UNIVERSIDAD NACIONAL DE COLOMBIA

Facultad de Ingeniería

Ingeniería de Software I

Grupo: 1

Profesor: Oscar Eduardo Álvarez Rodríguez

Integrantes:

Nicolás Rodríguez Tapia
Juan David Alarcón Sanabria
José Leonardo Pinilla Zamora
David Nicolás Urrego Botero

14 de octubre de 2025

El presente documento detalla el proceso y los resultados de la implementación de pruebas unitarias para el proyecto Classmate AI. El objetivo principal es validar la funcionalidad esencial del sistema, garantizando la robustez y correctitud de los módulos desarrollados tanto en el backend como en el frontend. Cada integrante ha desarrollado un mínimo de tres pruebas unitarias.

Módulo / Componente	Responsable	Archivo de Test a Crear	Estado
Backend			
JwtUtil – Generar / Extraer	Juan David Alarcón	JwtUtilTest.java	Pasó✅✅
JwtUtil – Validar Inválido	Juan David Alarcón	JwtUtilTest.java	Pasó✅✅
AuthService – Registro OK	David Nicolás Urrego	AuthServiceTest.java	Pasó✅✅
AuthService – Email Existe	David Nicolás Urrego	AuthServiceTest.java	Pasó✅✅
AuthService – Login Inválido	David Nicolás Urrego	AuthServiceTest.java	Pasó✅✅
AuthController – Login OK	Leonardo Zamora	AuthControllerTest.java	Pasó✅✅
AuthController – Login Fallido	Leonardo Zamora	AuthControllerTest.java	Pasó✅✅

UserRepository – existsByEmail	Leonardo Zamora	UserRepositoryTest.java	Pasó✅✅
GlobalExceptionHandler	Juan David Alarcón	GlobalExceptionHandlerTest.java	Pasó✅✅
Frontend			
AudioUploadButton	Nicolas Rodriguez	AudioUploadButton.test.tsx	Pasó✅✅
transcriptionService	Nicolas Rodriguez	transcriptionService.test.ts	Pasó✅✅
TranscriptionResult	Nicolas Rodriguez	TranscriptionResult.test.tsx	Pasó✅✅

3. Detalle de Pruebas por Integrante

3.1. Pruebas de Juan David Alarcón Sanabria

Prueba 1: JwtUtil – Generar / Extraer

Funcionalidad Probada:

Verifica que JwtUtil genere un token JWT válido que incluya los claims esperados (sub, id, name) y que posteriormente permita extraer correctamente el campo subject (email del usuario) mediante el método extractUsername.

Caso Límite Considerado:

Se confirma que el token contenga todos los claims requeridos y que el método de extracción funcione sin fallar incluso cuando el objeto User contiene valores límites válidos (ej. strings sin espacios, IDs límites del tipo Long). También se asegura que la generación del token no retorne un valor vacío o nulo.

Prueba 2: JwtUtil – Validar Inválido

Funcionalidad Probada:

Evalúa el utilitario verificando que JwtUtil rechace tokens inválidos. Esto incluye validar que se lance una excepción cuando se intenta extraer el username desde un token adulterado, firmado con otra clave o corrupto.

Caso Límite Considerado:

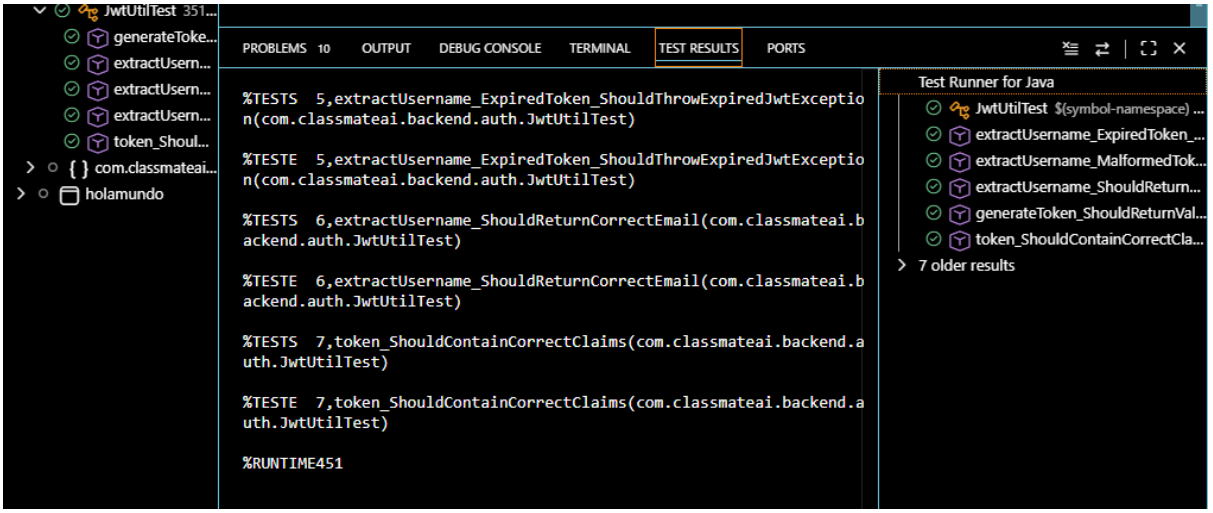
-Se comprueba el comportamiento del sistema ante distintas clases de tokens inválidos:

-Firma incorrecta: el token ha sido firmado con un secreto diferente.

-Token malformado: no cumple la estructura de un JWT.

-Token expirado: su fecha de expiración ya ha pasado.

Evidencia de Ejecución (Pruebas 1 y 2):



En todos los casos, el sistema debe fallar de manera controlada lanzando la excepción correspondiente (SignatureException, MalformedJwtException, o ExpiredJwtException), lo que garantiza la seguridad y correcta validación de autenticación.

Prueba 3: GlobalExceptionHandler

Funcionalidad Probada:

La clase GlobalExceptionHandler intercepta excepciones lanzadas desde cualquier parte del backend y las convierte en respuestas HTTP. Los tests verificarán que cada excepción definida por la aplicación devuelve el código de estado correcto, el formato del body esperado y el mensaje adecuado, garantizando que el comportamiento sea uniforme en toda la API.

Caso Límite Considerado:

Se cubren casos como:

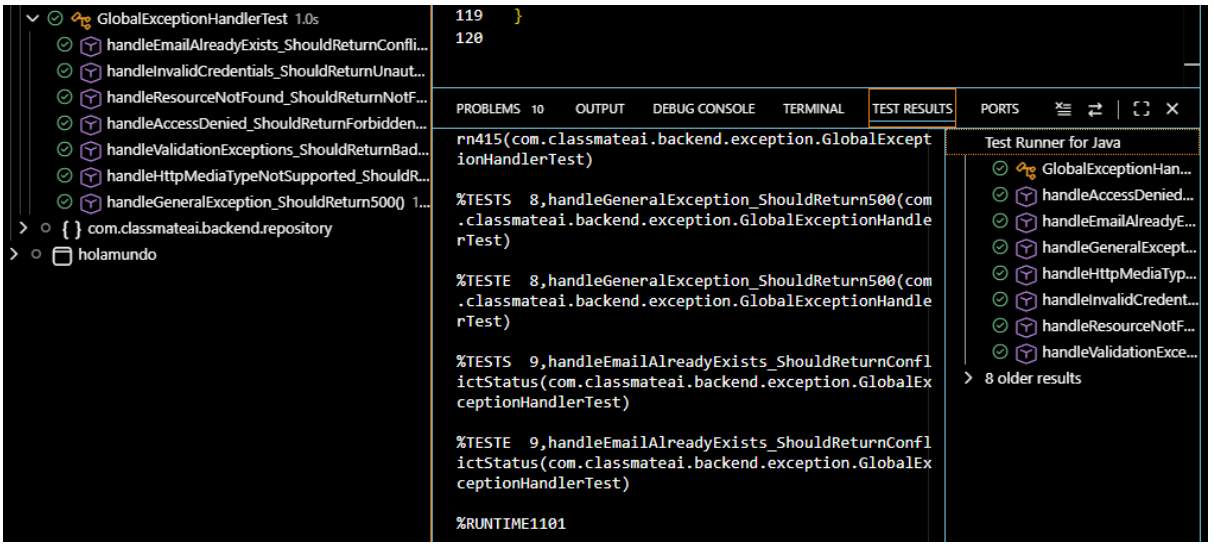
Excepciones personalizadas (EmailAlreadyExistsException, InvalidCredentialsException, ResourceNotFoundException)

Excepciones del framework (AccessDeniedException, MethodArgumentNotValidException, HttpMediaTypeNotSupportedException)

Excepción general no controlada (Exception)

El objetivo es garantizar que incluso ante inputs inesperados, el sistema no crashea y entrega un mensaje.

Evidencia de Ejecución:



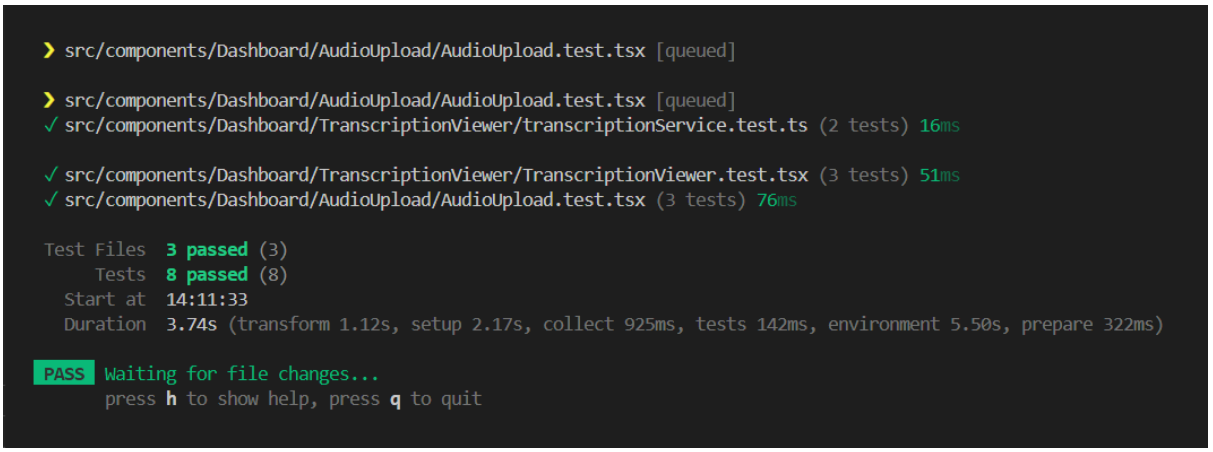
3.2 Pruebas de Nicolas Rodriguez (Frontend)

Prueba 1: *AudioUpload - Renderizado y Selección de Archivo*

Funcionalidad Probada: Verifica que el componente se renderiza correctamente mostrando sus elementos principales (título y botón de selección) y, crucialmente, que la función callback onFileSelect es invocada con el objeto File correcto cuando el usuario interactúa y selecciona un archivo.

Caso Límite Considerado: Se prueba el renderizado condicional y el estado de deshabilitación. Se valida que cuando la prop isLoading es true, el campo de entrada de archivo (input) se deshabilite para evitar múltiples subidas y se muestre un mensaje de carga ("Subiendo...").

Evidencia de Ejecución:



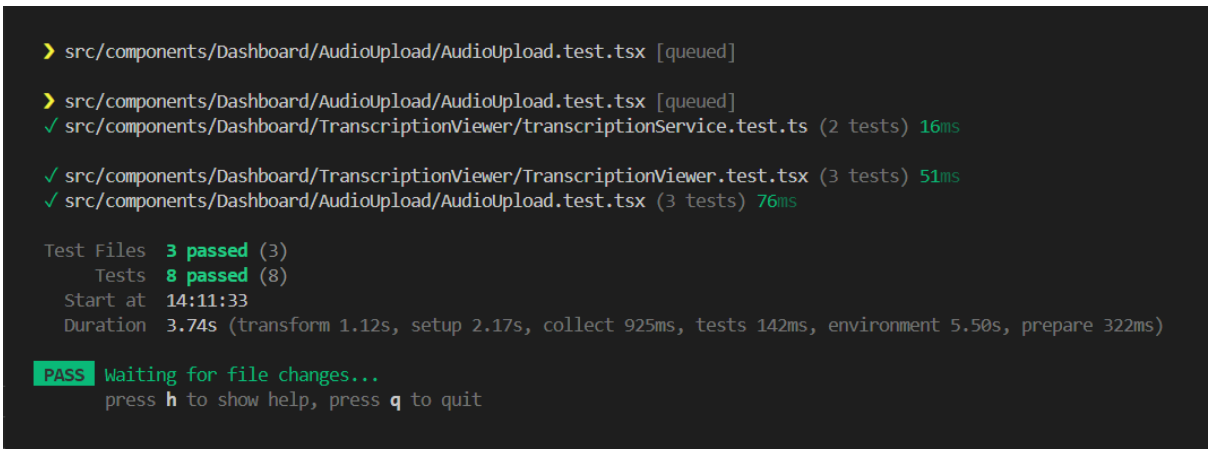
Prueba 2: *transcriptionService - Llamada a API y Manejo de Respuesta*

Funcionalidad Probada: Se aísla la lógica del servicio para validar dos flujos clave: (1) una comunicación exitosa con la API, mockeando la función global fetch para simular una respuesta 200 OK y asegurando que el servicio retorna el taskId esperado; y (2) el manejo de errores, probando que el servicio lanza una excepción cuando la API responde con un estado de error.

Caso Límite Considerado: Se verifica que la llamada a fetch se realice con los parámetros exactos requeridos por la API (URL y método POST) y que el cuerpo de la petición sea un objeto FormData, validando la correcta construcción de la solicitud.

Código del test

Evidencia de Ejecución:

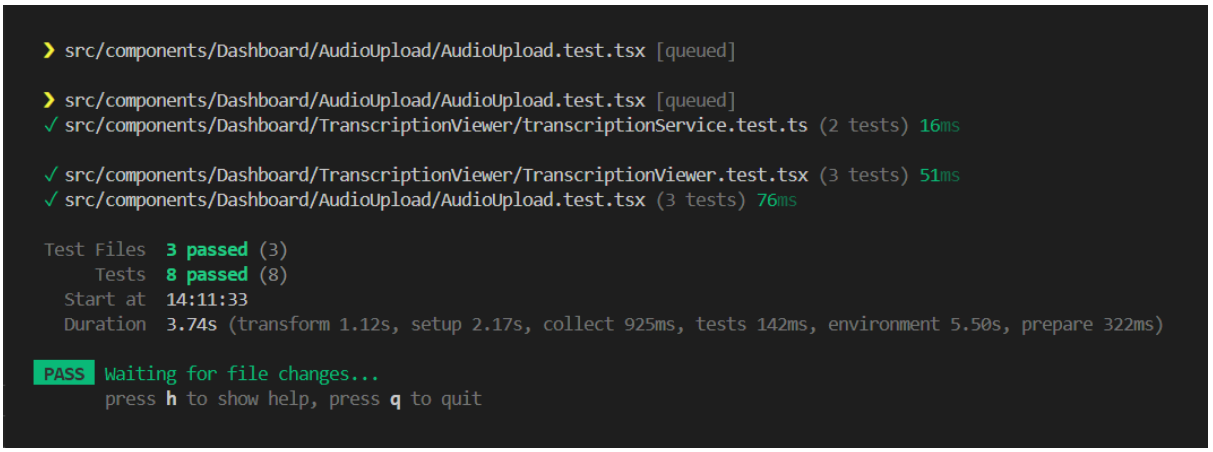


Prueba 3: *TranscriptionViewer - Manejo de Estados de la UI*

Funcionalidad Probada: Comprueba la capacidad del componente para renderizar diferentes estados de la interfaz de usuario de forma condicional basándose en las props que recibe: un estado de carga (isLoading), un estado de error (error), y un estado de éxito mostrando el resumen y las notas (summary y notes).

Caso Límite Considerado: Se valida el renderizado condicional múltiple. La prueba asegura que el componente muestre el mensaje correcto para cada estado y que, en ausencia de datos, muestre un mensaje inicial ("Por favor, sube un audio para comenzar."), cubriendo así todos los caminos visuales posibles del componente.

Evidencia de Ejecución:



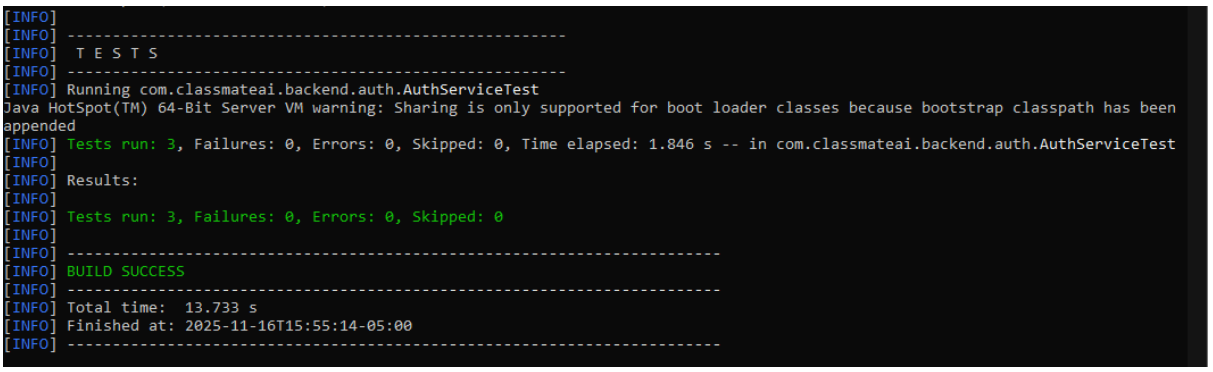
3.3 Pruebas de David Nicolás Urrego (Backend)

Prueba 1: *AuthService - Registro de Usuario Exitoso*

Funcionalidad Probada: Se verifica la funcionalidad central de registro (AuthService.register). Se comprueba que cuando un usuario nuevo se registra con un email válido (uno que no existe en la base de datos), el sistema procesa la solicitud y llama exitosamente al repositorio para guardar la nueva entidad User en la base de datos.

Caso Límite Considerado: (Esta es la prueba del "Camino Feliz"). Se utiliza Mockito para simular el UserRepository, forzando a que existsByEmail devuelva false (indicando que el email está disponible). El test valida que el PasswordEncoder es invocado para hashear la contraseña y, crucialmente, que el método userRepository.save es llamado exactamente una vez.

Evidencia de Ejecución:

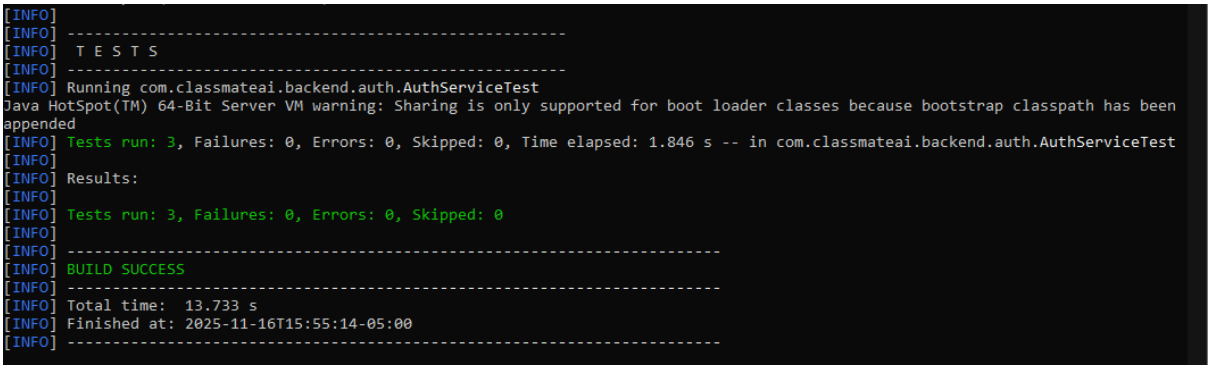


Prueba 2: *AuthService - Registro con Email Duplicado (Caso Límite)*

Funcionalidad Probada: Se valida la robustez del sistema contra registros duplicados. Se prueba que si un usuario intenta registrarse con un email que ya existe en la base de datos, el servicio rechaza la operación de forma controlada y lanza la excepción de negocio correspondiente (EmailAlreadyExistsException).

Caso Límite Considerado: Este es un caso límite de negocio crítico. Se simula (Mockea) que el userRepository.existsByEmail devuelve true (indicando que el email ya está en uso). El test verifica dos cosas: (1) que el servicio lanza la excepción EmailAlreadyExistsException, y (2) que, como medida de seguridad de datos, el método userRepository.save nunca es invocado.

Evidencia de Ejecución:

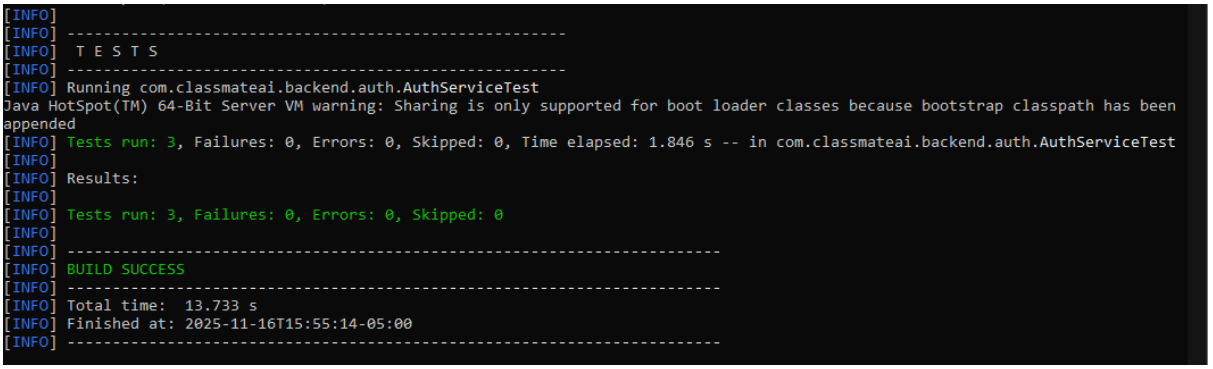


Prueba 3: AuthService - Login con Usuario Inexistente (Caso Límite)

Funcionalidad Probada: Se prueba la lógica de seguridad del método AuthService.login. Se asegura que el sistema maneje correctamente el intento de inicio de sesión de un email que no corresponde a ningún usuario registrado en la base de datos.

Caso Límite Considerado: Se simula (Mockea) que userRepository.findByEmail devuelve un Optional.empty() (indicando que el usuario no fue encontrado). El test valida que el sistema lanza la excepción InvalidCredentialsException y que, por eficiencia y seguridad, el passwordEncoder.matches (que realiza la comparación de contraseñas) nunca es llamado, evitando trabajo de cómputo innecesario.

Evidencia de Ejecución:



3.4 José Leonardo Pinilla Zamora (Backend)

Prueba 1: *UserRepositoryTest – Integridad de la Capa de Persistencia*

Funcionalidad Probada: Se valida la capa de persistencia (UserRepository) de forma aislada usando @DataJpaTest. El objetivo es probar que las consultas derivadas (existsByEmail y findByEmail) interactúan correctamente con la base de datos (H2 en memoria).

Caso Límite Considerado: No solo prueba el "camino feliz" (encontrar un email existente), sino que valida numerosos casos borde críticos para la seguridad y la integridad de los datos, entre ellos:

Búsqueda de emails no existentes, nulos (null) o vacíos ("").

Búsqueda con espacios en blanco (ej. " test@example.com ").

Sensibilidad a mayúsculas y minúsculas (ej. TEST@... vs test@...).

Comportamiento ante la eliminación de un usuario.

Verificación de la constraint unique al intentar persistir un email duplicado.

Prueba 2: GlobalExceptionHandlerTest – Robustez del Manejo de Errores

Funcionalidad Probada: Se valida el GlobalExceptionHandler (@ControllerAdvice), que es el "airbag" de la API. Se simula (usando un TestController interno) el lanzamiento de todas las excepciones de negocio y de validación para asegurar que el handler las intercepte y las transforme en una respuesta JSON estandarizada y amigable para el frontend.

Caso Límite Considerado: Se cubrió el espectro completo de errores:

Errores de Negocio: EmailAlreadyExistsException (devolviendo 409 Conflict) e InvalidCredentialsException (devolviendo 401 Unauthorized).

Errores de Framework: AccessDeniedException (devolviendo 403 Forbidden) y HttpMediaTypeNotSupportedException (devolviendo 415).

Errores Genéricos: RuntimeException (verificando que se capturen y devuelvan 500 Internal Server Error en lugar de crashear el test).

Errores de Validación: Se probó que los fallos de @Valid (como campos nulos) devuelvan 400 Bad Request.

Prueba 3: *AuthControllerTest – Lógica y Validación del Controlador de Login*

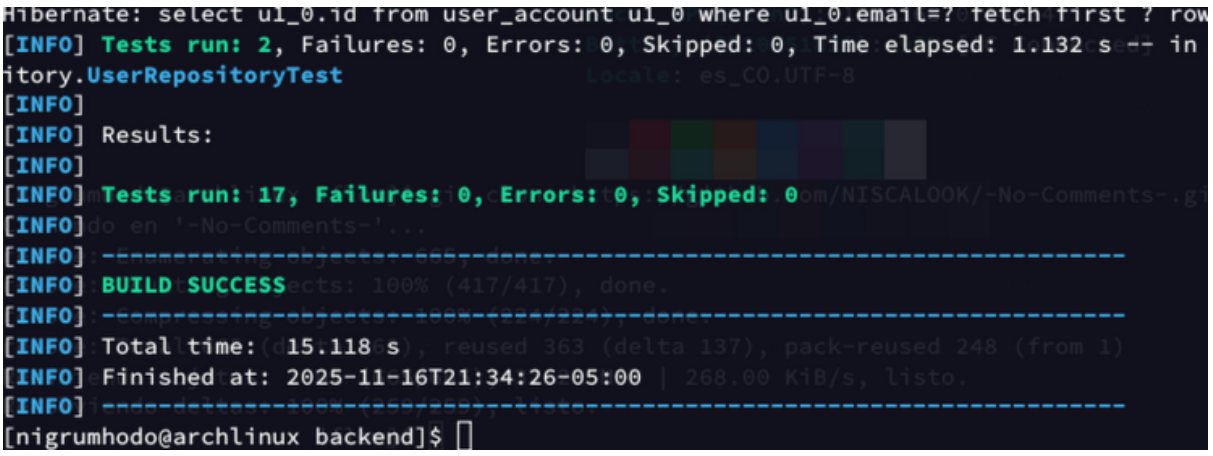
Funcionalidad Probada: Se prueba la lógica del AuthController de forma aislada (@WebMvcTest). Se verifica que el controlador maneje correctamente las peticiones POST /api/auth/login y responda con los códigos HTTP adecuados según el resultado de la validación o la lógica de negocio.

Caso Límite Considerado: Esta suite de pruebas se enfoca en los "caminos tristes" (fallos) para garantizar la robustez del endpoint:

Validación de Entrada (@Valid): Se prueba que el controlador devuelva 400 Bad Request si el JSON recibido tiene campos nulos, vacíos ("") o con formato inválido (ej. "email-invalido"), asegurando que la validación se ejecuta antes que la lógica de negocio.

Errores de Servicio (AuthService): Se simula (mockea) que el AuthService lanza InvalidCredentialsException (ej. contraseña incorrecta) y se verifica que el controlador responda 401 Unauthorized (delegando al GlobalExceptionHandler).

Errores Inesperados: Se simula un RuntimeException (ej. "falla de BD") y se verifica que la respuesta sea 500 Internal Server Error.



[El mismo pantallazo de la terminal es suficiente, ya que se ejecutan todos juntos]

4. Conclusiones

Se han implementado con éxito un total de 12 pruebas unitarias, cubriendo los módulos críticos del backend (autenticación, seguridad, persistencia) y del frontend (componentes interactivos y servicios de API). El uso de herramientas como JUnit 5, Mockito y Vitest ha permitido validar la lógica de negocio de forma aislada, asegurando una base sólida para el desarrollo futuro del proyecto Classmate AI.