
- **k6:** Herramienta orientada a pruebas de rendimiento automatizadas mediante scripts.
- **Postman:** Herramienta para pruebas manuales y automatizadas de APIs REST. Actualmente se encuentra en fase de planificación y aún no ha sido implementada.

5 Estrategia de Pruebas Unitarias

Cada prueba unitaria sigue el principio **AAA**:

- **Arrange:** Preparación del contexto y datos de prueba.
- **Act:** Ejecución del método o función bajo prueba.
- **Assert:** Comparación del resultado obtenido con el esperado.

Las dependencias externas, como bases de datos o servicios externos, son simuladas mediante *mocks* para garantizar el aislamiento de cada unidad.

6 Paso a Paso para la Implementación de Pruebas Unitarias

6.1 Paso 1: Identificación de Unidades

Se identifican las funciones críticas del sistema a partir de los diagramas de actividades y reglas de negocio:

- Validación de credenciales de usuario

- Cálculo de saldo pendiente de un crédito
- Registro y validación de pagos

6.2 Paso 2: Preparación del Entorno

- Configuración del framework de pruebas.
- Creación de una carpeta específica para pruebas (`/tests`).
- Configuración de datos simulados.
- Aislamiento de dependencias externas.

6.3 Paso 3: Diseño del Caso de Prueba

Cada caso de prueba debe definir claramente:

- Datos de entrada
- Resultado
- Condición de validación
- Resultado real

Ejemplo: Validación de contraseña

- Entrada: contraseña con menos de 8 caracteres
- Resultado: rechazo del sistema

6.4 Paso 4: Ejecución de las Pruebas

Las pruebas se ejecutan automáticamente mediante comandos del framework. Los resultados se almacenan en reportes que detallan pruebas exitosas, fallidas y omitidas.

6.5 Paso 5: Análisis de Resultados

Las pruebas fallidas permiten identificar:

- Errores lógicos
- Validaciones incompletas
- Falta de cobertura en reglas del negocio

Los defectos detectados se documentan en el sistema de gestión de incidencias.

7 Pruebas Automatizadas con Jest y Pruebas de Carga con k6

En esta sección se presentan ejemplos extendidos de pruebas automatizadas tanto a nivel de lógica individual del backend (Jest) como de pruebas de carga y rendimiento sobre los endpoints REST del sistema, utilizando la herramienta k6. Se asume que el backend está desarrollado con Node.js y Express, con rutas que exponen servicios de autenticación, gestión de usuarios y procesamiento de pagos.

7.1 Pruebas con Jest

Las pruebas con Jest permiten comprobar la lógica del código en forma aislada o en integración con funciones específicas. A continuación se ejemplifican escenarios de prueba que cubren varios endpoints del backend.

7.1.1 Prueba de autenticación de usuario

```
import request from "supertest";
import app from "../src/app.js";

describe("POST /api/auth/login", () => {
  it("debe autenticar al usuario con credenciales correctas",
   async () => {
    const response = await request(app)
      .post("/api/auth/login")
      .send({
        email: "usuario@ejemplo.com",
        password: "contrasenaSegura",
      });
    expect(response.statusCode).toBe(200);
  });

  it("debe rechazar con credenciales incorrectas", async () => {
    const response = await request(app)
      .post("/api/auth/login")
      .send({
        email: "usuario@ejemplo.com",
        password: "contrasenaIncorrecta",
      });
```

```
    expect(response.statusCode).toBe(401);
  });
});
```

Listing 1: Jest - Prueba de autenticación de usuario

7.1.2 Prueba de creación de usuario

```
describe("POST /api/usuarios", () => {
  it("debe crear un nuevo usuario", async () => {
    const response = await request(app)
      .post("/api/usuarios")
      .send({
        nombre: "Juan",
        email: "juan@dominio.com",
        password: "contrasena123",
      });
    expect(response.statusCode).toBe(201);
  });
});
```

Listing 2: Jest - Prueba de creación de usuario

7.1.3 Prueba de listado de pagos

```
describe("GET /api/pagos", () => {
  it("debe retornar la lista de pagos", async () => {
    const response = await request(app).get("/api/pagos");
    expect(response.statusCode).toBe(200);
    expect(Array.isArray(response.body)).toBe(true);
  });
});
```

Listing 3: Jest - Prueba de listado de pagos

7.1.4 Prueba de registro de pago

```
describe("POST /api/pagos", () => {
  it("debe registrar un nuevo pago con datos validos", async ()
   => {
```

```
    const response = await request(app)
      .post("/api/pagos")
      .send({
        referencia: "REF001",
        monto: 150.50,
        usuarioId: "12345",
      });
    expect(response.statusCode).toBe(201);
  });
});
```

Listing 4: Jest - Prueba de registro de nuevo pago

7.2 Pruebas de Carga y Rendimiento con k6

Las pruebas de carga con k6 permiten evaluar el comportamiento del sistema cuando múltiples usuarios simultáneos acceden a los endpoints durante un periodo definido de tiempo. Los siguientes ejemplos cubren pruebas de carga para los servicios principales.

7.2.1 Prueba de carga para endpoint de autenticación

```
import http from "k6/http";
import { check } from "k6";

export let options = {
  vus: 30,
  duration: "30s",
};

export default function () {
  const payload = JSON.stringify({
    email: "usuario@ejemplo.com",
    password: "contrasenaSegura",
  });

  const headers = { "Content-Type": "application/json" };

  const res = http.post("http://localhost:3000/api/auth/login",
   payload, { headers });

```

```
  check(res, {
    "status 200": (r) => r.status === 200,
  });
}
```

Listing 5: k6 - Prueba de carga para autenticación

7.2.2 Prueba de carga para listado de pagos

```
import http from "k6/http";
import { check } from "k6";

export let options = {
  vus: 50,
  duration: "45s",
};

export default function () {
  const res = http.get("http://localhost:3000/api/pagos");

  check(res, {
    "estado es 200": (r) => r.status === 200,
    "tiene datos": (r) => r.body.length > 0,
  });
}
```

Listing 6: k6 - Prueba de carga para listado de pagos

7.2.3 Prueba de estrés en registro de pagos

```
import http from "k6/http";
import { check } from "k6";

export let options = {
  vus: 40,
  duration: "1m",
};

export default function () {
  const payload = JSON.stringify({
```

```
    referencia: "REFK6",
    monto: Math.random() * 200,
    usuarioId: "abc123",
  });

  const headers = { "Content-Type": "application/json" };

  const res = http.post("http://localhost:3000/api/pagos",
   payload, { headers });

  check(res, {
    "estado 201": (r) => r.status === 201,
  });
}
```

Listing 7: k6 - Prueba de estres en registro de pagos

En cada uno de los scripts de k6, se especifican:

- La cantidad de usuarios virtuales (`vus`) que simulan accesos concurrentes.
- La duración de la prueba (por ejemplo, `"30s"` para 30 segundos).
- La verificación de los códigos de estado HTTP esperados.

Estas pruebas permiten observar el comportamiento del sistema frente a solicitudes simultáneas, aportando información valiosa para mejorar el rendimiento y escalabilidad del backend.

8 Métricas de Pruebas Unitarias

- Casos unitarios planificados: 150
- Casos ejecutados: 145
- Casos exitosos: 140
- Casos fallidos: 5
- Cobertura de código: 82 %

Estado de Pruebas con Postman

La herramienta Postman ha sido considerada para la validación manual y automatizada de los servicios REST del sistema. Sin embargo, en la versión actual del proyecto, las pruebas con Postman aún no han sido implementadas.

Se prevé su uso en futuras iteraciones para:

- Validación manual de endpoints
- Pruebas de autenticación con tokens
- Automatización de colecciones de pruebas

9 Conclusiones

- La implementación de pruebas automatizadas con Jest permitió validar de manera sistemática la lógica interna del backend, identificando errores en los flujos de autenticación, gestión de usuarios y procesamiento de pagos antes de su despliegue, lo que contribuyó a mejorar la calidad del software en etapas tempranas del desarrollo.
- Las pruebas de carga realizadas con k6 evidenciaron el comportamiento del sistema bajo múltiples solicitudes concurrentes, permitiendo analizar la estabilidad, tiempos de respuesta y capacidad de atención del servicio, aspectos fundamentales para una plataforma orientada a transacciones financieras.
- La integración de herramientas de pruebas en el proceso de desarrollo fortalece las buenas prácticas de ingeniería de software, ya que promueve la detección temprana de fallos, facilita el mantenimiento del sistema y proporciona mayor confianza en la evolución futura del proyecto.

10 Recomendaciones

- Se recomienda ampliar la cobertura de las pruebas unitarias incorporando escenarios negativos y casos límite, especialmente en los módulos críticos relacionados con seguridad, validación de datos y procesamiento de pagos.
- Integrar formalmente Postman en futuras iteraciones del proyecto permitirá complementar las pruebas automatizadas con pruebas manuales y de regresión, facilitando la validación de endpoints y la documentación de los servicios API.

- Automatizar la ejecución de las pruebas mediante pipelines de integración continua (CI) permitirá asegurar que cada cambio en el código sea validado de forma automática, reduciendo el riesgo de introducir errores en versiones posteriores del sistema.