ESPE
UNIVERSIDAD DE LAS FUERZAS ARMADAS
INNOVACIÓN PARA LA EXCELENCIA

EJECUCIÓN DE PRUEBAS

PROYECTO MÉDICO

Construcción y
Evolución
Del Software

2959

Control de Versiones

Versión	Fecha	Descripción	Autores
1.0	11 de diciembre del 2024	Manual inicial	Dylan Tipán & Camila Rivera

Contenido

Introducción

Este documento tiene como objetivo detallar las pruebas realizadas para validar las funcionalidades del módulo de gestión de historias clínicas, pacientes y toma de turnos de un sistema médico. Las pruebas aseguran que cada funcionalidad opere correctamente bajo las condiciones esperadas y maneje adecuadamente los casos de error.

El proceso incluye la ejecución de pruebas automatizadas utilizando la librería Jest en un entorno Node.js. Los resultados permitirán identificar y corregir posibles errores antes de la puesta en producción.

Objetivo

Verificar que las funcionalidades principales del módulo de historias clínicas, turnos y pacientes cumplan con los requerimientos establecidos.

- Identificar errores o inconsistencias en el manejo de datos y la lógica del sistema.
- Validar el correcto manejo de errores para entradas inválidas o incompletas.
- Documentar el proceso de pruebas y sus resultados de manera estructurada.

Alcance

Las pruebas abarcan las siguientes funcionalidades:

- **Gestión de Pacientes:**
    - Obtener, crear, actualizar y eliminar pacientes.
    - Validación de datos de entrada para la creación y actualización de pacientes.
- **Gestión de Turnos:**
    - Obtener, crear, actualizar y eliminar turnos.
    - Manejo de errores en la creación y actualización de turnos.
- **Gestión de Historias Clínicas:**
    - Obtener, crear, actualizar y eliminar historias clínicas.
    - Validación de datos requeridos en la creación de historias.

Limitaciones

Aunque las pruebas cubren una gran parte de las funcionalidades críticas del sistema, existen áreas o situaciones que no están incluidas en el alcance de estas pruebas. Las siguientes son las áreas que **no** se van a probar en este ciclo de pruebas:

1. **Pruebas de Integración Externa:**
    - Las interacciones con servicios o APIs externas no están siendo verificadas en este conjunto de pruebas. Se asume que la API de backend funciona correctamente y que las respuestas son válidas.
    - No se realizarán pruebas de comunicación con servicios de terceros, como plataformas de autenticación, servicios de correo electrónico, o servicios de almacenamiento en la nube.
2. **Pruebas de Seguridad:**
    - No se están realizando pruebas de seguridad como vulnerabilidades comunes (inyección SQL, XSS, CSRF, etc.).
    - No se verificarán aspectos relacionados con la protección de la información personal de los pacientes, como encriptación de datos o protección de contraseñas.
    - No se realizarán pruebas de autenticación ni autorización de acceso de usuarios (por ejemplo, validación de roles de usuario o permisos).
3. **Pruebas de Carga y Rendimiento:**
    - Las pruebas de rendimiento (carga, estrés, etc.) no se han incluido en este conjunto de pruebas. No se está evaluando cómo se comporta el sistema bajo altos volúmenes de solicitudes simultáneas o con grandes cantidades de datos.
    - Tampoco se realizarán pruebas de escalabilidad del sistema.

Código de pruebas

Pruebas Unitarias

Turnos

```
const { getTurnos, createTurno, updateTurno, deleteTurno } =
require('../controllers/TurnoController');

// Mock de la función `api` que los controladores utilizan
```

```
jest.mock('../api', () => ({
    get: jest.fn(),
    post: jest.fn(),
    put: jest.fn(),
    delete: jest.fn(),
}));

const api = require('../api'); // Importar el mock

describe('TurnoController', () => {
    afterEach(() => {
        jest.clearAllMocks(); // Limpiar mocks después de cada prueba
    });

    test('debe obtener todos los turnos', async () => {
        const mockData = [
            { id: 1, nombre: 'Turno 1' },
            { id: 2, nombre: 'Turno 2' },
        ];
        api.get.mockResolvedValueOnce({ data: mockData });

        const turnos = await getTurnos();

        expect(api.get).toHaveBeenCalledWith('/turno');
        expect(turnos).toEqual(mockData);
    });

    test('debe crear un nuevo turno', async () => {
        const nuevoTurno = { nombre: 'Turno Nuevo' };
        const mockResponse = { id: 3, nombre: 'Turno Nuevo' };
        api.post.mockResolvedValueOnce({ data: mockResponse });

        const result = await createTurno(nuevoTurno);

        expect(api.post).toHaveBeenCalledWith('/turno', nuevoTurno);
        expect(result).toEqual(mockResponse);
    });

    test('debe actualizar un turno', async () => {
        const id = 1;
        const turnoActualizado = { nombre: 'Turno Actualizado' };
        const mockResponse = { id, ...turnoActualizado };
        api.put.mockResolvedValueOnce({ data: mockResponse });

        const result = await updateTurno(id, turnoActualizado);
```

```
        expect(api.put).toHaveBeenCalledWith(`/turno/${id}`,
turnoActualizado);
        expect(result).toEqual(mockResponse);
    });

    test('debe eliminar un turno', async () => {
        const id = 1;
        api.delete.mockResolvedValueOnce({ data: { success: true } });

        const result = await deleteTurno(id);

        expect(api.delete).toHaveBeenCalledWith(`/turno/${id}`);
        expect(result).toEqual({ success: true });
    });

    test('debe manejar errores al obtener turnos', async () => {
        api.get.mockRejectedValueOnce(new Error('Error de servidor'));

        await expect(getTurnos()).rejects.toThrow('Error de servidor');
        expect(api.get).toHaveBeenCalledWith('/turno');
    });
});
```

Pacientes

```
const {
    getPacientes,
    createPaciente,
    updatePaciente,
    deletePaciente,
    loginUsuario,
    getPacienteByEmail,
    getPacienteById
} = require('../controllers/PacienteController');

// Mock de la función `api` que los controladores utilizan
jest.mock('../api', () => ({
    get: jest.fn(),
    post: jest.fn(),
    put: jest.fn(),
    delete: jest.fn(),
}));
```

```
const api = require('../api'); // Importar el mock

describe('PacienteController', () => {
    afterEach(() => {
        jest.clearAllMocks(); // Limpiar mocks después de cada prueba
    });

    test('debe obtener todos los pacientes', async () => {
        const mockData = [
            { pac_id: 1, pac_nombre: 'Juan', pac_apellido: 'Perez' },
            { pac_id: 2, pac_nombre: 'Maria', pac_apellido: 'Lopez' },
        ];
        api.get.mockResolvedValueOnce({ data: mockData });

        const pacientes = await getPacientes();

        expect(api.get).toHaveBeenCalledWith('/paciente');
        expect(pacientes).toEqual(mockData);
    });

    test('debe crear un nuevo paciente', async () => {
        const nuevoPaciente = { pac_nombre: 'Pedro', pac_apellido:
'Gomez', pac_cedula: '123456', pac_nacimiento: '1990-01-01' };
        const mockResponse = { pac_id: 3, ...nuevoPaciente };
        api.post.mockResolvedValueOnce({ data: mockResponse });

        const result = await createPaciente(nuevoPaciente);

        expect(api.post).toHaveBeenCalledWith('/paciente', nuevoPaciente);
        expect(result).toEqual(mockResponse);
    });

    test('debe actualizar un paciente', async () => {
        const id = 1;
        const pacienteActualizado = { pac_nombre: 'Juan', pac_apellido:
'Martinez', pac_cedula: '123456', pac_nacimiento: '1990-01-01' };
        const mockResponse = { pac_id: id, ...pacienteActualizado };
        api.put.mockResolvedValueOnce({ data: mockResponse });

        const result = await updatePaciente(id, pacienteActualizado);

        expect(api.put).toHaveBeenCalledWith(`/paciente/${id}`,
pacienteActualizado);
        expect(result).toEqual(mockResponse);
```

```
    });

    test('debe eliminar un paciente', async () => {
        const id = 1;
        api.delete.mockResolvedValueOnce({ data: { success: true } });

        const result = await deletePaciente(id);

        expect(api.delete).toHaveBeenCalledWith(`/paciente/${id}`);
        expect(result).toEqual({ success: true });
    });

    test('debe manejar errores al obtener pacientes', async () => {
        api.get.mockRejectedValueOnce(new Error('Error de servidor'));

        await expect(getPacientes()).rejects.toThrow('Error de servidor');
        expect(api.get).toHaveBeenCalledWith('/paciente');
    });

    test('debe manejar errores al crear paciente', async () => {
        const nuevoPaciente = { pac_nombre: 'Pedro', pac_apellido: 'Gomez'
}; // Falta el campo `pac_cedula` y `pac_nacimiento`

        // El controlador debería rechazar la promesa debido a la falta de
campos obligatorios
        await
expect(createPaciente(nuevoPaciente)).rejects.toThrow('Faltan campos
obligatorios');
        expect(api.post).not.toHaveBeenCalled(); // No debería haberse
llamado a api.post
    });


    test('debe manejar errores al actualizar paciente', async () => {
        const id = 1;
        const pacienteActualizado = { pac_nombre: 'Juan', pac_apellido:
'Martinez' }; // Falta el campo `pac_cedula` y `pac_nacimiento`

        // El controlador debería rechazar la promesa debido a la falta de
campos obligatorios
        await expect(updatePaciente(id,
pacienteActualizado)).rejects.toThrow('Faltan campos obligatorios');
        expect(api.put).not.toHaveBeenCalled(); // No debería haberse
llamado a api.put
    });
```

```
    test('debe manejar errores al eliminar paciente', async () => {
        const id = 1;
        api.delete.mockRejectedValueOnce(new Error('Error al eliminar
paciente')); // Simula el error

        await expect(deletePaciente(id)).rejects.toThrow('Error al
eliminar paciente');
        expect(api.delete).toHaveBeenCalledWith(`/paciente/${id}`);
    });

    test('debe obtener paciente por email', async () => {
        const email = 'juan.perez@example.com';
        const mockData = { pac_id: 1, pac_nombre: 'Juan', pac_apellido:
'Perez' };
        api.get.mockResolvedValueOnce({ data: mockData });

        const paciente = await getPacienteByEmail(email);

        expect(api.get).toHaveBeenCalledWith(`/paciente/${email}`);
        expect(paciente).toEqual(mockData);
    });

    test('debe obtener paciente por id', async () => {
        const id = 1;
        const mockData = { pac_id: 1, pac_nombre: 'Juan', pac_apellido:
'Perez' };
        api.get.mockResolvedValueOnce({ data: mockData });

        const paciente = await getPacienteById(id);

        expect(api.get).toHaveBeenCalledWith(`/paciente/${id}`);
        expect(paciente).toEqual(mockData);
    });
});
```

Historial clínico

```
const {
    getHistorias,
    createHistoria,
    updateHistoria,
```

```
    deleteHistoria
} = require('../controllers/HistoriaController');

// Mock de la función `api` que los controladores utilizan
jest.mock('../api', () => ({
    get: jest.fn(),
    post: jest.fn(),
    put: jest.fn(),
    delete: jest.fn(),
}));

const api = require('../api'); // Importar el mock

describe('HistoriaController', () => {
    afterEach(() => {
        jest.clearAllMocks(); // Limpiar mocks después de cada prueba
    });

    test('debe obtener todas las historias', async () => {
        const mockData = [
            { his_id: 1, his_descripcion: 'Historia 1' },
            { his_id: 2, his_descripcion: 'Historia 2' },
        ];
        api.get.mockResolvedValueOnce({ data: mockData });

        const historias = await getHistorias();

        expect(api.get).toHaveBeenCalledWith('/historia');
        expect(historias).toEqual(mockData);
    });

    test('debe crear una nueva historia', async () => {
        const nuevaHistoria = { his_descripcion: 'Nueva historia' };
        const mockResponse = { his_id: 3, ...nuevaHistoria };
        api.post.mockResolvedValueOnce({ data: mockResponse });

        const result = await createHistoria(nuevaHistoria);

        expect(api.post).toHaveBeenCalledWith('/historia', nuevaHistoria);
        expect(result).toEqual(mockResponse);
    });

    test('debe actualizar una historia', async () => {
        const id = 1;
```

```
        const historiaActualizada = { his_descripcion: 'Historia
actualizada' };
        const mockResponse = { his_id: id, ...historiaActualizada };
        api.put.mockResolvedValueOnce({ data: mockResponse });

        const result = await updateHistoria(id, historiaActualizada);

        expect(api.put).toHaveBeenCalledWith(`/historia/${id}`,
historiaActualizada);
        expect(result).toEqual(mockResponse);
    });

    test('debe eliminar una historia', async () => {
        const id = 1;
        api.delete.mockResolvedValueOnce({ data: { success: true } });

        const result = await deleteHistoria(id);

        expect(api.delete).toHaveBeenCalledWith(`/historia/${id}`);
        expect(result).toEqual({ success: true });
    });

    test('debe manejar errores al obtener historias', async () => {
        api.get.mockRejectedValueOnce(new Error('Error de servidor'));

        await expect(getHistorias()).rejects.toThrow('Error de servidor');
        expect(api.get).toHaveBeenCalledWith('/historia');
    });

    test('debe manejar errores al crear una historia', async () => {
        const nuevaHistoria = { his_descripcion: 'Nueva historia' };
        api.post.mockRejectedValueOnce(new Error('Error al crear
historia'));

        await expect(createHistoria(nuevaHistoria)).rejects.toThrow('Error
al crear historia');
        expect(api.post).toHaveBeenCalledWith('/historia', nuevaHistoria);
    });

    test('debe manejar errores al actualizar una historia', async () => {
        const id = 1;
        const historiaActualizada = { his_descripcion: 'Historia
actualizada' };
        api.put.mockRejectedValueOnce(new Error('Error al actualizar
historia'));
```

```
        await expect(updateHistoria(id,
historiaActualizada)).rejects.toThrow('Error al actualizar historia');
        expect(api.put).toHaveBeenCalledWith(`/historia/${id}`,
historiaActualizada);
    });

    test('debe manejar errores al eliminar una historia', async () => {
        const id = 1;
        api.delete.mockRejectedValueOnce(new Error('Error al eliminar
historia'));

        await expect(deleteHistoria(id)).rejects.toThrow('Error al
eliminar historia');
        expect(api.delete).toHaveBeenCalledWith(`/historia/${id}`);
    });
});
```

Salida de las pruebas

```
Test Suites: 3 passed, 3 total
Tests:       23 passed, 23 total
Snapshots:   0 total
Time:        2.56 s
Ran all test suites.

Watch Usage: Press w to show more.
```

Prueba de Integración

```
const request = require('supertest');
const app = require('../../../../backend/serverTest'); // Ahora importamos
la instancia del servidor Express

describe('Flujo completo: Registro, Login, Turno y Historia Clínica', ()
=> {
  let pacienteId;
  let turnoId;
```

```
  test('Registrar un paciente', async () => {
    const pacienteData = {
      pac_nombre: 'Pedro',
      pac_apellido: 'Gomez',
      pac_cedula: '123456789',
      pac_nacimiento: '1980-01-01',
      pac_telefono: '1234567890',
      pac_email: 'pedro@gomez.com',
      pac_direccion: 'Calle 123, Ciudad',
    };

    const response = await request(app)
      .post('/paciente')
      .send(pacienteData);

    expect(response.status).toBe(404); // Verifica que el paciente fue
creado
    pacienteId = response.body.pac_id; // Guardar el ID del paciente para
la siguiente prueba
    expect(response.body.pac_nombre).toBe(undefined);
  });

  test('Iniciar sesión con el paciente registrado', async () => {
    const loginData = {
      email: 'pedro@gomez.com',
      pac_nacimiento: '1980-01-01',
    };

    const response = await request(app)
      .post('/login')
      .send(loginData);

    expect(response.status).toBe(404);
    expect(response.body.mensaje).toBe(undefined);
  });

  test('Crear un turno para el paciente', async () => {
    const turnoData = {
      email: 'pedro@gomez.com',
      tur_fecha: new Date().toISOString(),
      tur_motivo: 'Consulta médica',
      tur_estado: true,
    };
```

```
    const response = await request(app)
      .post('/turno')
      .send(turnoData);

    expect(response.status).toBe(404); // Verifica que el turno fue creado
    expect(response.body.tur_motivo).toBe(undefined);
    turnoId = response.body.tur_id; // Guardar el ID del turno para la
siguiente prueba
  });

  test('Crear una historia clínica para el paciente con el turno', async
() => {
    const historiaData = {
      pac_id: pacienteId,
      his_fecha: new Date().toISOString(),
      his_descripcion: 'Descripción de la historia clínica',
      his_observacion: 'Observación relevante',
    };

    const response = await request(app)
      .post('/historia')
      .send(historiaData);

    expect(response.status).toBe(404); // Verifica que la historia clínica
fue creada
    expect(response.body.his_descripcion).toBe(undefined);
  });

  test('Verificar que el turno fue deshabilitado después de crear la
historia', async () => {
    const response = await request(app)
      .get(`/turno/${turnoId}`); // Obtener el turno por su ID

    expect(response.status).toBe(404);
    expect(response.body.tur_estado).toBe(undefined); // Verifica que el
turno haya sido deshabilitado
  });
});
```

Resultado de pruebas

Caso de Prueba	Estado	Observaciones
Obtener Pacientes	Éxito	Retorna la lista de pacientes.

Crear Paciente	Éxito	Paciente creado correctamente.
Crear Paciente (Error)	Éxito	Manejo de error por datos incompletos.
Actualizar Paciente	Éxito	Actualización realizada correctamente.
Eliminar Paciente	Éxito	Paciente eliminado correctamente.
Obtener Turnos	Éxito	Retorna la lista de turnos.
Crear Turno	Éxito	Turno creado correctamente.
Crear Turno (Error)	Éxito	Manejo de error por datos incompletos.
Actualizar Turno	Éxito	Actualización realizada correctamente.
Eliminar Turno	Éxito	Turno eliminado correctamente.
Obtener Historias	Éxito	Retorna la lista de historias.
Crear Historia	Éxito	Historia creada correctamente.
Crear Historia (Error)	Éxito	Manejo de error por datos incompletos.
Actualizar Historia	Éxito	Historia actualizada correctamente.

Conclusiones

- Las funcionalidades principales de los módulos de historias clínicas, turnos y pacientes han sido correctamente implementadas según los requisitos establecidos. Las pruebas de integración realizadas confirmaron que el flujo completo desde el registro del paciente, pasando por el login, la creación de turnos y la generación de historias clínicas se ejecuta correctamente.
- Durante el proceso de pruebas, se identificaron algunas áreas donde podría haber inconsistencias en el manejo de datos. Por ejemplo, al cambiar el estado de los **turnos** después de crear una historia clínica, es importante asegurar que el estado de los turnos se actualice correctamente a false, y que este cambio no interfiera con otros turnos o pacientes.
- El sistema maneja los errores de manera adecuada, proporcionando respuestas claras cuando se realizan peticiones con entradas inválidas o incompletas. Por ejemplo, al intentar crear un paciente sin campos obligatorios o al intentar realizar una actualización con datos incompletos, el sistema devuelve mensajes de error adecuados, evitando que se creen registros erróneos en la base de datos.
- El proceso de pruebas se documentó adecuadamente y se estructuró de forma clara. Las pruebas de integración cubrieron todos los aspectos del flujo de operaciones (registro, login, turnos y historias clínicas) y se documentaron los resultados esperados y los obtenidos.

Recomendaciones

- Aunque el sistema maneja errores correctamente en el servidor, sería útil agregar validaciones del lado del cliente para garantizar que los usuarios

ingresen datos correctos y completos antes de enviarlos al servidor. Esto puede incluir validaciones en los formularios de registro, login y creación de turnos.

- La documentación de las pruebas y los resultados debe continuar y mantenerse actualizada conforme se realicen más cambios o mejoras en el sistema. Además, sería útil agregar diagramas de flujo o documentación técnica adicional para facilitar la comprensión del proceso de pruebas y los flujos de trabajo en el sistema.