TESTING

Juan David Cruz Giraldo
Juan Sebastián Gámez Ariza
Juan Camilo López León
Alejandro Ramírez Núñez

Universidad Nacional de Colombia
Facultad de Ingeniería
Ingeniería de software I

Testing de product.service.ts: ***Juan Sebastian Gamez Ariza***

A continuación, se describen las pruebas unitarias implementadas para el servicio de productos, el cual encapsula la lógica de negocio relacionada con la gestión de productos. Para poder evaluar aisladamente el comportamiento del servicio, se creó un mock del repositorio mediante Jest, simulando la interfaz ProductRepositoryInterface.

```
let productService: ProductService;
let mockRepository: MockProductRepository;

// Datos de prueba
const mockProduct: ProductDTO = {
  id: 1,
  name: 'Aspirin',
  type: 'Analgesic',
  use_case: 'Pain relief',
  warnings: 'May cause stomach upset',
  contraindications: 'Allergic to salicylates',
  expiration_date: '2025-12-31',
  price: 5.99,
  stock: 100,
  created_at: new Date('2024-01-01'),
};

const mockCreateProductData: CreateProductDTO = {
  name: 'Ibuprofen',
  type: 'NSAID',
  use_case: 'Inflammation and pain relief',
  warnings: 'May cause GI upset',
  contraindications: 'Kidney disease',
  expiration_date: '2025-11-30',
  price: 7.99,
  stock: 50,
};

beforeEach(() => {
  mockRepository = new MockProductRepository();
  productService = new ProductService(mockRepository);
  jest.clearAllMocks();
});
```

1. Pruebas del método getProduct(id)

Este método es responsable de recuperar un producto por su ID. Se evaluaron tres comportamientos esenciales:

Caso 1 — Producto no encontrado

Objetivo: Verificar que el servicio arroje una excepción cuando el repositorio no encuentra un producto.

- El mock del repositorio retorna null.
- Se espera que el servicio lance el error:"Product with ID X not found"

Caso límite: ID inexistente.

Caso 2 — Producto encontrado exitosamente

Objetivo: Validar que el servicio retorne correctamente el producto cuando existe en la base de datos.

- El mock retorna un objeto Producto válido.
- Se verifica que el valor devuelto sea exactamente el mismo.

Caso típico: Lectura correcta desde la capa de datos.

Caso 3 — Error interno del repositorio

Objetivo: Comprobar que el servicio propaga errores inesperados provenientes del repositorio.

- El mock lanza un error como: "Database connection failed"
- Se espera que el servicio relance el mismo error.

Caso límite: Fallas de infraestructura o capa de persistencia.

```
describe('getProduct', () => {
  it('should throw error when product is not found', async () => {
    mockRepository.findById.mockResolvedValue(null);

    await expect(productService.getProduct(999)).rejects.toThrow(
      'Product with ID 999 not found'
    );
  });

  it('should return product when found', async () => {
    mockRepository.findById.mockResolvedValue(mockProduct);

    const result = await productService.getProduct(1);

    expect(result).toEqual(mockProduct);
  });

  it('should catch and re-throw repository errors', async () => {
    const repositoryError = new Error('Database connection failed');
    mockRepository.findById.mockRejectedValue(repositoryError);

    await expect(productService.getProduct(1)).rejects.toThrow(
      'Database connection failed'
    );
  });
});
```

2. Pruebas del método getAllProducts()

Este método obtiene todos los productos registrados.

Caso 1 — Lista de productos obtenida exitosamente

Objetivo: Confirmar que el servicio retorne la lista de productos recibida desde el repositorio.

- El mock retorna un arreglo con productos.
- El servicio debe retornar exactamente ese arreglo.

Caso 2 — Error del repositorio

Objetivo: Validar el manejo adecuado de errores al procesar la consulta.

- El mock lanza un error de base de datos.
- Se espera que el servicio relance la excepción.

```
describe('getAllProducts', () => {
  it('should return array of products from repository', async () => {
    const products = [mockProduct, { ...mockProduct, id: 2, name: 'Ibuprofen' }];
    mockRepository.findAll.mockResolvedValue(products);

    const result = await productService.getAllProducts();

    expect(result).toEqual(products);
  });

  it('should catch and re-throw repository errors', async () => {
    const repositoryError = new Error('Database query failed');
    mockRepository.findAll.mockRejectedValue(repositoryError);

    await expect(productService.getAllProducts()).rejects.toThrow(
      'Database query failed'
    );
  });
});
```

3. Pruebas del método createProduct(data)

Este método crea un nuevo producto a partir de los datos recibidos.

Caso 1 — Creación exitosa

Objetivo: Verificar que el servicio retorne el producto creado según lo devuelto por el repositorio.

- El mock retorna un Producto con ID y fecha de creación.
- Se valida que el servicio retorne el mismo objeto.

Caso típico: Proceso normal de registro.

Caso 2 — Error durante la creación

Objetivo: Evaluar que el servicio maneje correctamente errores como duplicidad de nombres u otros problemas de validación.

- El mock lanza un error como: "Duplicate product name"
- Se espera que el servicio relance la excepción.

Caso límite: Violación de restricciones de negocio.

```
describe('createProduct', () => {
  it('should return created product from repository', async () => {
    const createdProduct: ProductDTO = {
      ...mockCreateProductData,
      id: 1,
      created_at: new Date('2024-11-16'),
    };
    mockRepository.create.mockResolvedValue(createdProduct);

    const result = await productService.createProduct(mockCreateProductData);

    expect(result).toEqual(createdProduct);
  });

  it('should catch and re-throw repository errors', async () => {
    const repositoryError = new Error('Duplicate product name');
    mockRepository.create.mockRejectedValue(repositoryError);

    await expect(
      productService.createProduct(mockCreateProductData)
    ).rejects.toThrow('Duplicate product name');
  });
});
```

Testing de user.service.ts: ***Juan David Cruz Giraldo***

A continuación, se describen las pruebas unitarias implementadas para el servicio de usuarios (UserService), el cual encapsula la lógica de negocio central de la aplicación, como el registro (createUser), la autenticación (login) y la consulta de usuarios (getUser).

Para poder evaluar aisladamente el comportamiento del servicio, se aplicó mocking a todas sus dependencias externas clave utilizando Jest. Esto incluye:

- La simulación de la interfaz UserRepositoryInterface para aislar la base de datos.
- El mock de la librería bcrypt para simular el hasheo y la comparación de contraseñas de forma instantánea.
- El mock de la librería crypto para controlar la generación de tokens de verificación.

```
import { UserService } from "./user.service";
import { UserRepositoryInterface } from "../repository/user.repository";
import bcrypt from 'bcrypt';
import userDTO from '../dto/user.dto';
import { ServiceUserDTO } from "../dto/user.dto";
import crypto from 'crypto';
import { sendVerificationEmail } from "./email.service";
import { create } from "domain";

jest.mock('./email.service', () => ({
  __esModule: true, // Necesario para ES Modules
  // Creamos una función mock falsa para 'sendVerificationEmail'
  sendVerificationEmail: jest.fn(),
}));


//Remplazamos las dependencias por mocks
jest.mock('../repository/user.repository');
jest.mock('bcrypt');
jest.mock('crypto');


//Para no tener problemas con los tipos :)
const mockedBcrypt = bcrypt as jest.Mocked<typeof bcrypt>;
const mockedCrypto = crypto as jest.Mocked<typeof crypto>;
const mockedSendVerificationEmail = sendVerificationEmail as jest.Mock;
const MockedUserRepository = {
  findByEmailForLogin: jest.fn(),
  findByEmail: jest.fn(),
  create: jest.fn(),
  findById: jest.fn(),
} as any;
```

1. Pruebas del método login

Este método es responsable de autenticar a un usuario. Se evaluaron tres comportamientos esenciales:

Caso 1 — Login Exitoso

- Objetivo: Validar que el servicio autentica y retorna un DTO de usuario cuando las credenciales son correctas y el usuario está verificado.
- Configuración (Mocks):
    - El userRepository.findByEmailForLogin retorna un usuario completo (con hash) y is_verified: true.
    - bcrypt.compare retorna true.
    - El userRepository.findByEmail retorna el DTO final (sin hash).
- Resultado Esperado: El servicio retorna el userDTO del usuario.
- Tipo: Caso esencial.

```
test('should login successfully with correct credentials', async () => {

    //Usuario existe y esta verificado con contraseña correcta
    const email='jest@gmail.com';
    const password="Correctpassword_1"

    //Mockeamos respuesta del repositorio
    const mockuser = {
        id: 1,
        username: "UserJest",
        email: email,
        hash_password: "hashedpassword",
        role: "user",
        verification_token: "some_token",
        token_expiry_at: new Date(Date.now()),
        is_verified: true
    }

    MockedUserRepository.findByEmailForLogin.mockResolvedValue(mockuser);

    //Simulamos que la contraseña coincide
    mockedBcrypt.compare.mockResolvedValue(true as never);

    //Mockeamos findByEmail para retornar el userDTO sin el hash_password
    const userDTOResult: userDTO = {
        id: 1,
        username: "UserJest",
        email: email,
        role: "user",
        created_at: new Date(Date.now()),
    };

    MockedUserRepository.findByEmail.mockResolvedValue(userDTOResult);

    const result = await userService.login(email, password);

    expect(result).toEqual(userDTOResult);
```

Caso 2 — Contraseña Inválida

- Objetivo: Verificar que el servicio arroja una excepción de seguridad si la contraseña es incorrecta.
- Configuración (Mocks):
- El userRepository.findByEmailForLogin retorna un usuario válido.
- bcrypt.compare retorna false.
- Resultado Esperado: El servicio lanza el error: "Invalid password".
- Tipo: Caso de fallo / Seguridad.

```
test('should throw an error for invalid password', async () => {
    //Usuario existe y esta verificado pero la contraseña es incorrecta

    const email='jest@gmail.com';
    const password="Wrongpassword_1"

    //Mockeamos respuesta del repositorio
    const mockuser = {
        username: "UserJest",
        email: email,
        hash_password: "hashedpassword",
        role: "user",
        verification_token: "some_token",
        token_expiry_at: new Date(Date.now()),
        is_verified: true
    }

    MockedUserRepository.findByEmailForLogin.mockResolvedValue(mockuser);

    //Simulamos que la contraseña no coincide
    mockedBcrypt.compare.mockResolvedValue(false as never);

    await expect(userService.login(email, password)).rejects.toThrow('Invalid password');

    //Verificamos que se hayan llamado los mocks
    expect(MockedUserRepository.findByEmailForLogin).toHaveBeenCalledWith(email);
    expect(mockedBcrypt.compare).toHaveBeenCalledWith(password, mockuser.hash_password);
})

test('should login successfully with correct credentials', async () => {
```

Caso 3 — Usuario No Verificado

- Objetivo: Verificar que un usuario que no ha confirmado su email no puede iniciar sesión.
- Configuración (Mocks):
- El userRepository.findByEmailForLogin retorna un usuario con is_verified: false.
- Resultado Esperado: El servicio lanza el error: "User email is not verified".
- Tipo: Caso límite / Seguridad.

```
test('should throw an error if user email is not verified', async () => {

    const email="jest@gmail.com";
    const password="Somepassword_1";

    //Mockeamos respuesta del repositorio
    const mockuser = {
        username: "UserJest",
        email: email,
        hash_password: "hashedpassword",
        role: "user",
        verification_token: "some_token",
        token_expiry_at: new Date(Date.now()),
        is_verified: false
    }

    MockedUserRepository.findByEmailForLogin.mockResolvedValue(mockuser);

    await expect(userService.login(email, password)).rejects.toThrow('User email is not verified')

    //Verificamos que se hayan llamado los mocks
    expect(MockedUserRepository.findByEmailForLogin).toHaveBeenCalledWith(email);
```

2. Pruebas del método createUser

Este método es responsable de registrar un nuevo usuario, validar sus datos y orquestar las dependencias (hash, token, email).

Caso 1 — Creación Exitosa

- Objetivo: Validar la orquestación completa: el servicio hashea la contraseña, genera un token, guarda en el repositorio y envía un email de verificación.
- Configuración (Mocks):
    - bcrypt.hash retorna un hash simulado.
    - crypto.randomBytes retorna un token simulado.
    - userRepository.create retorna el DTO del nuevo usuario.
    - sendVerificationEmail se resuelve exitosamente (undefined).
- Resultado Esperado: El servicio retorna el userDTO creado y se verifica que todos los mocks (bcrypt, crypto, repositorio, email) fueron llamados correctamente.

- Tipo: Caso esencial.

```
test('Should create user successfully with valid data', async () => {

    const email = "test@example.com";
    const username = "TestUser";
    const password = "Validpassword_1";
    const role = "user";

    const mockCreateUserDTO = {
        username: username,
        email: email,
        password: password,
        role: role
    };

    //Mockeamos bcrypt.hash
    const hashedPassword = "hashedValidpassword_1";
    mockedBcrypt.hash.mockResolvedValue(hashedPassword as never);

    //Mockeamos crypto.randomBytes para el token de verificación
    const mockToken = 'mi-token-fijo-de-prueba';
    // Forzamos el tipo a 'jest.Mock' para evitar el error de sobrecarga
    (mockedCrypto.randomBytes as jest.Mock).mockReturnValue({
        toString: (encoding: string) => {
            // Tu código llama a .toString('hex'), así que lo simulamos
            if (encoding === 'hex') {
                return mockToken;
            }
            return 'token-default';
        }
    });

    //Mockeamos la creación del usuario en el repositorio
    const mockCreatedUser: userDTO = {
        id: 1,
        username: username,
        email: email,
        role: role,
        created_at: new Date(Date.now()),
    };

    MockedUserRepository.create.mockResolvedValue(mockCreatedUser);

    //Mockear envio de correo
    mockedSendVerificationEmail.mockResolvedValue(undefined);

    const result = await userService.createUser(mockCreateUserDTO);
```

Caso 2 — Fallos de Validación de Entrada

- Objetivo: Verificar que el servicio rechaza datos de entrada que no cumplen con las reglas de negocio, antes de interactuar con dependencias externas.
- Pruebas cubiertas:
- Should throw error if email format is invalid: Se prueba con un email mal formateado.

```
test('Should throw error if email format is invalid', async () => {

    const invalidEmail = "invalidemailformat";
    const username = "TestUser";
    const password = "Validpassword_1";
    const role = "user";

    const mockCreateUserDTO = {
        username: username,
        email: invalidEmail,
        password: password,
        role: role
    };

    await expect(userService.createUser(mockCreateUserDTO)).rejects.toThrow('Invalid email format')
});
```

- Should throw error if password format is invalid: Se prueba con una contraseña débil.

```
test('Should throw error if password format is invalid', async () => {

    const email = "test@example.com";
    const username = "TestUser";
    const invalidPassword = "short";
    const role = "user";

    const mockCreateUserDTO = {
        username: username,
        email: email,
        password: invalidPassword,
        role: role
    };

    await expect(userService.createUser(mockCreateUserDTO)).rejects.toThrow('Password must be 8-64 characters and include upperc
});
```

- Should throw error if role is invalid: Se prueba con un rol no permitido.

```
test('Should throw error if role is invalid', async () => {

    const email = "test@example.com";
    const username = "TestUser";
    const password = "Validpassword_1";
    const invalidRole = "invalidRole";

    const mockCreateUserDTO = {
        username: username,
        email: email,
        password: password,
        role: invalidRole
    };

    await expect(userService.createUser(mockCreateUserDTO)).rejects.toThrow('Invalid role. Allowed roles are: admin, user');
});
```

- **Configuración (Mocks):** No se requieren mocks, ya que el error ocurre antes de llamarlos.
- **Resultado Esperado:** El servicio lanza la excepción correspondiente a cada validación ("Invalid email format", "Password must be 8-64 characters...", etc.).
- **Tipo:** Caso de fallo / Validación.

3. Pruebas del método getUser

Este método es responsable de recuperar un producto por su ID.

Caso 1 — Producto Encontrado Exitosamente

- Objetivo: Validar que el servicio retorna correctamente el DTO del producto cuando este existe.
- Configuración (Mocks):
- El userRepository.findById retorna un userDTO simulado.
- Resultado Esperado: El servicio retorna el userDTO proporcionado por el repositorio.
- Tipo: Caso esencial.

```
test('should return user data if user is found', async () => {
    const userId = 1;

    const mockUserDTO: userDTO = {
        id: userId,
        username: "TestUser",
        email: "test@example.com",
        role: "user",
        created_at: new Date(Date.now()),
    };

    MockedUserRepository.findById.mockResolvedValue(mockUserDTO);

    const result = await userService.getUser(userId);

    expect(result).toEqual(mockUserDTO);
    expect(MockedUserRepository.findById).toHaveBeenCalledWith(userId);
});
```

Caso 2 — Producto No Encontrado

- Objetivo: Verificar que el servicio arroja una excepción cuando el repositorio no encuentra un producto.
- Configuración (Mocks):
- El userRepository.findById retorna null.
- Resultado Esperado: El servicio lanza el error: "User with ID X not found".
- Tipo: Caso límite.

```
test('should throw an error if user not found', async () => {
    const userId = 1;

    MockedUserRepository.findById.mockResolvedValue(null);

    await expect(userService.getUser(userId)).rejects.toThrow(`User with ID ${userId} not found`);

    expect(MockedUserRepository.findById).toHaveBeenCalledWith(userId);
});
```

Testing de use-auth.tsx: *Juan Camilo Lopez Leon*

A continuación, se describen las pruebas unitarias implementadas para el hook useAuth, responsable de gestionar la autenticación del usuario, incluyendo inicio de sesión, cierre de sesión y persistencia del estado mediante localStorage. Para aislar su comportamiento, se emplean mocks tanto de almacenamiento local como de la API de autenticación.

Pruebas de la inicialización del hook:

Caso 1 — Inicialización sin usuario autenticado

- **Objetivo:** Verificar que el hook inicia correctamente cuando no existe información previa en localStorage.
- localStorage está vacío.
- Se espera que user sea null y isAuthenticated = false.

```
it('debe inicializar con usuario nulo si no hay sesión guardada', () => {
  const { result } = renderHook(() => useAuth());

  expect(result.current.user).toBeNull();
  expect(result.current.isAuthenticated).toBe(false);
  expect(result.current.loading).toBe(false);
  expect(result.current.error).toBeNull();
});
```

Caso 2 — Inicialización con sesión activa

- **Objetivo:** Validar que el hook carga correctamente el usuario almacenado en localStorage.
- El mock de localStorage retorna un JSON válido de usuario.
- Se espera que user contenga los datos cargados.
-

```
it('debe inicializar con el usuario guardado en localStorage', () => {
  // Configurar localStorage con un usuario
  localStorage.setItem('user', JSON.stringify(mockUser));
  // ... resto de la prueba
});
```

Caso 3 — Datos corruptos en localStorage

- **Objetivo:** Comprobar que el hook maneja errores de parseo sin romper la aplicación.
- localStorage.setItem('user', 'invalid-json').
- Se espera que el hook ignore el valor y establezca user = null.

```
it('debe manejar datos corruptos en localStorage', () => {
  // Configurar localStorage con datos inválidos
  localStorage.setItem('user', 'invalid-json');

  const { result } = renderHook(() => useAuth());

  expect(result.current.user).toBeNull();
  expect(result.current.isAuthenticated).toBe(false);
  expect(result.current.error).toBeDefined();
});
```

Pruebas del método login:

Caso 1 — Login exitoso

- **Objetivo:** Confirmar que el estado del hook se actualiza correctamente cuando la autenticación es válida.
- El mock de la API retorna un usuario válido.
- Se espera que user se actualice y se guarde en localStorage.

```
it('debe hacer login exitosamente', async () => {
  // Configurar el mock para que retorne un usuario
  (api.post as jest.Mock).mockResolvedValueOnce({ data: mockUser });

  const { result, waitForNextUpdate } = renderHook(() => useAuth());

  // Llamar a la función de login
  await act(async () => {
    await result.current.login('test@example.com', 'password');
  });

  // Verificar que el usuario se guardó en el estado
  expect(result.current.user).toEqual(mockUser);
  expect(result.current.isAuthenticated).toBe(true);

  // Verificar que se guardó en localStorage
  expect(localStorage.setItem).toHaveBeenCalledWith('user', JSON.stringify(mockUser));
});
```

Caso 2 — Credenciales inválidas

- **Objetivo:** Validar el manejo de errores cuando la API rechaza las credenciales.
- La API mock lanza un error de autenticación.
- Se espera que el estado de error se actualice y no se guarde información en localStorage.

```
it('debe manejar credenciales inválidas', async () => {
  // Configurar el mock para que falle con un error de autenticación
  (api.post as jest.Mock).mockRejectedValueOnce({
    response: { status: 401, data: { message: 'Credenciales inválidas' } }
  });

  const { result } = renderHook(() => useAuth());

  await act(async () => {
    await expect(result.current.login('test@example.com', 'wrongpass')).rejects.toThrow();
  });

  // Verificar que no hay usuario autenticado
  expect(result.current.user).toBeNull();
  expect(result.current.isAuthenticated).toBe(false);
  expect(result.current.error).toBeDefined();

  // Verificar que no se guardó nada en localStorage
  expect(localStorage.setItem).not.toHaveBeenCalledWith('user', expect.anything());
});
```

Caso 3 — Sin conexión

- **Objetivo:** Comprobar que el hook reacciona correctamente ante problemas de red.
- El mock simula error tipo "Network error".
- Se espera que el hook establezca estado de error.

```
it('debe manejar errores de red', async () => {
  // Configurar el mock para simular un error de red
  (api.post as jest.Mock).mockRejectedValueOnce(new Error('Network Error'));

  const { result } = renderHook(() => useAuth());

  await act(async () => {
    await expect(result.current.login('test@example.com', 'password')).rejects.toThrow('Network Error');
  });

  // Verificar que se estableció el estado de error
  expect(result.current.error).toBeDefined();
  expect(result.current.user).toBeNull();

  // Verificar que no se guardó nada en localStorage
  expect(localStorage.setItem).not.toHaveBeenCalled();
});
```

Pruebas del método logout:

Caso 1 — Cierre de sesión exitoso

- **Objetivo:** Confirmar que el hook limpia correctamente el estado y elimina la información persistida.
- El usuario se encuentra autenticado.
- Se espera que user = null y se borre la clave en localStorage.

```
  describe('logout', () => {
    it('debe cerrar la sesión correctamente', async () => {
      // Configurar un usuario autenticado
      localStorage.setItem('user', mockUserString);

      const { result } = renderHook(() => useAuth());

      // Verificar que el usuario está autenticado
      expect(result.current.isAuthenticated).toBe(true);

      // Llamar a la función de logout
      act(() => {
        result.current.logout();
      });

      // Verificar que se cerró la sesión
      expect(result.current.user).toBeNull();
      expect(result.current.isAuthenticated).toBe(false);

      // Verificar que se eliminó del localStorage
      expect(localStorage.getItem('user')).toBeNull();
    });
  });
});
```

Testing deNavLink.tsx: *Juan Camilo Lopez Leon*

A continuación se describen las pruebas unitarias para el componente NavLink, una extensión personalizada del NavLink de React Router que incorpora manejo adicional de clases condicionales mediante la función cn

Renderizado básico

Caso 1 — Renderizado mínimo

- **Objetivo:** Verificar que el componente se renderiza con las propiedades esenciales.
- Se renderiza con to="/test" y texto "Test Link".
- Se espera que el elemento exista y su href sea correcto.

```
it('aplica la clase base correctamente', () => {
  const className = 'custom-class';
  renderNavLink({ className });

  const link = screen.getByRole('link', { name: 'Test Link' });
  expect(link).toHaveClass(className);
});
```

Clases CSS

Caso 1 — Clase base personalizada

- **Objetivo:** Confirmar que se aplican correctamente las clases provenientes del prop className.
- className="custom-class"
- El enlace debe contener esta clase.

Caso 2 — Estado pendiente

- **Objetivo:** Validar el comportamiento cuando el enlace se encuentra en transición (isPending = true).
- Se verifica que cn sea llamado con los parámetros esperados.

```
it('aplica la clase pending cuando el enlace está pendiente', () => {
  const pendingClassName = 'pending-class';

  // Limpiamos las llamadas anteriores al mock
  mockCn.mockClear();

  // Sobrescribimos el mock de NavLink
  const originalNavLink = vi.mocked(require('react-router-dom')).NavLink;

  try {
    vi.mocked(require('react-router-dom')).NavLink = vi.fn().mockImplementation(({ className, ...props }) => {
      const isActive = false;
      const isPending = true;
      const computedClassName = typeof className === 'function'
        ? className({ isActive, isPending })
        : className;

      return (
        <a
          href={props.to}
          className={computedClassName}
          data-testid={props['data-testid']}
          onClick={props.onClick}
        >
          {props.children}
        </a>
      );
    });

    renderNavLink({
      to: '/other-route',
      pendingClassName
    });

    // Verificamos que se llamó a cn con los parámetros correctos
    expect(mockCn).toHaveBeenCalledWith(
      undefined,
      false,
      false
    );
  } finally {
    // Restauramos el mock original
    vi.mocked(require('react-router-dom')).NavLink = originalNavLink;
  }
});
```

Testing use-mobile.tsx: *Juan Camilo Lopez Leon*

Las siguientes pruebas evalúan el comportamiento del hook useIsMobile, cuya función es determinar si el dispositivo actual debe considerarse móvil en función del ancho de la ventana o del resultado de matchMedia.

Detección de tamaños de pantalla

Caso 1 — Pantalla móvil (< 768px)

- **Objetivo:** Validar que el hook identifique correctamente dispositivos móviles.
- Se simula un ancho de 767px.
- El resultado esperado es true.

```
it('debería devolver true cuando el ancho de la pantalla es menor que 768px', () => {
  window.innerWidth = 767;  // <-- Ancho de pantalla móvil
  const { result } = renderHook(() => useIsMobile());
  expect(result.current).toBe(true);  // <-- Verifica que es móvil
});
```

Caso 2 — Pantalla de escritorio (≥ 768px)

- **Objetivo:** Confirmar la detección de pantallas grandes.
- Se simula un ancho de 1024px.
- El valor esperado es false.

```
it('debería devolver false cuando el ancho de la pantalla es 768px o más', () => {
  window.innerWidth = 1024;  // <-- Ancho de pantalla de escritorio
  const { result } = renderHook(() => useIsMobile());
  expect(result.current).toBe(false);  // <-- Verifica que no es móvil
});
```

Reacción a cambios del tamaño

Caso 1 — Cambio de escritorio a móvil

- **Objetivo:** Verificar que el hook responda a variaciones en el viewport.
- Se inicia con 1024px y luego se cambia a 500px.
- El estado debe actualizarse a true.

```
it('debería actualizar el estado cuando cambia el tamaño de la ventana', () => {
  window.innerWidth = 1024;
  const { result } = renderHook(() => useIsMobile());

  // Verificar estado inicial
  expect(result.current).toBe(false);

  // Simular cambio a móvil
  act(() => {
    window.innerWidth = 500;
    // Disparar manualmente el evento de cambio
    if (eventListeners.change) {
      eventListeners.change({ matches: true });
    }
  });

  // Verificar que el estado se actualizó
  expect(result.current).toBe(true);
});
```

Limpieza de listeners

Caso 1 — Desmontaje del componente

- **Objetivo:** Confirmar que el hook elimina correctamente los event listeners asociados a matchMedia.
- Se desmonta el componente que usa el hook.
- Se espera que el listener sea removido.

```
  it('debería limpiar el event listener al desmontar', () => {
    const { unmount } = renderHook(() => useIsMobile());

    // Verificar que se agregó el event listener
    expect(Object.keys(eventListeners)).toContain('change');

    // Desmontar el componente
    unmount();

    // Verificar que se eliminó el event listener
    expect(eventListeners.change).toBeUndefined();
  });
});
```

Testing product.service.ts - Alejandro Ramirez Nunez

A continuación se documenta la prueba unitaria implementada para el método getAllProducts() del servicio de productos, responsable de consultar al backend la lista completa de productos disponibles. Las pruebas fueron implementadas utilizando Vitest, herramienta adecuada para proyectos en TypeScript + React, cumpliendo con los criterios de aislamiento mediante mocks y validación de la funcionalidad esencial.

1. Prueba del método getAllProducts()

Esta prueba verifica el comportamiento principal del servicio al consultar la API para obtener el inventario de productos. Se utilizaron mocks sobre apiClient, lo que permite evaluar exclusivamente la lógica del servicio sin depender del backend real.

Caso 1 — Obtención exitosa de productos

Objetivo: Validar que el servicio:

- Realiza correctamente la llamada a la API.
- Recibe y retorna un arreglo de productos válido.
- Mantiene la estructura esencial de cada producto.
- Soporta correctamente el caso límite donde el backend puede retornar un arreglo vacío.

Configuración (Mocks):
`apiClient.get` es sobreescrito mediante `vi.mocked()` para retornar una respuesta simulada con status 200 y un arreglo de productos representativo.

Datos simulados:
Se simula la existencia de dos productos:

- Paracetamol 500mg
- Ibuprofeno 400mg

Cada uno con campos típicos: `id`, `name`, `type`, `price`, `stock`.

Resultado esperado:
El servicio debe retornar un arreglo con ese mismo contenido. Se valida específicamente:

- Que el tamaño del arreglo sea 2.
- Que los nombres y precios coincidan con los definidos en la respuesta simulada.
- Caso límite: si el backend devolviera un arreglo vacío, debería retornarse sin errores.

Tipo: Caso esencial / Integridad de datos.

Testing de auth.service.ts - Alejandro Ramirez Nunez

A continuación, se describen las pruebas unitarias implementadas para el método **login** del `AuthService`, el cual es responsable de autenticar a los usuarios enviando sus credenciales al backend y procesando las respuestas correspondientes. El servicio depende del cliente HTTP `apiClient`, el cual es simulado mediante *mocking* con Vitest, permitiendo aislar completamente la lógica del servicio y validar su comportamiento bajo diferentes escenarios.

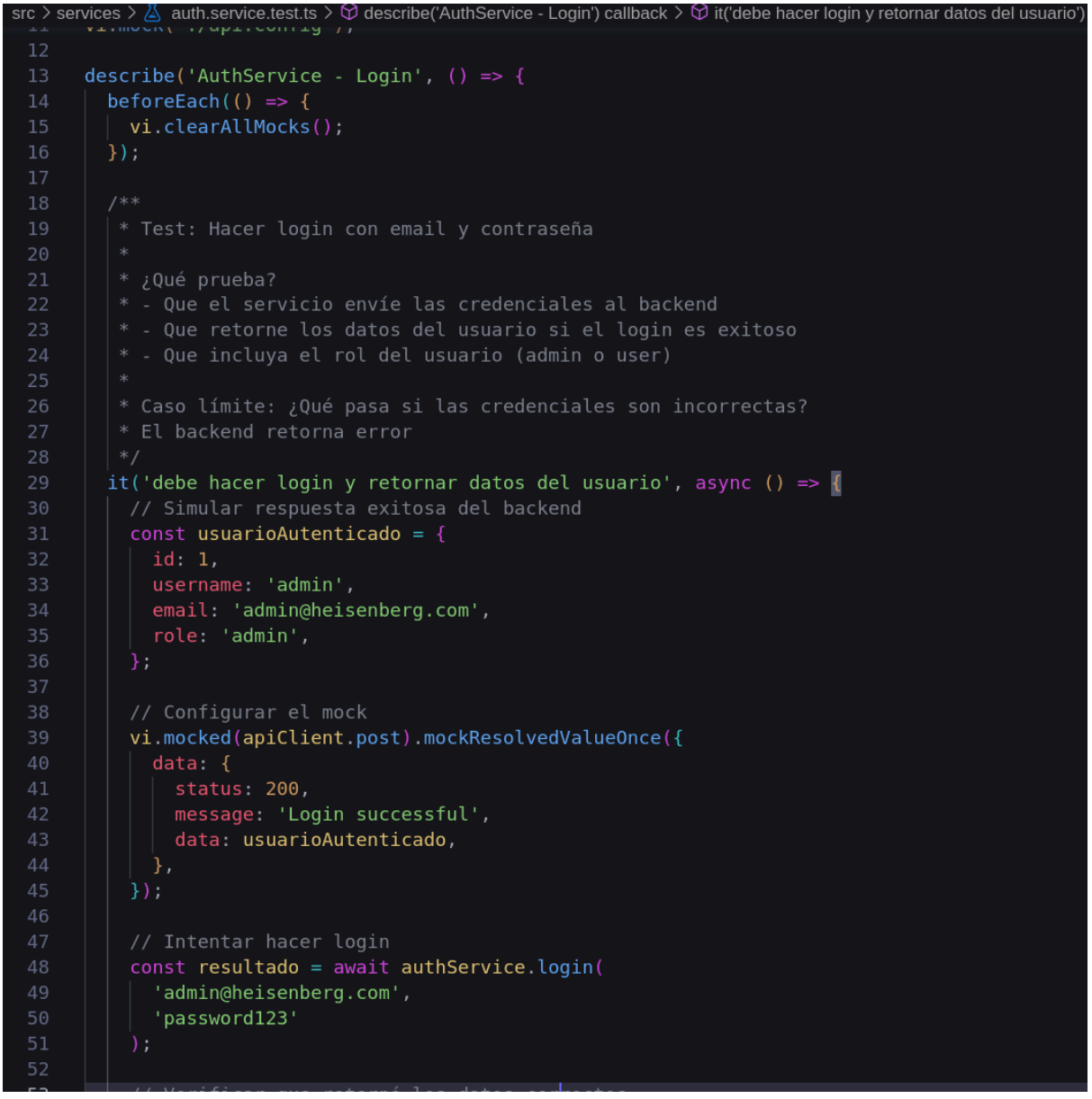
Las pruebas fueron implementadas usando **Vitest**, una herramienta adecuada para proyectos en JavaScript/TypeScript y compatible con entornos frontend.
Cada caso está diseñado para validar la funcionalidad esencial del servicio, incluyendo casos exitosos, fallos esperados y errores límite.

1. Prueba del método login(email, password)

Este método es responsable de autenticar al usuario mediante una petición al backend. Se evaluaron tres comportamientos fundamentales:

Caso 1 — Login exitoso

Objetivo: Verificar que el servicio envía correctamente las credenciales al backend y retorna los datos del usuario autenticado (incluyendo el rol).



Configuración (Mocks):

- `apiClient.post` retorna una respuesta exitosa con los datos del usuario.

Resultado esperado:

- El servicio debe retornar un objeto con las propiedades:

    - `username`
    - `email`
    - `role`

- Todos los campos deben coincidir con los enviados por el backend.

Tipo: Caso esencial / Flujo principal del login.

Caso 2 — Credenciales incorrectas

Objetivo: Validar que el servicio maneje adecuadamente las respuestas de error del backend cuando el usuario o la contraseña son inválidos.

```
it('debe lanzar error cuando las credenciales son incorrectas', async () => {
  // Simular respuesta del backend indicando credenciales inválidas
  vi.mocked(apiClient.post).mockResolvedValueOnce({
    data: {
      status: 401,
      message: 'Credenciales inválidas',
      data: null,
    },
  });

  await expect(
    authService.login('noexiste@heisenberg.com', 'badpassword')
  ).rejects.toThrow('Credenciales inválidas');
});
```

Configuración (Mocks):

- `apiClient.post` retorna `{ status: 401, message: "Credenciales inválidas" }`.

Resultado esperado:

- El servicio debe lanzar una excepción con el mensaje:
  "Credenciales inválidas"

Tipo: Caso de fallo / Seguridad.

Caso 3 — Error de red o fallo de conexión

Objetivo: Comprobar que el servicio maneje correctamente errores de red, caídas del servidor o fallos en la conexión.

```
  it('debe lanzar error cuando hay fallo de conexión al servidor', async () => {
    // Simular un AxiosError sin response (p. ej. network error)
    vi.mocked(apiClient.post).mockRejectedValueOnce(new AxiosError('Network Error'));

    await expect(
      authService.login('admin@heisenberg.com', 'password123')
    ).rejects.toThrow('Error al conectar con el servidor');
  });
});
```

Configuración (Mocks):

- `apiClient.post` lanza un `AxiosError` sin `response` (típico de errores de red).

Resultado esperado:

- El servicio debe lanzar una excepción con el mensaje:
  "Error al conectar con el servidor"

Tipo: Caso límite / Infraestructura.