Logotipo

Descripción generada automáticamente con confianza baja



Tabla de contenido

[1. Introducción 2](#_heading=h.30j0zll)

[2. Alcance de las pruebas 2](#_heading=h.1fob9te)

[3. Herramientas y entornos de prueba 3](#_heading=h.3znysh7)

[4. Criterios de Entrada y Salida 4](#_heading=h.3dy6vkm)

[5. Tipos de Pruebas 5](#_heading=h.2s8eyo1)

[**6. Criterios de Aceptación** 7](#_heading=h.lnxbz9)

[7. Resultados 9](#_heading=h.2jxsxqh)

[**7.16** **Pruebas unitarias para la función updateProfile** 64](#_heading=h.3jtnz0s)

[**7.16** **Pruebas unitarias para la función updateUser** 68](#_heading=h.3bj1y38)

[**7.17** **Pruebas unitarias para la función verifyRecoveryCode** 72](#_heading=h.2pta16n)

[**7.18** **Pruebas unitarias para el controlador symptom\_classifier\_proba.py** 75](#_heading=h.j8sehv)

[**Resultados** 77](#_heading=h.2hio093)

[**Conclusión** 79](#_heading=h.wnyagw)

[**Análisis General** 80](#_heading=h.3gnlt4p)

# 1. Introducción

* **Objetivo**: Este plan de pruebas detalla las estrategias, procedimientos y alcance para verificar la calidad y funcionamiento del sistema Medical AID.
* **Alcance del proyecto**: Garantizar la correcta operación de las funcionalidades críticas del sistema para asegurar una experiencia de usuario óptima.

# 2. Alcance de las pruebas

**Las pruebas cubrirán los siguientes módulos:**

* Inicio de sesión.
* Registro de usuario.
* Recuperación de contraseña.
* Perfil de usuario.
* Administración de usuario.
* Diagnóstico médico.
* Biblioteca de diagnósticos.

# 3. Herramientas y entornos de prueba

#### **Herramientas de prueba:**

1. **Backend:**
   * **Node.js**:
     + Librerías: **Jest** y **Supertest** para pruebas unitarias e integración del backend.
     + IDE: **Visual Studio Code** para el desarrollo y ejecución de las pruebas.
2. **Modelo de Machine Learning:**
   * **Flask**:
     + Librerías: **pytest**, **numpy**, y **numpy.testing** para validar la conexión y el procesamiento de datos en el modelo.
     + IDE: **Visual Studio Code** para la implementación de pruebas en Flask.

#### **Entornos de prueba:**

1. **Entorno de desarrollo:**
   * Sistema operativo: Windows 10 o Windows 11.
   * Servidor de pruebas: Localhost con configuración de los entornos correspondientes para backend.
   * Versiones de herramientas:
     + Node.js: 20.18.0.
     + Flask: Flask 3.0.3.
     + Python: 3.12.7.
     + Werkzeug: 3.0.4.
2. **Entorno de integración:**
   * Base de datos de pruebas en el mismo servidor que aloja la aplicación para pruebas de integración.

# 4. Criterios de Entrada y Salida

#### **Criterios de Entrada**

Los criterios que deben cumplirse antes de iniciar las pruebas incluyen:

1. **Disponibilidad del código fuente:**
   * Todas las funcionalidades incluidas en el alcance están implementadas en el entorno de desarrollo.
2. **Definición clara de los requisitos:**
   * Las historias de usuario y criterios de aceptación están documentados y aprobados.
3. **Entornos configurados:**
   * El entorno de pruebas está correctamente configurado y accesible, incluyendo bases de datos, APIs y dependencias necesarias.
4. **Datos de prueba preparados:**
   * Conjuntos de datos de prueba representativos para validar cada funcionalidad.
5. **Herramientas listas para su uso:**
   * Jest, Supertest, Flask, pytest, numpy, y numpy.testing instalados y configurados en el entorno de desarrollo.
6. **Acceso a la documentación técnica:**
   * Documentación del sistema y del modelo de datos disponible para referencia.
7. **Entrega del plan de pruebas:**
   * El plan de pruebas revisado y aprobado por el equipo.

#### **Criterios de Salida**

Las condiciones que deben cumplirse para finalizar las pruebas incluyen:

1. **Cobertura de pruebas:**
   * Se han ejecutado todas las pruebas planificadas para las funcionalidades definidas en el alcance.
2. **Resultados satisfactorios:**
   * Al menos el 90% de los casos de prueba han sido exitosos, y los errores críticos han sido resueltos.
3. **Reporte de defectos:**
   * Todos los defectos encontrados están documentados y resueltos o clasificados como no críticos para la versión actual.
4. **Validación por el equipo de QA:**
   * El equipo de QA ha aprobado los resultados y asegura que el sistema cumple con los requisitos.
5. **Entrega de informes finales:**
   * Informes de ejecución de pruebas y matrices de cobertura están entregados y archivados.
6. **Preparación para la próxima fase:**
   * Se han identificado y planificado los pendientes para iteraciones futuras, si los hubiera.

# 5. Tipos de Pruebas

#### **Pruebas Unitarias**

**Objetivo**: Verificar el correcto funcionamiento de las unidades más pequeñas del sistema (funciones, métodos, controladores).

* **Características**:
  + Asegurar que cada unidad de código cumple con su propósito específico.
  + Detectar errores en una etapa temprana del desarrollo.
* **Herramientas**:
  + **Node.js**: Librería **Jest** para pruebas en el backend.
  + **Flask**: Librería **pytest** para validar funciones y componentes individuales del modelo de datos.
* **Ejemplos de funcionalidades probadas**:
  + Validación de credenciales en el inicio de sesión.
  + Formato de los datos registrados por el usuario.
  + Algoritmo de diagnóstico médico.

#### **Pruebas de Integración**

**Objetivo**: Validar que los módulos y componentes del sistema funcionan correctamente al interactuar entre sí.

* **Características**:
  + Detectar problemas de comunicación entre componentes.
  + Asegurar la correcta integración de servicios como APIs, bases de datos y modelos de datos.
* **Herramientas**:
  + **Node.js**: Librería **Supertest** para probar endpoints y APIs REST.
* **Ejemplos de funcionalidades probadas**:
  + Envío de datos desde el frontend al backend para registrar suscriptores.
  + Conexión con la base de datos.

#### **Pruebas Funcionales**

**Objetivo**: Garantizar que las funcionalidades del sistema cumplen con los requisitos especificados.

* **Características**:
  + Simular escenarios reales de uso para validar el comportamiento del sistema desde la perspectiva del usuario.
  + Basadas en los criterios de aceptación definidos en las historias de usuario.
* **Herramientas**:
  + Simulación manual o automatización con herramientas específicas según disponibilidad.
* **Ejemplos de funcionalidades probadas**:
  + Flujo completo de recuperación de contraseña, desde la solicitud hasta la actualización.
  + Registro de diagnóstico médico únicamente cuando el usuario está identificado.
  + Visualización de datos de perfil con las modificaciones realizadas.

### **6. Criterios de Aceptación**

#### **Clasificación de errores:**

1. **Errores Graves:**
   * Información crítica presentada de manera incorrecta o incompleta.
   * Datos mal registrados o inconsistentes en la base de datos.
   * Fallos que provocan caídas en el sistema o indisponibilidad de funciones principales.
   * Incumplimiento de los objetivos de funcionalidades clave (inicio de sesión, diagnóstico médico, etc.).
   * Vulnerabilidades de seguridad que exponen datos confidenciales.
2. **Errores Medios:**
   * Información incorrecta en documentos generados o exportados.
   * Fallos en la presentación de datos.
   * Incumplimiento de objetivos en funciones secundarias (visualización de datos de usuario no críticos, mensajes secundarios de alerta).
3. **Errores Leves:**
   * Presentación incorrecta de datos secundarios (información estética o no crítica).
   * No cumplimiento de estándares de diseño o usabilidad.
   * Dificultades menores en la operación que no impiden el flujo principal.

#### **Criterios de Aprobación:**

1. **Cobertura Total**:
   * Todas las pruebas planificadas (100%) deben ser ejecutadas, cubriendo tanto las funcionalidades críticas como las secundarias.
2. **Porcentaje de Aceptación**:
   * Al menos el 90% de las pruebas deben ser exitosas, sin errores graves ni fallos críticos en las funcionalidades principales.
3. **Errores Permitidos**:
   * Hasta un 10% de las pruebas pueden presentar errores de tipo medio o leve, pero no graves.
   * Los errores medios o leves no deben comprometer la seguridad, disponibilidad o integridad de los datos.

#### **Rechazo del Proyecto:**

* Si los errores graves exceden el 0% o el porcentaje de aceptación exitoso cae por debajo del 90%, el proyecto será rechazado en la etapa de certificación.
* Los errores detectados deben ser corregidos, y las pruebas reejecutadas antes de una nueva evaluación.

# 7. Resultados

**7.1 Prueba de integración para subscribeNewsletter**

#### **Objetivo:** Validar la funcionalidad del endpoint POST /api/newsletter/subscribe para:

1. Registrar nuevos suscriptores en la base de datos.
2. Evitar duplicados en las suscripciones.
3. Enviar un correo de confirmación al suscriptor.
4. Manejar errores de servidor de forma adecuada.

#### **Métodos:**

**Configuración inicial:**

* + Se creó una base de datos de prueba con la tabla NewsletterSubscribers sincronizada.
  + Se utilizó una instancia mock de nodemailer para simular el envío de correos.
  + Las pruebas se ejecutaron utilizando Supertest para realizar solicitudes HTTP al endpoint y Jest para las aserciones.

**Casos de prueba:**

* + **Caso 1:** Registrar un nuevo suscriptor y enviar un correo de confirmación.
  + **Caso 2:** Manejar intentos de suscripción duplicados con una respuesta de error (HTTP 400).
  + **Caso 3:** Simular un error en el servidor al intentar registrar un suscriptor, verificando la respuesta (HTTP 500).

#### **Resultados:**

**Caso 1: Registrar un nuevo suscriptor y enviar un correo de confirmación.**

* + **Resultado esperado:**
    - Respuesta con estado HTTP 201.
    - Mensaje: "¡Suscripción exitosa!"
    - Verificación de la entrada en la base de datos.
    - Correo de confirmación enviado al usuario.
  + **Resultado obtenido:** Cumplió todas las expectativas.

**Caso 2: Manejar intentos de suscripción duplicados.**

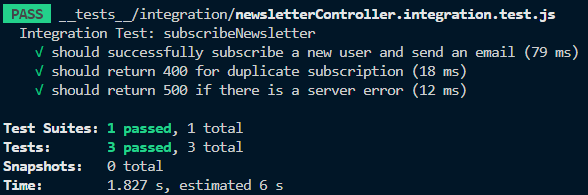
* + **Resultado esperado:**
    - Respuesta con estado HTTP 400.
    - Mensaje: "Este correo ya está suscrito."
  + **Resultado obtenido:** Cumplió todas las expectativas.

**Caso 3: Simular un error en el servidor.**

* + **Resultado esperado:**
    - Respuesta con estado HTTP 500.
    - Mensaje de error: "Error al suscribirse al boletín informativo."
  + **Resultado obtenido:** Cumplió todas las expectativas.

#### **Resumen de la ejecución:**

* **Número total de pruebas:** 3
* **Pruebas exitosas:** 3
* **Pruebas fallidas:** 0
* **Tiempo de ejecución:** 1.836 segundos



**7.2 Prueba de integración para conexión y consultas a la base de datos**

#### **Objetivo:** Validar la capacidad del sistema para:

1. Conectar correctamente a la base de datos.
2. Ejecutar consultas SQL básicas, garantizando la disponibilidad y consistencia de la base de datos.

#### **Métodos:**

1. **Configuración inicial:**
   * Conexión a la base de datos mediante el cliente proporcionado por el módulo dbtest.
   * Uso de beforeAll y afterAll para gestionar el ciclo de vida de la conexión y asegurar la liberación de recursos.
2. **Caso de prueba:**
   * Caso único: Establecer conexión con la base de datos y ejecutar una consulta para obtener la hora actual (SELECT NOW()).
3. **Herramientas utilizadas:**
   * Librería de pruebas: Jest.
   * Base de datos de prueba configurada previamente para evitar interferencias con datos reales.

#### **Resultados:**

**Caso único:**

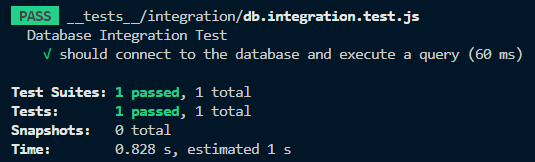
* + **Resultado esperado:**
    - Conexión exitosa a la base de datos.
    - Ejecución exitosa de la consulta SQL.
    - Resultado no nulo de la consulta.
  + **Resultado obtenido:**
    - La conexión fue exitosa.
    - La consulta fue ejecutada correctamente.
    - Los resultados de la consulta (rows) no fueron nulos, cumpliendo las expectativas.

**Logs generados:**

* + **Mensaje de conexión exitosa:** Connected to the test database.
  + **Liberación del cliente de la base de datos:** Database client released.
  + **Cierre de la conexión:** Database connection pool closed.

#### **Resumen de la ejecución:**

* **Número total de pruebas:** 1
* **Pruebas exitosas:** 1
* **Pruebas fallidas:** 0
* **Tiempo de ejecución:** 0.62 segundos



**7.3 Prueba unitaria para el controlador symptomsController**

#### **Objetivo:**

Validar el correcto funcionamiento de la única función del controlador, la cuál es getAllSymptomNames, asegurando que:

1. Recupera correctamente los nombres de los síntomas desde la base de datos.
2. Maneja adecuadamente los errores en caso de fallos.

#### **Métodos:**

1. **Configuración inicial:**
   * Se utilizó un mock del modelo Symptoms para simular la respuesta de la base de datos.
   * Se configuraron objetos mock de req y res para imitar solicitudes y respuestas HTTP.
   * Se empleó la librería **Jest** para realizar las pruebas unitarias.
2. **Casos de prueba:**
   * **Caso 1:** Recuperar todos los nombres de los síntomas exitosamente.
   * **Caso 2:** Manejar errores de base de datos y devolver un código de estado HTTP 500.
3. **Herramientas utilizadas:**
   * Mock de **Sequelize** para simular el modelo de base de datos.
   * **Jest** para realizar aserciones y verificar el comportamiento del controlador.

#### **Resultados:**

**Caso 1: Recuperar todos los nombres de los síntomas exitosamente.**

* + **Resultado esperado:**
    - Llamada al método findAll del modelo Symptoms con los atributos nombre, model\_order, y grupo\_sintomatico.
    - Respuesta con código de estado HTTP 200.
    - JSON con la lista de síntomas recuperados.
  + **Resultado obtenido:** Cumplió todas las expectativas.

**Caso 2: Manejar errores de base de datos y devolver un código de estado HTTP 500.**

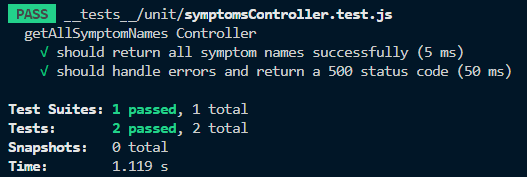
* + **Resultado esperado:**
    - Llamada al método findAll del modelo Symptoms.
    - Respuesta con código de estado HTTP 500.
    - JSON con el mensaje: "An error occurred while fetching symptom names".
  + **Resultado obtenido:** Cumplió todas las expectativas.

**Logs generados:**

* + Mensaje de error manejado: "Error fetching symptom names: Error: Database error".

#### **Resumen de la ejecución:**

* **Número total de pruebas:** 2
* **Pruebas exitosas:** 2
* **Pruebas fallidas:** 0
* **Tiempo de ejecución:** 0.841 segundos



**7.4 Prueba unitaria para el controlador subscribeNewsletter**

#### **Objetivo:** Validar el comportamiento del controlador subscribeNewsletter para:

1. Manejar correctamente suscripciones duplicadas.
2. Registrar nuevos suscriptores en la base de datos y enviar correos de confirmación.
3. Gestionar errores internos y devolver respuestas adecuadas.

#### **Métodos:**

1. **Configuración inicial:**
   * Se utilizaron mocks para:
     + El modelo NewsletterSubscribers para simular interacciones con la base de datos.
     + La librería nodemailer para simular el envío de correos electrónicos.
   * Objetos mock de req y res para imitar solicitudes y respuestas HTTP.
   * Uso de **Jest** para las aserciones y validaciones.
2. **Casos de prueba:**
   * **Caso 1:** Retornar un estado HTTP 400 si el correo ya está suscrito.
   * **Caso 2:** Agregar un nuevo suscriptor y enviar un correo de confirmación.
   * **Caso 3:** Manejar errores internos y devolver un estado HTTP 500.
3. **Herramientas utilizadas:**
   * Mock de Sequelize para el modelo.
   * Mock de nodemailer para simular envíos de correos.
   * **Jest** para realizar aserciones.

#### **Resultados:**

**Caso 1: Retornar un estado HTTP 400 si el correo ya está suscrito.**

* + **Resultado esperado:**
    - Verificación en la base de datos para identificar el correo duplicado.
    - Respuesta con código HTTP 400.
    - Mensaje: "Este correo ya está suscrito."
  + **Resultado obtenido:** Cumplió todas las expectativas.

**Caso 2: Agregar un nuevo suscriptor y enviar un correo de confirmación.**

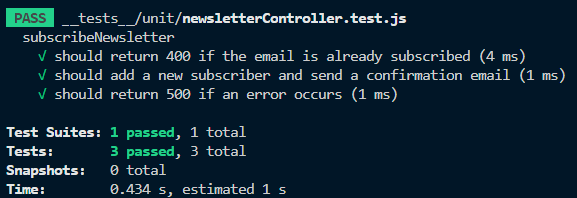
* + **Resultado esperado:**
    - Verificación en la base de datos que confirma que el correo no está registrado.
    - Inserción exitosa del nuevo suscriptor.
    - Envío del correo de confirmación.
    - Respuesta con código HTTP 201.
    - Mensaje: "¡Suscripción exitosa!"
  + **Resultado obtenido:** Cumplió todas las expectativas.

**Caso 3: Manejar errores internos y devolver un estado HTTP 500.**

* + **Resultado esperado:**
    - Respuesta con código HTTP 500.
    - Mensaje: "Error al suscribirse al boletín informativo."
  + **Resultado obtenido:** Cumplió todas las expectativas.

#### **Resumen de la ejecución:**

* **Número total de pruebas:** 3
* **Pruebas exitosas:** 3
* **Pruebas fallidas:** 0
* **Tiempo de ejecución:** 0.479 segundos



**7.5 Prueba unitaria para la función createDiagnosis**

#### **Objetivo:** Verificar que la función createDiagnosis:

1. Crea diagnósticos nuevos con los datos, síntomas y enfermedades asociados.
2. Identifica y responde adecuadamente si ya existe una sesión de diagnóstico registrada.
3. Maneja errores internos y responde con el estado correspondiente.

#### **Métodos:**

1. **Configuración inicial:**
   * Se utilizaron mocks para los modelos de Sequelize: Diagnoses, Symptoms, DiagnosisSymptoms, y DiagnosisDisease.
   * Datos de entrada proporcionados mediante el objeto mock req con información de diagnóstico y sesión.
   * Respuesta HTTP simulada mediante un objeto mock res.
   * Uso de **Jest** para las aserciones y validación de comportamientos.
2. **Casos de prueba:**
   * **Caso 1:** Crear un diagnóstico nuevo con los datos proporcionados**.**
   * **Caso 2:** Manejar solicitudes para una sesión de diagnóstico ya existente.
   * **Caso 3:** Gestionar errores internos y devolver un estado HTTP 500.
3. **Herramientas utilizadas:**
   * Mock de Sequelize para interacciones con la base de datos.
   * **Jest** para la ejecución de pruebas y validaciones.

#### **Resultados:**

**Caso 1: Crear un diagnóstico nuevo con los datos proporcionados.**

* + **Resultado esperado:**
    - Verificación de que la sesión de diagnóstico no existe en la base de datos.
    - Creación de un nuevo registro en Diagnoses.
    - Asociación de síntomas y enfermedades al diagnóstico.
    - Respuesta con código HTTP 201.
    - Mensaje: "Diagnosis data, symptoms, and diseases saved successfully".
  + **Resultado obtenido:** Cumplió todas las expectativas.

**Caso 2: Manejar solicitudes para una sesión de diagnóstico ya existente.**

* + **Resultado esperado:**
    - Verificación de que la sesión de diagnóstico ya existe.
    - Respuesta con código HTTP 409.
    - Mensaje: "Diagnosis already registered for this session."
  + **Resultado obtenido: Cumplió todas las expectativas.**

**Caso 3: Gestionar errores internos y devolver un estado HTTP 500.**

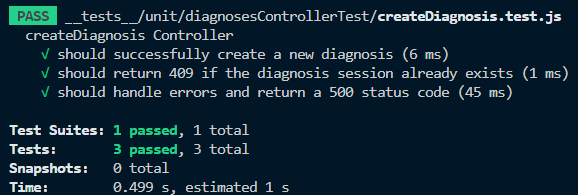
* + **Resultado esperado:**
    - Registro de un error en la consola.
    - Respuesta con código HTTP 500.
    - Mensaje: "Failed to save diagnosis data."
  + **Resultado obtenido:** Cumplió todas las expectativas**.**

**Logs generados:**

* + Mensaje de error manejado: "Error saving diagnosis data: Error: Database error".

#### **Resumen de la ejecución:**

* **Número total de pruebas:** 3
* **Pruebas exitosas:** 3
* **Pruebas fallidas:** 0
* **Tiempo de ejecución:** 0.45 segundos



**7.5 Prueba unitaria para la función getUserDiagnosticRecords**

#### **Objetivo:** Validar que la función getUserDiagnosticRecords:

1. Recupera correctamente los registros de diagnóstico de un usuario específico.
2. Formatea los datos para incluir enfermedades y síntomas relacionados con cada diagnóstico.
3. Maneja errores internos y responde con el estado correspondiente.

#### **Métodos:**

1. **Configuración inicial:**
   * Se configuraron mocks para los modelos de Sequelize: Diagnoses, DiagnosisDisease, Disease, DiagnosisSymptoms, y Symptoms.
   * Se utilizó un objeto mock req con el parámetro userId para identificar al usuario.
   * Respuesta HTTP simulada mediante un objeto mock res.
   * Uso de **Jest** para realizar las aserciones.
2. **Casos de prueba:**
   * **Caso 1:** Recuperar y formatear correctamente los registros de diagnóstico.
   * **Caso 2:** Manejar errores internos y devolver un estado HTTP 500.
3. **Herramientas utilizadas:**
   * Mock de Sequelize para simular interacciones con la base de datos.
   * **Jest** para la ejecución de pruebas y validaciones.

#### **Resultados:**

**Caso 1: Recuperar y formatear correctamente los registros de diagnóstico.**

* + **Resultado esperado:**
    - Consultar la base de datos para recuperar diagnósticos del usuario.
    - Incluir enfermedades y síntomas relacionados con cada diagnóstico.
    - Respuesta con el formato esperado y código HTTP 200.
  + **Resultado obtenido:** Cumplió todas las expectativas.

**Respuesta esperada:**[

{

"id\_diagnosis": 101,

"id\_user": 1,

"diagnosis\_date": "2024-11-28T12:00:00Z",

"diseases": [

{ "id\_disease": 1, "disease\_name": "Disease A" },

{ "id\_disease": 2, "disease\_name": "Disease B" }

],

"symptoms": [

{ "id\_symptom": 1, "symptom\_name": "Symptom A" },

{ "id\_symptom": 2, "symptom\_name": "Symptom B" }

]

}

]

**Caso 2: Manejar errores internos y devolver un estado HTTP 500.**

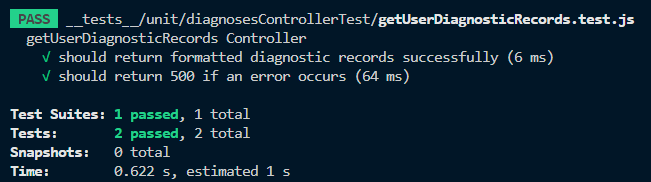
* + **Resultado esperado:**
    - Registrar un error en la consola.
    - Responder con código HTTP 500.
    - Mensaje: "Error fetching diagnostic records".
  + **Resultado obtenido:** Cumplió todas las expectativas.

**Logs generados:**

* + Mensaje de error manejado: "Error fetching diagnostic records: Error: Database error".

#### **Resumen de la ejecución:**

* **Número total de pruebas:** 2
* **Pruebas exitosas:** 2
* **Pruebas fallidas:** 0
* **Tiempo de ejecución:** 0.484 segundos



**7.6 Prueba unitaria para la función getAllDiseases**

#### **Objetivo**: Validar que la función getAllDiseases:

1. Recupera correctamente todas las enfermedades desde la base de datos.
2. Maneja errores internos y responde con un estado adecuado.

#### **Métodos:**

1. **Configuración inicial:**
   * Se configuró un mock para el modelo Disease de Sequelize, simulando el comportamiento del método findAll.
   * Respuesta HTTP simulada mediante un objeto mock res.
   * Uso de **Jest** para realizar las aserciones.
2. **Casos de prueba:**
   * **Caso 1:** Recuperar correctamente todas las enfermedades.
   * **Caso 2:** Manejar errores internos y devolver un estado HTTP 500.
3. **Herramientas utilizadas:**
   * Mock de Sequelize para el modelo Disease.
   * **Jest** para la ejecución de pruebas y validaciones.

#### **Resultados:**

**Caso 1: Recuperar correctamente todas las enfermedades.**

* + **Resultado esperado:**
    - Llamada exitosa al método findAll del modelo Disease.
    - Respuesta con el listado de enfermedades en formato JSON.
    - Código HTTP 200.
  + **Resultado obtenido:** Cumplió todas las expectativas.

**Ejemplo de respuesta esperada:**  
[

{ "id": 1, "nombre": "Resfrío", "model\_order": 1 },

{ "id": 2, "nombre": "Tos", "model\_order": 2 }

]

**Caso 2: Manejar errores internos y devolver un estado HTTP 500.**

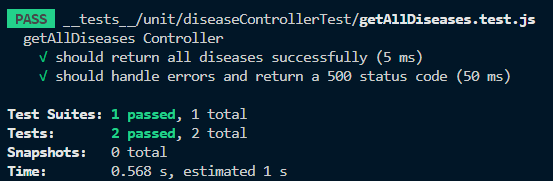
* + **Resultado esperado:**
    - Registro del error en la consola.
    - Respuesta con código HTTP 500.
    - Mensaje: "Server error".
  + **Resultado obtenido:** Cumplió todas las expectativas.

**Logs generados:**

* + Mensaje de error manejado: "Error fetching diseases: Error: Database error".

#### **Resumen de la ejecución:**

* **Número total de pruebas:** 2
* **Pruebas exitosas:** 2
* **Pruebas fallidas:** 0
* **Tiempo de ejecución:** 0.482 segundos



**7.5 Prueba unitaria para el controlador getDiseaseByModelOrder**

#### **Objetivo:**

Validar que el controlador getDiseaseByModelOrder:

1. Devuelve la información de una enfermedad asociada a un model\_order válido.
2. Maneja casos en los que no se encuentra una enfermedad para el model\_order.
3. Gestiona errores internos devolviendo el estado correspondiente.

#### **Métodos:**

1. **Configuración inicial:**
   * Se configuró un mock para el modelo Disease de Sequelize, simulando el comportamiento del método findOne.
   * Respuesta HTTP simulada mediante un objeto mock res.
   * Uso de **Jest** para realizar las aserciones.
2. **Casos de prueba:**
   * **Caso 1:** Retornar una enfermedad válida para un model\_order dado.
   * **Caso 2:** Manejar casos donde no se encuentra una enfermedad para el model\_order.
   * **Caso 3:** Manejar errores internos y devolver un estado HTTP 500.
3. **Herramientas utilizadas:**
   * Mock de Sequelize para el modelo Disease.
   * Jest para la ejecución de pruebas y validaciones.

#### **Resultados:**

**Caso 1: Retornar una enfermedad válida para un model\_order dado.**

* + **Resultado esperado:**
    - Llamada exitosa al método findOne del modelo Disease.
    - Respuesta con los datos de la enfermedad en formato JSON.
    - Código HTTP 200.
  + **Resultado obtenido:** Cumplió todas las expectativas.

**Ejemplo de respuesta esperada:**{ "id": 1, "nombre": "Flu", "model\_order": 1 }

**Caso 2: Manejar casos donde no se encuentra una enfermedad para el model\_order.**

* + **Resultado esperado:**
    - Llamada al método findOne que retorna null.
    - Respuesta con código HTTP 404.
    - Mensaje: "Disease not found".
  + **Resultado obtenido:** Cumplió todas las expectativas.

**Caso 3: Manejar errores internos y devolver un estado HTTP 500.**

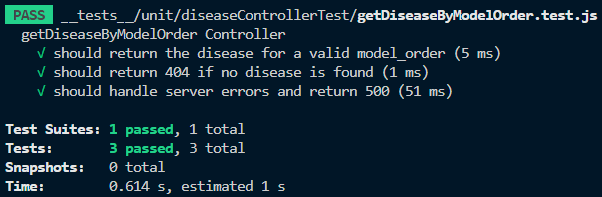
* + **Resultado esperado:**
    - Registro del error en la consola.
    - Respuesta con código HTTP 500.
    - Mensaje: "Server error".
  + **Resultado obtenido:** Cumplió todas las expectativas.

**Logs generados:**

* + Mensaje de error manejado: "Error fetching disease: Error: Database error".

#### **Resumen de la ejecución:**

* **Número total de pruebas:** 3
* **Pruebas exitosas:** 3
* **Pruebas fallidas:** 0
* **Tiempo de ejecución:** 0.456 segundos



**7.6 Prueba unitaria para la función getDiseaseByName**

#### **Objetivo:** Validar que el controlador getDiseaseByName**:**

1. Devuelve correctamente la información de una enfermedad basada en su nombre.
2. Maneja casos en los que no se encuentra una enfermedad con el nombre proporcionado.
3. Gestiona errores internos devolviendo un estado HTTP 500.

#### **Métodos:**

1. **Configuración inicial:**
   * Se configuró un mock para el modelo Disease de Sequelize, simulando el comportamiento del método findOne.
   * Respuesta HTTP simulada mediante un objeto mock res.
   * Uso de **Jest** para realizar las aserciones.
2. **Casos de prueba:**
   * **Caso 1:** Retornar una enfermedad válida cuando el nombre existe.
   * **Caso 2:** Manejar casos donde no se encuentra una enfermedad con el nombre proporcionado.
   * **Caso 3:** Manejar errores internos y devolver un estado HTTP 500.
3. **Herramientas utilizadas:**
   * Mock de Sequelize para el modelo Disease.
   * Jest para la ejecución de pruebas y validaciones.

#### **Resultados:**

**Caso 1: Retornar una enfermedad válida cuando el nombre existe.**

* + **Resultado esperado:**
    - Llamada exitosa al método findOne del modelo Disease.
    - Respuesta con los datos de la enfermedad en formato JSON.
    - Código HTTP 200.
  + **Resultado obtenido: Cumplió todas las expectativas.**

**Ejemplo de respuesta esperada:**{ "id": 1, "nombre": "Flu", "model\_order": 1 **}**

**Caso 2: Manejar casos donde no se encuentra una enfermedad con el nombre proporcionado.**

* + **Resultado esperado:**
    - Llamada al método findOne que retorna null.
    - Respuesta con código HTTP 404.
    - Mensaje: "Disease not found".
  + **Resultado obtenido:** Cumplió todas las expectativas.

**Caso 3: Manejar errores internos y devolver un estado HTTP 500.**

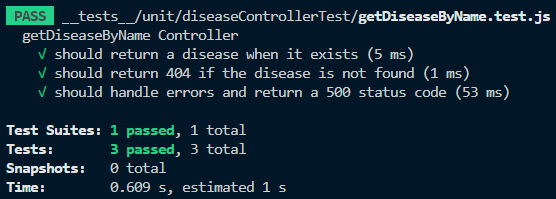
* + **Resultado esperado:**
    - Registro del error en la consola.
    - Respuesta con código HTTP 500.
    - Mensaje: "Server error".
  + **Resultado obtenido:** Cumplió todas las expectativas.

**Logs generados:**

* + **Mensaje de error manejado:** "Error fetching disease: Error: Database error".

#### **Resumen de la ejecución:**

* **Número total de pruebas:** 3
* **Pruebas exitosas:** 3
* **Pruebas fallidas:** 0
* **Tiempo de ejecución:** 0.447 segundos

****

**7.7 Prueba unitaria para la función changePassword**

#### **Objetivo:** Validar que la función changePassword:

1. Cambia correctamente la contraseña de un usuario autenticado.
2. Maneja casos donde el usuario no existe.
3. Gestiona errores internos devolviendo respuestas adecuadas.

#### **Métodos:**

1. **Configuración inicial:**
   * Se configuró un mock para el modelo Users de Sequelize, simulando el comportamiento del método findOne y update.
   * Se utilizó un mock para bcryptjs para simular la generación de salts y el hash de la contraseña.
   * Respuesta HTTP simulada mediante un objeto mock res.
   * Uso de **Jest** para realizar las aserciones.
2. **Casos de prueba:**
   * **Caso 1:** Cambiar correctamente la contraseña de un usuario autenticado.
   * **Caso 2:** Manejar casos donde el usuario no existe y devolver un estado HTTP 404.
   * **Caso 3:** Gestionar errores internos y devolver un estado HTTP 500.
3. **Herramientas utilizadas:**
   * Mock de Sequelize para el modelo Users.
   * Mock de bcryptjs para simular la generación de contraseñas.
   * Jest para la ejecución de pruebas y validaciones.

#### **Resultados:**

**Caso 1: Cambiar correctamente la contraseña de un usuario autenticado.**

* + **Resultado esperado:**
    - Llamada exitosa al método findOne para verificar la existencia del usuario.
    - Generación de salt y hash de la nueva contraseña con bcryptjs.
    - Actualización de la contraseña en la base de datos.
    - Respuesta con código HTTP 200.
    - Mensaje: "Password updated successfully."
  + **Resultado obtenido: Cumplió todas las expectativas.**

**Caso 2: Manejar casos donde el usuario no existe y devolver un estado HTTP 404.**

* + **Resultado esperado:**
    - Llamada al método findOne que retorna null.
    - Respuesta con código HTTP 404.
    - Mensaje: "User not found."
  + **Resultado obtenido: Cumplió todas las expectativas.**

**Caso 3: Gestionar errores internos y devolver un estado HTTP 500.**

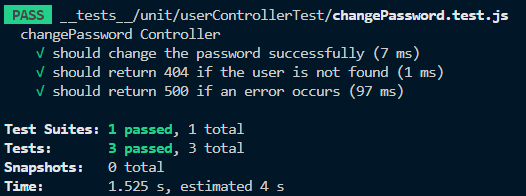
* + **Resultado esperado:**
    - Registro del error en la consola.
    - Respuesta con código HTTP 500.
    - Mensaje: "An error occurred while changing the password."
  + **Resultado obtenido:** Cumplió todas las expectativas.

**Logs generados:**

* + **Mensaje de error manejado:** "Error changing password: Error: Database error".

#### **Resumen de la ejecución:**

* **Número total de pruebas:** 3
* **Pruebas exitosas:** 3
* **Pruebas fallidas:** 0
* **Tiempo de ejecución:** 1.627 segundos



**7.8 Prueba unitaria para la función deleteUser**

#### **Objetivo:** Validar que la función deleteUser:

1. Realiza correctamente la eliminación lógica de un usuario existente.
2. Maneja casos en los que no se encuentra el usuario solicitado.
3. Gestiona errores internos devolviendo respuestas adecuadas.

#### **Métodos:**

1. **Configuración inicial:**
   * Se configuró un mock para el modelo Users de Sequelize, simulando el comportamiento de los métodos findOne y update.
   * Respuesta HTTP simulada mediante un objeto mock res.
   * Uso de **Jest** para realizar las aserciones.
2. **Casos de prueba:**
   * **Caso 1:** Eliminar lógicamente a un usuario existente.
   * **Caso 2:** Manejar casos en los que el usuario no existe y devolver un estado HTTP 404.
   * **Caso 3:** Gestionar errores internos y devolver un estado HTTP 500.
3. **Herramientas utilizadas:**
   * Mock de Sequelize para el modelo Users.
   * **Jest** para la ejecución de pruebas y validaciones.

#### **Resultados:**

**Caso 1: Eliminar lógicamente a un usuario existente.**

* + **Resultado esperado:**
    - Llamada exitosa al método findOne para verificar la existencia del usuario.
    - Ejecución del método update para cambiar el estado del usuario a inactivo.
    - Respuesta con código HTTP 200.
    - Mensaje: "Usuario eliminado lógicamente con éxito."
  + **Resultado obtenido:** Cumplió todas las expectativas.

**Caso 2: Manejar casos en los que el usuario no existe y devolver un estado HTTP 404.**

* + **Resultado esperado:**
    - Llamada al método findOne que retorna null.
    - Respuesta con código HTTP 404.
    - Mensaje: "Usuario no encontrado."
  + **Resultado obtenido:** Cumplió todas las expectativas.

**Caso 3: Gestionar errores internos y devolver un estado HTTP 500.**

* + **Resultado esperado:**
    - Registro del error en la consola.
    - Respuesta con código HTTP 500.
    - Mensaje: "Error del servidor."
  + **Resultado obtenido:** Cumplió todas las expectativas.

**Logs generados:**

* + Mensaje de error manejado: "Error deleting user: Error: Database error".

#### **Resumen de la ejecución:**

* **Número total de pruebas:** 3
* **Pruebas exitosas:** 3
* **Pruebas fallidas:** 0
* **Tiempo de ejecución:** 1.249 segundos

**7.9 Prueba unitaria para la función deleteUser**

Prueba unitaria para el controlador getAllUsers

#### **Objetivo:** Validar que el controlador getAllUsers:

1. Recupera correctamente todos los usuarios junto con sus perfiles y roles asociados.
2. Maneja errores internos y devuelve respuestas adecuadas.

#### **Métodos:**

1. **Configuración inicial:**
   * Se configuró un mock para el modelo Users de Sequelize, simulando el comportamiento del método findAll.
   * Respuesta HTTP simulada mediante un objeto mock res.
   * Uso de **Jest** para realizar las aserciones.
2. **Casos de prueba:**
   * **Caso 1:** Recuperar todos los usuarios con éxito.
   * **Caso 2:** Manejar errores internos y devolver un estado HTTP 500.
3. **Herramientas utilizadas:**
   * Mock de Sequelize para el modelo Users.
   * **Jest** para la ejecución de pruebas y validaciones.

#### **Resultados:**

1. **Caso 1: Recuperar todos los usuarios con éxito.**
   * **Resultado esperado:**
     + Llamada exitosa al método findAll del modelo Users.
     + Recuperación de datos de usuarios junto con sus perfiles y roles.
     + Respuesta con el listado de usuarios en formato JSON.
     + Código HTTP 200.
   * **Resultado obtenido:** Cumplió todas las expectativas.

* **Ejemplo de respuesta esperada:**[

{

"id\_user": 1,

"email": "test1@example.com",

"created\_at": "2024-11-23T10:00:00Z",

"active": true,

"profile": {

"names": "John",

"last\_names": "Doe"

},

"roles": [

{

"id\_role": 1,

"role": {

"role\_name": "Admin"

}

}

]

},

{

"id\_user": 2,

"email": "test2@example.com",

"created\_at": "2024-11-22T10:00:00Z",

"active": false,

"profile": {

"names": "Jane",

"last\_names": "Smith"

},

"roles": [

{

"id\_role": 2,

"role": {

"role\_name": "User"

}

}

]

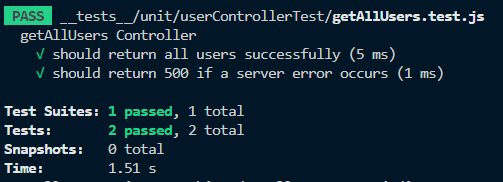
}

]

1. **Caso 2: Manejar errores internos y devolver un estado HTTP 500.**
   * **Resultado esperado:**
     + Registro del error en la consola.
     + Respuesta con código HTTP 500.
     + Mensaje: "Error fetching users."
   * **Resultado obtenido:** Cumplió todas las expectativas.
2. **Logs generados:**
   * Mensaje de error manejado: "Error fetching users:".

#### **Resumen de la ejecución:**

* **Número total de pruebas:** 2
* **Pruebas exitosas:** 2
* **Pruebas fallidas:** 0
* **Tiempo de ejecución:** 1.154 segundos



**7.10 Prueba unitaria para la función getUserById**

#### **Objetivo:** Validar que la función getUserById:

1. Devuelve correctamente la información de un usuario existente, incluyendo su perfil y roles.
2. Maneja casos en los que el ID del usuario es inválido.
3. Gestiona casos en los que el usuario no existe.
4. Gestiona errores internos devolviendo respuestas adecuadas.

#### **Métodos:**

1. **Configuración inicial:**
   * Se configuró un mock para los modelos Users, UserProfiles, UserRoles y Roles de Sequelize, simulando el comportamiento del método findOne.
   * Respuesta HTTP simulada mediante un objeto mock res.
   * Uso de **Jest** para realizar las aserciones.
2. **Casos de prueba:**
   * **Caso 1:** Validar un ID de usuario inválido.
   * **Caso 2:** Manejar casos donde el usuario no existe y devolver un estado HTTP 404.
   * **Caso 3:** Recuperar los datos del usuario correctamente.
   * **Caso 4:** Manejar errores internos y devolver un estado HTTP 500.
3. **Herramientas utilizadas:**
   * Mock de Sequelize para los modelos relacionados con usuarios.
   * **Jest** para la ejecución de pruebas y validaciones.

#### **Resultados:**

**Caso 1: Validar un ID de usuario inválido.**

* + **Resultado esperado:**
    - Respuesta con código HTTP 400.
    - Mensaje: "Invalid user ID provided."
  + **Resultado obtenido:** Cumplió todas las expectativas.

**Caso 2: Manejar casos donde el usuario no existe y devolver un estado HTTP 404.**

* + **Resultado esperado:**
    - Llamada al método findOne que retorna null.
    - Respuesta con código HTTP 404.
    - Mensaje: "User not found."
  + **Resultado obtenido:** Cumplió todas las expectativas.

**Caso 3: Recuperar los datos del usuario correctamente.**

* + **Resultado esperado:**
    - Llamada exitosa al método findOne para recuperar los datos del usuario, incluyendo su perfil y roles.
    - Respuesta con los datos del usuario en formato JSON.
    - Código HTTP 200.
  + **Resultado obtenido:** Cumplió todas las expectativas.

**Ejemplo de respuesta esperada:**json  
Copiar código  
{

"id\_user": 1,

"email": "test@example.com",

"active": true,

"profile": {

"names": "John",

"last\_names": "Doe",

"birthdate": "1990-01-01",

"weight": 75,

"height": 180,

"gender": "Masculino",

"address": "123 Test St",

"comune": "Test City",

"phone\_number": "123456789"

},

"roles": [

{

"role": {

"role\_name": "Admin"

}

}

]

}

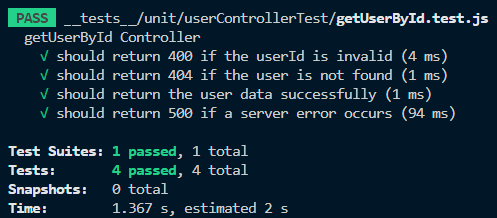
**Caso 4: Manejar errores internos y devolver un estado HTTP 500.**

* + **Resultado esperado:**
    - Registro del error en la consola.
    - Respuesta con código HTTP 500.
    - Mensaje: "Error fetching user data."
  + **Resultado obtenido:** Cumplió todas las expectativas.

1. **Logs generados:**
   * Mensaje de error manejado: "Error fetching user data: Error: Database error"

#### **Resumen de la ejecución:**

* **Número total de pruebas:** 4
* **Pruebas exitosas:** 4
* **Pruebas fallidas:** 0
* **Tiempo de ejecución:** 1.289 segundos



#### **7.11 Prueba unitaria para el controlador loginUser**

#### **Objetivo:**

Validar que el controlador loginUser:

1. Permite el inicio de sesión de un usuario con credenciales válidas y crea una nueva sesión.
2. Actualiza correctamente la sesión activa existente.
3. Maneja correctamente errores como credenciales inválidas, usuarios bloqueados y errores internos.

#### **Métodos:**

1. **Configuración inicial:**
   * Simulación de dependencias externas como bcryptjs, jsonwebtoken, y modelos Sequelize (Users, Sessions).
   * Creación de objetos req y res simulados.
   * Uso de **Jest** para la simulación y validación.
2. **Casos de prueba:**
   * Validación correcta del método bcrypt.compare.
   * Inicio de sesión exitoso con creación de una nueva sesión.
   * Actualización de una sesión activa existente.
   * Gestión de errores:
     + Email no encontrado.
     + Usuario bloqueado.
     + Error interno en el servidor.
3. **Herramientas utilizadas:**
   * Mock de Sequelize para las interacciones con la base de datos.
   * Mock de bcryptjs y jsonwebtoken para el manejo de autenticación.
   * **Jest** para la ejecución de pruebas y validaciones.

#### **Resultados:**

1. **Caso 1: Validar bcrypt.compare.**
   * **Resultado esperado:**
     + Respuesta booleana correcta (true) para contraseñas coincidentes.
   * **Resultado obtenido:** Cumplió las expectativas.
   * **Log generado:** bcrypt.compare result: true

**Caso 2: Inicio de sesión exitoso con creación de nueva sesión.**

* + **Resultado esperado:**
    - Creación de una nueva sesión en la base de datos.
    - Generación de un token JWT.
    - Respuesta HTTP con código 200.
  + **Resultado obtenido:** Cumplió las expectativas.

**Respuesta esperada:**  
{

"message": "Inicio de sesión exitoso",

"token": "mockToken",

"userId": 1,

"email": "test@example.com",

"nombre": "John Doe"

}

**Caso 3: Actualización de sesión activa.**

* + **Resultado esperado:**
    - Actualización del token y el tiempo de expiración de la sesión activa.
    - Respuesta HTTP con código 200.
  + **Resultado obtenido:** Cumplió las expectativas.

**Respuesta esperada:**  
{

"message": "Ya existe una sesión activa",

"token": "updatedMockToken"

}

**Caso 4: Gestión de errores.**

* + **Email no encontrado:**
    - **Resultado esperado:** Respuesta con código 400 y mensaje de error.
    - **Resultado obtenido:** Cumplió las expectativas.

**Respuesta esperada:**  
{

"error": "Correo o contraseña incorrectos"

}

* + **Usuario bloqueado:**
    - **Resultado esperado:** Respuesta con código 403 y tiempo restante para desbloqueo.
    - **Resultado obtenido:** Cumplió las expectativas.

**Respuesta esperada:**  
{

"error": "Cuenta bloqueada. Intente nuevamente en 5 minutos."

}

* + **Error interno en el servidor:**
    - **Resultado esperado:** Respuesta con código 500 y mensaje de error genérico.
    - **Resultado obtenido:** Cumplió las expectativas.

**Respuesta esperada:**  
{

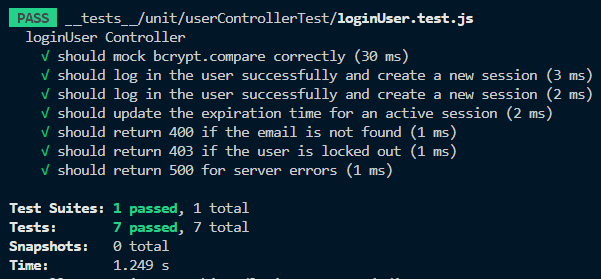
"error": "Error del servidor"

}

1. **Logs generados:**
   * Para errores: "Error del servidor"

#### **Resumen de la ejecución:**

* **Número total de pruebas:** 7
* **Pruebas exitosas:** 7
* **Pruebas fallidas:** 0
* **Tiempo de ejecución:** 1.18 segundos



#### **7.12 Prueba unitaria para la función logoutUser**

#### **Objetivo:**

Validar que la función logoutUser:

1. Maneja correctamente los casos en los que falta el token.
2. Elimina exitosamente una sesión de usuario utilizando el token proporcionado.
3. Maneja correctamente errores como sesiones inexistentes y errores internos del servidor.

#### **Métodos:**

1. **Configuración inicial:**
   * Simulación de dependencias externas como el modelo Sequelize Sessions.
   * Creación de objetos req y res simulados con encabezados y respuestas HTTP.
   * Uso de **Jest** para la simulación y validación.
2. **Casos de prueba:**
   * Token no proporcionado.
   * Sesión inexistente.
   * Éxito al cerrar sesión.
   * Error interno del servidor.
3. **Herramientas utilizadas:**
   * Mock de Sequelize para las interacciones con la base de datos.
   * **Jest** para la ejecución de pruebas y validaciones.

#### **Resultados:**

**Caso 1: Token no proporcionado.**

* + **Resultado esperado:**
    - Respuesta con código 400 y mensaje de error indicando la falta del token.
  + **Resultado obtenido:** Cumplió las expectativas.

**Respuesta esperada:**  
{

"message": "Token not provided."

}

**Caso 2: Sesión inexistente.**

* + **Resultado esperado:**
    - Respuesta con código 404 y mensaje indicando que la sesión no existe o ya se ha cerrado.
  + **Resultado obtenido:** Cumplió las expectativas.

**Respuesta esperada:**  
{

"message": "Session not found or already logged out."

}

**Caso 3: Éxito al cerrar sesión.**

* + **Resultado esperado:**
    - Respuesta con código 200 indicando que el cierre de sesión fue exitoso.
  + **Resultado obtenido:** Cumplió las expectativas.

**Respuesta esperada:**  
{

"message": "Logged out successfully."

}

**Caso 4: Error interno del servidor.**

* + **Resultado esperado:**
    - Respuesta con código 500 y mensaje de error genérico.
    - Registro del error en los logs.
  + **Resultado obtenido:** Cumplió las expectativas.

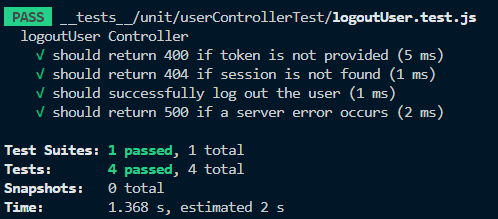
**Respuesta esperada:**  
{

"message": "Server error during logout."

}

#### **Resumen de la ejecución:**

* **Número total de pruebas:** 4
* **Pruebas exitosas:** 4
* **Pruebas fallidas:** 0
* **Tiempo de ejecución:** 1.216 segundos



#### **7.13 Prueba unitaria para la función reactivateUser**

#### **Objetivo:** Validar que la función reactivateUser:

1. Maneja correctamente los casos en los que el usuario no existe.
2. Reactiva con éxito a un usuario desactivado.
3. Maneja errores internos del servidor.

#### **Métodos:**

1. **Configuración inicial:**
   * Simulación de dependencias externas como el modelo Sequelize Users.
   * Creación de objetos req y res simulados con parámetros HTTP y respuestas.
   * Uso de **Jest** para la simulación y validación.
2. **Casos de prueba:**
   * Usuario no encontrado.
   * Reactivación exitosa.
   * Error interno del servidor.
3. **Herramientas utilizadas:**
   * Mock de Sequelize para interacciones con la base de datos.
   * **Jest** para la ejecución de pruebas y validaciones.

#### **Resultados:**

**Caso 1: Usuario no encontrado.**

* + **Resultado esperado:**
    - Respuesta con código 404 y mensaje de error indicando que el usuario no fue encontrado.
  + **Resultado obtenido:** Cumplió las expectativas.

**Respuesta esperada:**  
{

"error": "User not found"

}

**Caso 2: Reactivación exitosa.**

* + **Resultado esperado:**
    - Respuesta con código 200 y mensaje indicando que la reactivación fue exitosa.
  + **Resultado obtenido:** Cumplió las expectativas.

**Respuesta esperada:**  
{

"message": "User reactivated successfully"

}

**Caso 3: Error interno del servidor.**

* + **Resultado esperado:**
    - Respuesta con código 500 y mensaje de error genérico.
    - Registro del error en los logs.
  + **Resultado obtenido:** Cumplió las expectativas.

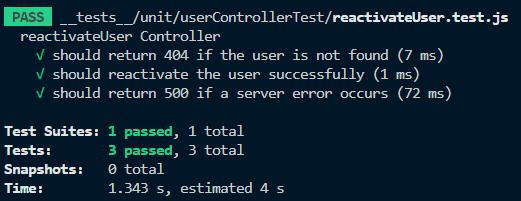
**Respuesta esperada:**  
{

"error": "An error occurred while reactivating the user"

}

#### **Resumen de la ejecución:**

* **Número total de pruebas:** 3
* **Pruebas exitosas:** 3
* **Pruebas fallidas:** 0
* **Tiempo de ejecución:** 1.398 segundos



**7.14 Pruebas unitarias para la función registerAdmin**

#### **Objetivo:** Validar el comportamiento de la función registerAdmin en escenarios clave:

1. Registro exitoso de un administrador.
2. Manejo de errores cuando el correo ya está registrado.
3. Manejo de errores de validación de datos.
4. Manejo de errores internos del servidor.

#### **Casos de Prueba y Resultados:**

**Caso 1: Registro exitoso.**

* + **Configuración:** No existe un usuario registrado con el correo proporcionado.
  + **Resultados esperados:**
    - Usuario creado con éxito.
    - Respuesta con código HTTP 201.
    - Mensaje de éxito retornado.
  + **Resultados obtenidos:** Cumplió las expectativas.

**Ejemplo de respuesta:**  
{

"message": "Admin registrado con éxito",

"userId": 1

}

**Caso 2: Correo ya registrado.**

* + **Configuración:** Existe un usuario registrado con el correo proporcionado.
  + **Resultados esperados:**
    - Respuesta con código HTTP 400.
    - Mensaje de error retornado.
  + **Resultados obtenidos:** Cumplió las expectativas.

**Ejemplo de respuesta:**  
{

"error": "Correo ya registrado en el sistema."

}

**Caso 3: Errores de validación.**

* + **Configuración:** Los datos enviados contienen errores, como un correo vacío.
  + **Resultados esperados:**
    - Respuesta con código HTTP 400.
    - Mensaje de error indicando el problema específico.
  + **Resultados obtenidos:** Cumplió las expectativas.

**Ejemplo de respuesta:**  
{

"error": "Debe ingresar un correo electrónico."

}

**Caso 4: Error interno del servidor.**

* + **Configuración:** Error simulado en la base de datos durante la verificación del usuario.
  + **Resultados esperados:**
    - Respuesta con código HTTP 500.
    - Mensaje de error genérico retornado.
    - Registro del error en los logs.
  + **Resultados obtenidos:** Cumplió las expectativas.

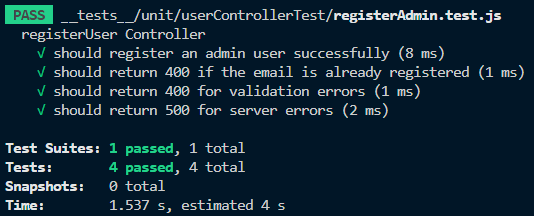
**Ejemplo de respuesta:**  
{

"error": "Error del servidor"

}

#### **Resumen de la Ejecución:**

* **Número total de pruebas:** 4
* **Pruebas exitosas:** 4
* **Pruebas fallidas:** 0
* **Tiempo de ejecución total:** 1.433 segundos



**7.16** **Pruebas unitarias para la función registerUser**

#### **Objetivo:** Verificar el comportamiento del controlador registerUser en diferentes escenarios clave:

1. Registro exitoso de un nuevo usuario.
2. Manejo de errores cuando el correo ya está registrado.
3. Manejo de errores en la validación de datos.
4. Manejo de errores internos del servidor.

#### **Casos de Prueba y Resultados:**

**Caso 1: Registro exitoso.**

* + **Configuración:**
    - No existe un usuario registrado con el correo proporcionado.
    - Los datos del usuario son válidos.
  + **Resultados esperados:**
    - Usuario creado con éxito.
    - Respuesta con código HTTP 201.
    - Mensaje de éxito retornado.
  + **Resultados obtenidos:** Cumplió las expectativas.

**Ejemplo de respuesta:**  
{

"message": "Usuario registrado con éxito"

}

**Caso 2: Correo ya registrado.**

* + **Configuración:**
    - Existe un usuario registrado con el correo proporcionado.
  + **Resultados esperados:**
    - Respuesta con código HTTP 400.
    - Mensaje de error indicando que el correo ya está registrado.
  + **Resultados obtenidos:** Cumplió las expectativas.

**Ejemplo de respuesta:**  
{

"error": "Correo ya registrado en el sistema."

}

**Caso 3: Errores de validación.**

* + **Configuración:**
    - Los datos proporcionados no son válidos (por ejemplo, un correo vacío).
  + **Resultados esperados:**
    - Respuesta con código HTTP 400.
    - Mensaje de error indicando el problema en los datos.
  + **Resultados obtenidos:** Cumplió las expectativas.

**Ejemplo de respuesta:**  
{

"error": "Debe ingresar un correo electrónico."

}

**Caso 4: Error interno del servidor.**

* + **Configuración:**
    - Error simulado en la base de datos durante la verificación del usuario.
  + **Resultados esperados:**
    - Respuesta con código HTTP 500.
    - Mensaje de error genérico retornado.
    - Registro del error en los logs.
  + **Resultados obtenidos:** Cumplió las expectativas.

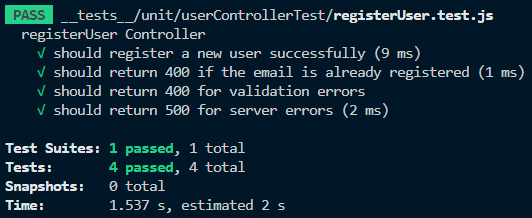
**Ejemplo de respuesta:**  
{

"error": "Error del servidor"

}

#### **Resumen de la Ejecución:**

* **Número total de pruebas:** 4
* **Pruebas exitosas:** 4
* **Pruebas fallidas:** 0
* **Tiempo de ejecución total:** 1.178 segundos



**7.16** **Pruebas unitarias para la función resetPassword**

#### **Objetivo:** Validar el comportamiento la función resetPassword bajo diferentes escenarios, asegurando que maneje adecuadamente los casos de éxito y error:

1. Restablecimiento exitoso de contraseña.
2. Manejo de errores por usuario no encontrado.
3. Manejo de errores por código de recuperación inválido o expirado.
4. Manejo de errores internos del servidor.

#### **Casos de Prueba y Resultados:**

**Caso 1: Restablecimiento exitoso de contraseña.**

* + **Configuración:**
    - El usuario existe en la base de datos.
    - El código de recuperación es válido y no ha expirado.
  + **Resultados esperados:**
    - La contraseña del usuario se actualiza correctamente.
    - Código HTTP 200 retornado con un mensaje de éxito.
  + **Resultados obtenidos:** Cumplió las expectativas.

**Ejemplo de respuesta:**  
{

"message": "Password reset successful."

}

**Caso 2: Usuario no encontrado.**

* + **Configuración:**
    - No se encuentra un usuario con el correo electrónico proporcionado.
  + **Resultados esperados:**
    - Respuesta con código HTTP 400.
    - Mensaje de error indicando que el usuario no fue encontrado.
  + **Resultados obtenidos:** Cumplió las expectativas.

**Ejemplo de respuesta:**  
{

"error": "User not found."

}

**Caso 3: Código de recuperación inválido.**

* + **Configuración:**
    - El código de recuperación proporcionado no coincide con el registrado para el usuario.
  + **Resultados esperados:**
    - Respuesta con código HTTP 400.
    - Mensaje de error indicando código inválido.
  + **Resultados obtenidos:** Cumplió las expectativas.

**Ejemplo de respuesta:**  
{

"error": "Invalid recovery code."

}

**Caso 4: Código de recuperación expirado.**

* + **Configuración:**
    - El código de recuperación proporcionado ha expirado.
  + **Resultados esperados:**
    - Respuesta con código HTTP 400.
    - Mensaje de error indicando expiración del código.
  + **Resultados obtenidos:** Cumplió las expectativas.

**Ejemplo de respuesta:**  
{

"error": "Recovery code has expired."

}

**Caso 5: Error interno del servidor.**

* + **Configuración:**
    - Simulación de un error durante la consulta en la base de datos.
  + **Resultados esperados:**
    - Respuesta con código HTTP 500.
    - Mensaje de error genérico.
  + **Resultados obtenidos:** Cumplió las expectativas.

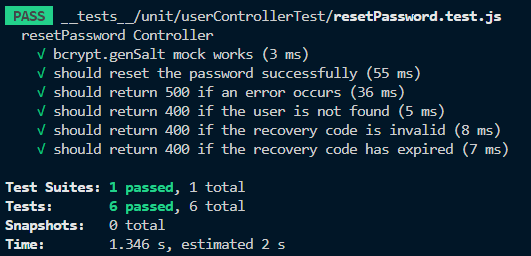
**Ejemplo de respuesta:**  
{

"error": "Server error."

}

#### **Resumen de la Ejecución:**

* **Número total de pruebas:** 6
* **Pruebas exitosas:** 6
* **Pruebas fallidas:** 0
* **Tiempo de ejecución total:** 1.231 segundos



**7.16** **Pruebas unitarias para la función sendRecoveryCode**

#### **Objetivo:** Verificar el comportamiento del controlador sendRecoveryCode bajo distintos escenarios, asegurando que:

1. Se envía correctamente un código de recuperación al correo proporcionado.
2. Maneje adecuadamente casos de errores, como usuario no encontrado o problemas internos del servidor.

#### **Casos de Prueba y Resultados:**

**Caso 1: Usuario no encontrado.**

* + **Configuración:**
    - No existe un usuario asociado al correo proporcionado.
  + **Resultados esperados:**
    - Respuesta con código HTTP 404.
    - Mensaje de error indicando que el usuario no fue encontrado.
  + **Resultados obtenidos:** Cumplió las expectativas.

#### **Ejemplo de respuesta:** {

#### "error": "User not found."

#### }

**Caso 2: Envío exitoso del código de recuperación.**

* + **Configuración:**
    - El usuario existe y el sistema genera un código de recuperación válido.
    - Se envía un correo con el código al usuario.
  + **Resultados esperados:**
    - El código de recuperación y su tiempo de expiración son almacenados en la base de datos.
    - Se envía un correo con el código al usuario.
    - Respuesta con código HTTP 200 y mensaje de éxito.
  + **Resultados obtenidos:** Cumplió las expectativas.

#### **Ejemplo de respuesta:** {

#### "message": "Recovery code sent to email."

#### }

**Caso 3: Error interno del servidor.**

* + **Configuración:**
    - Simulación de un error durante la búsqueda del usuario en la base de datos.
  + **Resultados esperados:**
    - Respuesta con código HTTP 500.
    - Mensaje de error genérico indicando un problema del servidor.
  + **Resultados obtenidos:** Cumplió las expectativas.

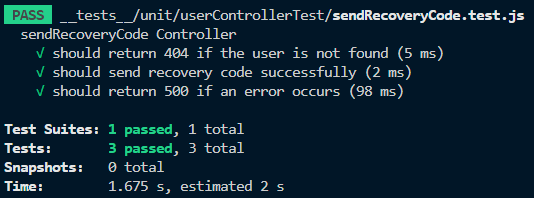
#### **Ejemplo de respuesta:** {

#### "error": "Server error."

#### }

#### **Resumen de la Ejecución:**

* **Número total de pruebas:** 3
* **Pruebas exitosas:** 3
* **Pruebas fallidas:** 0
* **Tiempo de ejecución total:** 1.239 segundos



### **7.16** **Pruebas unitarias para la función updateProfile**

#### **Objetivo:** Evaluar el comportamiento de la función updateProfile para garantizar que:

1. Valide correctamente los datos del perfil proporcionados.
2. Identifique y actualice correctamente el perfil del usuario existente.
3. Cree un nuevo perfil en caso de que no exista.
4. Maneje adecuadamente errores comunes como usuarios inexistentes, correos electrónicos duplicados, y errores internos del servidor.

#### **Casos de Prueba y Resultados:**

**Caso 1: Validación de datos fallida.**

* + **Configuración:**
    - Se omite la fecha de nacimiento en el perfil enviado.
  + **Resultados esperados:**
    - Respuesta con código HTTP 400.
    - Mensaje de error detallando la falta de la fecha de nacimiento.
  + **Resultados obtenidos:** Cumplió las expectativas.

#### **Ejemplo de respuesta:** {

#### "isValid": false,

#### "message": "Debe ingresar su fecha de nacimiento."

#### }

**Caso 2: Usuario no encontrado.**

* + **Configuración:**
    - El usuario especificado no existe en la base de datos.
  + **Resultados esperados:**
    - Respuesta con código HTTP 404.
    - Mensaje de error indicando que el usuario no fue encontrado.
  + **Resultados obtenidos:** Cumplió las expectativas.

**Ejemplo de respuesta:**  
{

"error": "User not found."

}

**Caso 3: Email ya registrado por otra cuenta.**

* + **Configuración:**
    - Otro usuario ya utiliza el correo electrónico proporcionado.
  + **Resultados esperados:**
    - Respuesta con código HTTP 400.
    - Mensaje de error indicando que el correo ya está en uso.
  + **Resultados obtenidos:** Cumplió las expectativas.

**Ejemplo de respuesta:**  
{

"error": "Email is already in use by another account."

}

**Caso 4: Actualización exitosa del perfil existente.**

* + **Configuración:**
    - El usuario y su perfil existen y los datos proporcionados son válidos.
  + **Resultados esperados:**
    - El perfil se actualiza correctamente en la base de datos.
    - Respuesta con código HTTP 200 y mensaje de éxito.
  + **Resultados obtenidos:** **FALLÓ.**
    - El método mockUserProfile.update no fue llamado.

**Ejemplo de respuesta esperada:**  
{

"message": "Profile updated successfully."

}

**Caso 5: Creación de un nuevo perfil cuando no existe.**

* + **Configuración:**
    - El usuario existe, pero no tiene un perfil asociado en la base de datos.
  + **Resultados esperados:**
    - Se crea un nuevo perfil en la base de datos.
    - Respuesta con código HTTP 200 y mensaje de éxito.
  + **Resultados obtenidos:** **FALLÓ.**
    - El método UserProfiles.create no fue llamado.

**Ejemplo de respuesta esperada:**  
{

"message": "Profile updated successfully."

}

**Caso 6: Error interno del servidor.**

* + **Configuración:**
    - Simulación de un error en la consulta a la base de datos.
  + **Resultados esperados:**
    - Respuesta con código HTTP 500.
    - Mensaje de error indicando un problema interno del servidor.
  + **Resultados obtenidos:** Cumplió las expectativas.

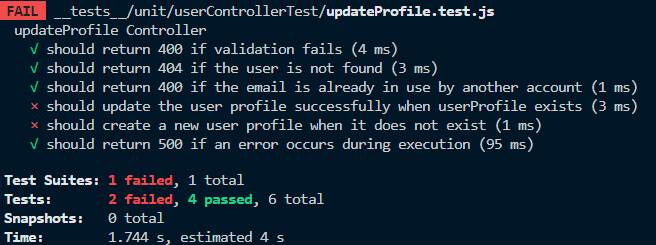
**Ejemplo de respuesta:**  
{

"error": "An error occurred while updating the profile."

}

#### **Resumen de la Ejecución:**

* **Número total de pruebas:** 6
* **Pruebas exitosas:** 4
* **Pruebas fallidas:** 2
* **Tiempo de ejecución total:** 1.258 segundos



### **7.16** **Pruebas unitarias para la función updateUser**

**Objetivo:** Evaluar el comportamiento de la función updateUser para garantizar que:

1. Valide correctamente los datos proporcionados.
2. Identifique y actualice correctamente la información del usuario y su perfil.
3. Cree un perfil de usuario cuando no exista.
4. Maneje adecuadamente errores como usuarios inexistentes, roles inválidos y errores del servidor.

#### **Casos de Prueba y Resultados:**

**Caso 1: Usuario no encontrado.**

* + **Configuración:**
    - El usuario especificado no existe en la base de datos.
  + **Resultados esperados:**
    - Respuesta con código HTTP 404.
    - Mensaje de error indicando que el usuario no fue encontrado.
  + **Resultados obtenidos:** Cumplió las expectativas.

**Ejemplo de respuesta:**  
{

"error": "User not found."

}

**Caso 2: Rol inválido.**

* + **Configuración:**
    - El rol proporcionado no pertenece a la lista de roles válidos.
  + **Resultados esperados:**
    - Respuesta con código HTTP 400.
    - Mensaje de error indicando que el rol es inválido.
  + **Resultados obtenidos:** Cumplió las expectativas.

**Ejemplo de respuesta:**  
{

"error": "Invalid role."

}

**Caso 3: Rol no encontrado.**

* + **Configuración:**
    - El rol especificado no existe en la base de datos.
  + **Resultados esperados:**
    - Respuesta con código HTTP 400.
    - Mensaje de error indicando que el rol no fue encontrado.
  + **Resultados obtenidos:** Cumplió las expectativas.

**Ejemplo de respuesta:**  
{

"error": "Role not found."

}

**Caso 4: Actualización exitosa del usuario.**

* + **Configuración:**
    - El usuario y su rol existen, y los datos proporcionados son válidos.
  + **Resultados esperados:**
    - El usuario se actualiza correctamente en la base de datos.
    - Respuesta con código HTTP 200 y mensaje de éxito.
  + **Resultados obtenidos:** Cumplió las expectativas.

**Ejemplo de respuesta:**  
{

"message": "User updated successfully."

}

**Caso 5: Creación de un perfil de usuario cuando no existe.**

* + **Configuración:**
    - El usuario existe, pero no tiene un perfil asociado en la base de datos.
  + **Resultados esperados:**
    - Se crea un nuevo perfil de usuario en la base de datos.
    - Respuesta con código HTTP 200 y mensaje de éxito.
  + **Resultados obtenidos:** **FALLÓ.**
    - El método UserProfiles.create no fue llamado.

**Ejemplo de respuesta esperada:**  
{

"message": "User updated successfully."

}

**Caso 6: Error interno del servidor.**

* + **Configuración:**
    - Simulación de un error en la consulta a la base de datos.
  + **Resultados esperados:**
    - Respuesta con código HTTP 500.
    - Mensaje de error indicando un problema interno del servidor.
  + **Resultados obtenidos:** Cumplió las expectativas.

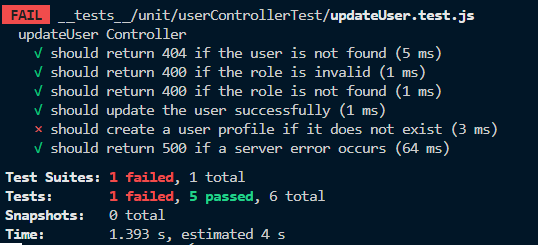
**Ejemplo de respuesta:**  
{

"error": "An error occurred while updating the user."

}

#### **Resumen de la Ejecución:**

* **Número total de pruebas:** 6
* **Pruebas exitosas:** 5
* **Pruebas fallidas:** 1
* **Tiempo de ejecución total:** 1.444 segundos



### **7.17** **Pruebas unitarias para la función verifyRecoveryCode**

#### **Objetivo:** Verificar el correcto funcionamiento de la función verifyRecoveryCode para garantizar que:

1. Valide si el usuario existe.
2. Compruebe si el código de recuperación es válido y no ha expirado.
3. Responda adecuadamente a diferentes escenarios de error y éxito.

#### **Casos de Prueba y Resultados:**

**Caso 1: Usuario no encontrado.**

* + **Configuración:**
    - No existe un usuario asociado con el correo proporcionado.
  + **Resultados esperados:**
    - Respuesta con código HTTP 400.
    - Mensaje de error indicando que el usuario no fue encontrado.
  + **Resultados obtenidos:** Cumplió las expectativas.

**Ejemplo de respuesta:**  
{

"error": "User not found."

}

**Caso 2: Código de recuperación inválido.**

* + **Configuración:**
    - El código de recuperación proporcionado no coincide con el registrado en la base de datos.
  + **Resultados esperados:**
    - Respuesta con código HTTP 400.
    - Mensaje de error indicando que el código de recuperación es inválido.
  + **Resultados obtenidos:** Cumplió las expectativas.

**Ejemplo de respuesta:** {

"error": "Invalid recovery code."

}

**Caso 3: Código de recuperación expirado.**

* + **Configuración:**
    - El código de recuperación ha expirado.
  + **Resultados esperados:**
    - Respuesta con código HTTP 400.
    - Mensaje de error indicando que el código de recuperación ha expirado.
  + **Resultados obtenidos:** Cumplió las expectativas.

**Ejemplo de respuesta:**  
{ "error": "Recovery code has expired." }

**Caso 4: Verificación exitosa del código de recuperación.**

* + **Configuración:**
    - El código de recuperación proporcionado es válido y no ha expirado.
  + **Resultados esperados:**
    - Respuesta con código HTTP 200.
    - Mensaje indicando que el código de recuperación fue verificado exitosamente.
  + **Resultados obtenidos:** Cumplió las expectativas.

**Ejemplo de respuesta:**{ "message": "Recovery code verified. Proceed to reset password."}

**Caso 5: Error interno del servidor.**

* + **Configuración:**
    - Simulación de un error en la consulta a la base de datos.
  + **Resultados esperados:**
    - Respuesta con código HTTP 500.
    - Mensaje de error indicando un problema interno del servidor.
  + **Resultados obtenidos:** Cumplió las expectativas.

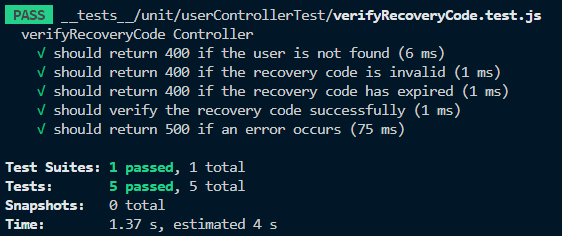
**Ejemplo de respuesta:**  
{

"error": "Server error."

}

#### **Resumen de la Ejecución:**

* **Número total de pruebas:** 5
* **Pruebas exitosas:** 5
* **Pruebas fallidas:** 0
* **Tiempo de ejecución total:** 1.372 segundos



### **7.18** **Pruebas unitarias para el controlador symptom\_classifier\_proba.py**

#### **Objetivo:** Validar el correcto funcionamiento del endpoint /predict\_proba en el servicio Flask symptom\_classifier\_proba, asegurando que:

1. Devuelve una respuesta exitosa para entradas válidas.
2. Maneja adecuadamente casos de errores como falta de datos o excepciones durante la predicción.
3. Valida la interacción con el modelo subyacente mediante mocking.

#### **Casos de Prueba y Resultados:**

1. **Caso 1: Predicción exitosa.**
   * **Configuración:**
     + Entrada: [0.5, 1.2, 3.4].
     + Mock del método predict\_proba del modelo devuelve [[0.7, 0.3]].
   * **Resultados esperados:**
     + Código de estado HTTP 200.
     + Respuesta con las probabilidades escaladas: [[70.0, 30.0]].
     + Validación de que el método predict\_proba fue llamado con los datos correctos.
   * **Resultados obtenidos:** Cumplió las expectativas.
     + **Ejemplo de respuesta:**{ "probabilities": [[70.0, 30.0]] }
2. **Caso 2: Entrada vacía.**
   * **Configuración:**
     + No se envía ningún dato en el cuerpo de la solicitud.
   * **Resultados esperados:**
     + Código de estado HTTP 400.
     + Mensaje de error indicando que no se proporcionaron datos de entrada.
   * **Resultados obtenidos:** Cumplió las expectativas.

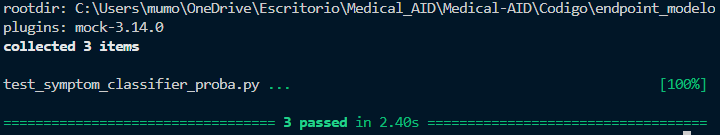
**Ejemplo de respuesta:** { "error": "No input data provided" }

1. **Caso 3: Excepción durante la predicción.**
   * **Configuración:**
     + Entrada: [0.5, 1.2, 3.4].
     + Mock del método predict\_proba lanza una excepción con el mensaje "Mock prediction error".
   * **Resultados esperados:**
     + Código de estado HTTP 500.
     + Mensaje de error indicando el problema.
   * **Resultados obtenidos:** Cumplió las expectativas.

**Ejemplo de respuesta:**{ "error": "Mock prediction error" }

#### **Resumen de la Ejecución:**

* **Número total de pruebas:** 3
* **Pruebas exitosas:** 3
* **Pruebas fallidas:** 0
* **Tiempo de ejecución total:** 8.16 segundos

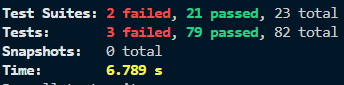


# **Resultados**

1. **Prueba unitarias y de integración**

Se llevaron a cabo un total de 2 pruebas de integración para validar la correcta conexión entre la base de datos con el backend y 80 pruebas unitarias para validar el correcto funcionamiento funciones y controladores. De estas pruebas:

* 79 pruebas fueron exitosas, cumpliendo con los criterios de aceptación establecidos.
* 3 pruebas no pasaron, representando un 2.46% de fallas. Estas se distribuyen de la siguiente manera:
  + 3 pruebas de severidad baja, relacionadas con aspectos no críticos que no afectan directamente el uso o la funcionalidad principal de la aplicación.



1. **Prueba funcionales**

El detalle y resultado de las pruebas funcionales se encuentra en el siguiente documento: [**Pruebas Funcionales.docx**](https://docs.google.com/document/d/1oxDVPo3UZ5_3nlHmcGw4algEZ3SnusQV/edit?usp=sharing&ouid=111519466906793012124&rtpof=true&sd=true)

# **Conclusión**

El plan de pruebas incluyó pruebas funcionales, unitarias e integración, con el objetivo de garantizar el correcto funcionamiento del sistema y su interacción entre componentes. A continuación, se resumen los resultados obtenidos:

1. **Pruebas Funcionales:**
   * Se ejecutaron un total de 99 pruebas, de las cuales:
     + 94 pruebas fueron exitosas (94.95% de éxito).
     + 5 pruebas no pasaron (5.05% de fallas), distribuidas de la siguiente manera:
       - 4 fallas de severidad baja.
       - 1 falla de severidad media.
2. **Pruebas Unitarias:**
   * Se llevaron a cabo 80 pruebas unitarias, enfocadas en validar las funciones y controladores:
     + 77 pruebas exitosas (96.25% de éxito).
     + 3 pruebas no pasaron (3.75% de fallas), todas clasificadas como de severidad baja.
3. **Pruebas de Integración:**
   * Se realizaron 2 pruebas de integración para verificar la conexión entre el backend y la base de datos:
     + Ambas pruebas fueron exitosas, garantizando la correcta integración entre la base de datos con el backend.

### **Análisis General**

El 95.58% de las pruebas funcionales, unitarias y de integración fue exitoso, destacando que los componentes principales del sistema operan correctamente. Aunque 4.42% del total de las pruebas presentaron fallas, estas se clasificaron mayoritariamente como de severidad baja, relacionadas con aspectos no críticos que no afectan directamente la funcionalidad central del sistema. Además, no se identificaron fallas críticas o bloqueantes.