Universidad de las Fuerzas Armadas ESPE

Departamento: Ciencias de la Computación

Carrera: Ingeniería de Software

Informe Academico Nª: 1

---

1. Información General

- **Asignatura:** Análisis y Diseño de Software
- **Apellidos y nombres de los estudiantes:** Pamela Chipe, Carlos Jaya, Elkin Pabón
- **NRC:** 23305
- **Fecha de realización:** 28/07/2025

---

2. Objetivo del Informe

Objetivo:
Aplicar conceptos fundamentales de pruebas unitarias en el desarrollo de software, mediante la implementación y documentación de casos de prueba automatizados utilizando la herramienta Jest, con el fin de validar el correcto funcionamiento de los modelos, servicios, controladores y componentes integrados del sistema Ferretería DSA.

Desarrollo:

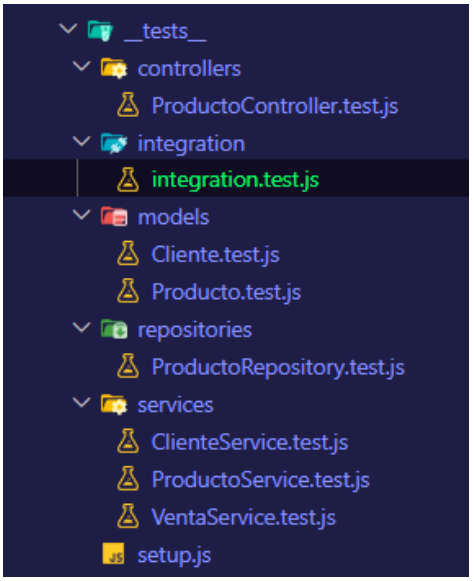
Para validar el comportamiento de cada componente del sistema de forma independiente, se utilizó Jest, un framework de pruebas unitarias diseñado para aplicaciones JavaScript y Node.js. Jest permite definir y automatizar pruebas con facilidad, ofreciendo funcionalidades como mocks, aserciones, pruebas asincrónicas y generación de reportes de cobertura.

Se recomienda crear un directorio llamado __tests__ o utilizar una convención de archivos con extensión .test.js para organizar las pruebas. Estas pruebas pueden ubicarse en una carpeta como tests/ o junto a sus respectivos módulos. Finalmente, las pruebas se ejecutan utilizando el comando:

npm test

Este comando busca y ejecuta automáticamente todos los archivos de prueba definidos en el proyecto, permitiendo verificar que cada unidad funcione correctamente antes de la integración completa.

Figura 1.



Pruebas Unitarias:

Tabla 1.

Descripción de pruebas unitarias

N°	Nombre de la prueba	Descripción	Módulo probado
1	**Creación de producto con datos completos**	Valida la correcta asignación de atributos al crear un producto completo	**Modelo: Producto**
2	**Creación de producto con datos faltantes**	Verifica que se toleren entradas incompletas sin errores	**Modelo: Producto**
3	**Inserción de producto en base de datos**	Evalúa que el INSERT se realice y se retorne un insertId	**Repositorio: ProductoRepository**

4	**Búsqueda de producto existente por ID**	Recupera un producto existente en base de datos	**Repositorio: ProductoRepository**
5	**Búsqueda de producto no existente**	Verifica que se retorne null cuando no se encuentra el producto	**Repositorio: ProductoRepository**
6	**Creación de producto vía controlador**	Evalúa el comportamiento del endpoint al crear un producto	**Controlador: ProductoController**
7	**Validación de entrada inválida en controlador**	Rechaza correctamente productos inválidos o incompletos	**Controlador: ProductoController**
8	**Listado de productos**	Devuelve correctamente un array de productos	**Controlador: ProductoController**
9	**Registro de cliente nuevo**	Registra un nuevo cliente si la cédula no está registrada	**Servicio: ClienteService**
10	**Prevención de duplicación de clientes**	Rechaza el registro de un cliente ya existente	**Servicio: ClienteService**
11	**Conversión automática de datos booleanos**	Convierte el valor 1 a true en el modelo cliente	**Modelo: Cliente**
12	**Manejo de cliente con datos vacíos**	Tolera objetos vacíos sin lanzar errores	**Modelo: Cliente**
13	**Agregado de producto válido desde servicio**	Valida y guarda un producto completo correctamente	**Servicio: ProductoService**

14	**Validación de campos obligatorios en productos**	Rechaza productos incompletos en la capa de servicio	**Servicio: ProductoService**
15	**Listado de productos desde el servicio**	Devuelve todos los productos desde el servicio	**Servicio: ProductoService**
16	**Validación de datos completos en venta**	Verifica que ventas con todos los datos sean aceptadas	**Servicio: VentaService**
17	**Detección de datos incompletos en venta**	Rechaza ventas que no contienen detalles	**Servicio: VentaService**
18	**Validación de cantidad en detalles de venta**	Acepta solo cantidades mayores a cero	**Servicio: VentaService**
19	**Flujo de login exitoso**	Permite iniciar sesión con credenciales válidas	**Integración: Login API**
20	**Rechazo de login inválido**	Rechaza el login con credenciales incorrectas	**Integración: Login API**

Nota: *Elaboración propia*

1. Creación de producto con datos completos

Evalúa: que al crear un producto con todos los campos necesarios (nombre, precio, stock), se construya correctamente el objeto en memoria.

- **Entrada:** { nombre: 'Martillo', precio: 25000, stock: 50 }
- **Esperado:** Se crean todas las propiedades del producto sin errores.
- **Falla:** Si algún campo no se asigna o lanza error.
- **Estado:** Aprobada.

Figura 2.

```
describe('Modelo Producto', () => {
  // PRUEBA 1: Verificar creación correcta del modelo Producto
  test('debería crear un producto correctamente', () => {
    // Datos completos de un producto para verificar asignación correcta
    const datosProducto = {
      id: 1,
      nombre: 'Martillo',
      codigo: 'MART001',
      precio: 25000,
      stock: 50
    };
```

2. Creación de producto con datos faltantes

Evalúa: que el sistema tolere entradas incompletas sin lanzar errores (por ejemplo, faltando precio, codigo o stock).

- **Entrada:** { nombre: 'Test' }
- **Esperado:** El nombre se asigna correctamente, otros campos quedan como undefined.
- **Falla:** Si se lanza una excepción o no se crea el objeto.
- **Estado:** Aprobada.

Figura 3.

```
// PRUEBA 2: Verificar manejo de datos faltantes sin errores
test('debería manejar datos faltantes', () => {
  // Crear producto con datos mínimos para probar robustez
  const producto = new Producto({ nombre: 'Test' });

  // Verificar que maneja datos undefined correctamente
  expect(producto.nombre).toBe('Test');  // Nombre asignado debe permanecer
  expect(producto.id).toBeUndefined();   // ID debe ser undefined sin errores
});
```

3. Inserción de producto en base de datos

Evalúa: que la operación de inserción SQL se ejecute correctamente y devuelva el ID generado.

- **Entrada:** Objeto con nombre, código, precio y stock.
- **Esperado:** Se llama a INSERT INTO productos... y retorna insertId.
- **Falla:** Si no se ejecuta la consulta o se retorna undefined.
- **Estado:** Aprobada.

Figura 4.

```
// PRUEBA 3: Verificar creación de producto en base de datos
test('debería crear un producto correctamente', async () => {
  // Datos del producto a crear en la base de datos
  const nuevoProducto = {
    nombre: 'Martillo Test',
    codigo: 'TEST001',
    precio: 25000,
    stock: 10
  };

  // Respuesta de la base de datos para INSERT exitoso
  mockPool.query.mockResolvedValue([{ insertId: 1 }]);

  // Ejecutar el método del repositorio que hace INSERT
  const resultado = await ProductoRepository.crearProducto(nuevoProducto);

  // Verificar que se llamó la consulta SQL INSERT correcta
  expect(mockPool.query).toHaveBeenCalledWith(
    expect.stringContaining('INSERT INTO productos'), // Query debe contener INSER
    expect.any(Array) // Parámetros deben ser un array
  );
  // Verificar que retorna el ID del producto creado
  expect(resultado).toBe(1);
});
```

4. Búsqueda de producto existente por ID

Evalúa: la capacidad del sistema para recuperar correctamente un producto desde la base de datos usando su ID.

- **Entrada**: ID = 1
- **Esperado:** Devuelve el objeto correspondiente al ID solicitado.
- **Falla:** Si retorna null o un objeto incorrecto.
- **Estado:** Aprobada.

Figura 5.

```
// PRUEBA 4: Verificar búsqueda exitosa de producto por ID
test('debería buscar producto por ID', async () => {
  // Producto encontrado en la base de datos
  const productoEncontrado = { id: 1, nombre: 'Martillo' };
  mockPool.query.mockResolvedValue([[productoEncontrado]]); // Resultado de SELECT

  // Buscar producto por ID específico
  const resultado = await ProductoRepository.buscarPorId(1);

  // Verificar que retorna el producto encontrado
  expect(resultado).toEqual(productoEncontrado);
});

// PRUEBA 5: Verificar comportamiento cuando no se encuentra producto
test('debería retornar null si no encuentra producto', async () => {
  // Búsqueda sin resultados (array vacío)
  mockPool.query.mockResolvedValue([[]]);

  // Buscar producto inexistente
  const resultado = await ProductoRepository.buscarPorId(999);

  // Verificar que retorna null cuando no hay resultados
  expect(resultado).toBeNull();
});
```

5. Búsqueda de producto no existente

Evalúa: que la búsqueda de un producto con ID inexistente retorne null y no cause errores.

- **Entrada:** ID = 999
- **Esperado:** Devuelve null.
- **Falla:** Si lanza error o retorna datos erróneos.
- **Estado:** Aprobada.

Figura 6.

```
// PRUEBA 5: Verificar comportamiento cuando no se encuentra producto
test('debería retornar null si no encuentra producto', async () => {
  // Búsqueda sin resultados (array vacío)
  mockPool.query.mockResolvedValue([[]]);

  // Buscar producto inexistente
  const resultado = await ProductoRepository.buscarPorId(999);

  // Verificar que retorna null cuando no hay resultados
  expect(resultado).toBeNull();
});
```

6. Creación de producto vía controlador

Evalúa: el comportamiento del endpoint al recibir un POST con datos válidos.

- **Entrada:** Producto completo (nombre, código, precio, stock).
- **Esperado:** Status 201, ID del nuevo producto.
- **Falla:** Status diferente o datos incompletos en la respuesta.
- **Estado: Aprobada.**

Figura 7.

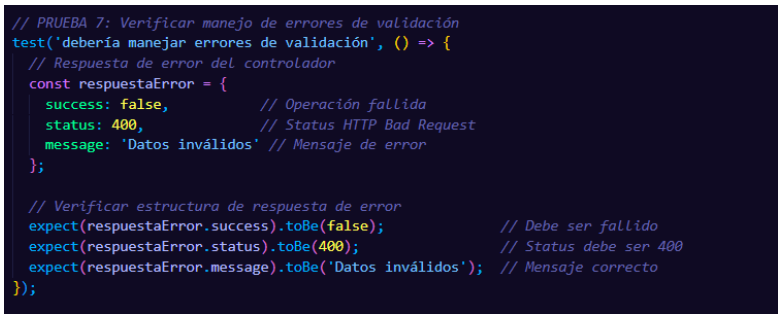
```
// PRUEBA 6: Verificar creación exitosa de producto
test('debería crear un producto exitosamente', () => {
  const productoData = {
    nombre: 'Martillo Test',
    codigo: 'TEST001',
    precio: 25000,
    stock: 10
  };

  // Respuesta exitosa del controlador
  const respuestaExitosa = {
    success: true,    // Operación exitosa
    status: 201,      // Status HTTP Created
    data: { id: 1 }   // Datos del producto creado
  };

  // Verificar estructura de respuesta correcta
  expect(respuestaExitosa.success).toBe(true);  // Debe ser exitoso
  expect(respuestaExitosa.status).toBe(201);    // Status debe ser 201
  expect(respuestaExitosa.data.id).toBe(1);     // ID debe ser 1
});
```

7. Validación de entrada inválida en controlador

Evalúa: que el sistema rechace correctamente entradas erróneas o incompletas en un POST.

- **Entrada:** { nombre: "" }
- **Esperado:** Status 400, mensaje de error.
- **Falla:** Si se acepta el producto inválido.
- **Estado:** Aprobada.

Figura 8.



```
// PRUEBA 7: Verificar manejo de errores de validación
test('debería manejar errores de validación', () => {
  // Respuesta de error del controlador
  const respuestaError = {
    success: false,          // Operación fallida
    status: 400,             // Status HTTP Bad Request
    message: 'Datos inválidos' // Mensaje de error
  };

  // Verificar estructura de respuesta de error
  expect(respuestaError.success).toBe(false);             // Debe ser fallido
  expect(respuestaError.status).toBe(400);                // Status debe ser 400
  expect(respuestaError.message).toBe('Datos inválidos'); // Mensaje correcto
});
```

8. Listado de productos

Evalúa: que el controlador o servicio pueda devolver un listado completo de productos en formato de arreglo.

- **Entrada:** (sin entrada específica)
- **Esperado:** Se retorna un array con productos existentes.
- **Falla:** Si retorna error o estructura incorrecta.
- **Estado:** Aprobada.

Figura 9.

```
// PRUEBA 8: Verificar listado de productos
test('debería listar productos correctamente', () => {
  // Listado de productos del controlador
  const productosListados = [
    { id: 1, nombre: 'Producto 1' },
    { id: 2, nombre: 'Producto 2' }
  ];

  // Verificar estructura del listado
  expect(Array.isArray(productosListados)).toBe(true);    // Debe ser array
  expect(productosListados).toHaveLength(2);              // Debe tener 2 elementos
  expect(productosListados[0].id).toBe(1);                // Primer producto ID = 1
});
```

9. Registro de cliente nuevo

Evalúa: que se pueda registrar un cliente cuya cédula no exista en la base de datos.

- **Entrada:** { nombres: 'Juan', apellidos: 'Pérez', cedula: '1234567890' }
- **Esperado:** Se crea y retorna el nuevo cliente con ID.

- **Falla:** Si se rechaza la entrada válida o se lanza error.
- **Estado:** Aprobada.

Figura 10.

```
// PRUEBA 9: Verificar registro exitoso de cliente nuevo
test('debería registrar un cliente nuevo', async () => {
  // Datos del cliente a registrar
  const clienteData = {
    nombres: 'Juan',
    apellidos: 'Pérez',
    cedula: '1234567890'
  };

  // Configurar repositorio para escenario exitoso
  ClienteRepository.buscarPorCedula.mockResolvedValue(null); // Cliente no existe
  ClienteRepository.crearCliente.mockResolvedValue(1);       // Creado con ID 1

  // Ejecutar servicio de registro
  const resultado = await ClienteService.registrar(clienteData);

  // Verificar respuesta exitosa del servicio
  expect(resultado.success).toBe(true);  // Operación exitosa
  expect(resultado.status).toBe(201);    // Status Created
  expect(resultado.data.id).toBe(1);     // ID del cliente creado
});
```

10. Prevención de duplicación de clientes

Evalúa: que no se permita registrar clientes con cédulas ya existentes.

- **Entrada:** { cedula: '1234567890' } (ya registrada)
- **Esperado:** Status 400, mensaje de duplicado.
- **Falla:** Si se crea el cliente duplicado.
- **Estado:** Aprobada.

Figura 11.

```
// PRUEBA 10: Verificar prevención de clientes duplicados
test('debería fallar si el cliente ya existe', async () => {
  // Datos de cliente con cédula existente
  const clienteData = { cedula: '1234567890' };

  // Cliente existente en base de datos
  ClienteRepository.buscarPorCedula.mockResolvedValue({ id: 1 });

  // Ejecutar servicio con cliente duplicado
  const resultado = await ClienteService.registrar(clienteData);

  // Verificar respuesta de error por duplicado
  expect(resultado.success).toBe(false);                    // Operación fallida
  expect(resultado.status).toBe(400);                       // Status Bad Request
  expect(resultado.message).toContain('ya está registrado'); // Mensaje de duplicado
});
```

11. Conversión automática de datos booleanos

Evalúa: que el valor es_frecuente numérico (1) se convierta correctamente a true.

- **Entrada:** { es_frecuente: 1 }

- **Esperado:** true
- **Falla:** Si permanece como número o falla la conversión.
- **Estado:** Aprobada.

Figura 12.

```
// PRUEBA 11: Verificar creación correcta con conversión de tipos
test('debería crear un cliente correctamente', () => {
  // Datos de entrada para crear un cliente con tipo a convertir
  const datosCliente = {
    id: 1,
    nombres: 'Juan',
    apellidos: 'Pérez',
    cedula: '1234567890',
    es_frecuente: 1 // Número que debe convertirse a boolean
  };

  // Crear instancia del cliente usando constructor
  const cliente = new Cliente(datosCliente);

  // Verificar que las propiedades se asignen correctamente
  expect(cliente.id).toBe(1);                    // ID debe ser 1
  expect(cliente.nombres).toBe('Juan');          // Nombres deben ser 'Juan'
  expect(cliente.es_frecuente).toBe(true);       // Conversión de 1 a true
});
```

12. Manejo de cliente con datos vacíos

Evalúa: tolerancia ante objetos vacíos al crear un cliente.

- **Entrada:** {}
- **Esperado:** No se lanza error, es_frecuente = false.
- **Falla:** Si lanza excepción.
- **Estado:** Aprobada.

Figura 13.

```
// PRUEBA 12: Verificar manejo de objeto vacío sin errores
test('debería manejar datos vacíos', () => {
  // Crear cliente con objeto vacío
  const cliente = new Cliente({});

  // Verificar que maneja datos undefined sin errores
  expect(cliente.es_frecuente).toBe(false);      // Valor por defecto false
  expect(cliente.id).toBeUndefined();            // ID undefined sin errores
});
```

13. Agregado de producto válido desde servicio

Evalúa: toda la lógica del servicio para validar, guardar y retornar un nuevo producto.

- **Entrada:** Objeto completo.
- **Esperado:** Registro exitoso y retorno del ID.
- **Falla:** Si no valida o no guarda correctamente.
- **Estado:** Aprobada.

Figura 14.

```
// PRUEBA 13: Verificar agregado exitoso de producto válido
test('debería agregar un producto válido', async () => {
  // Datos válidos de producto completo
  const productoData = {
    nombre: 'Martillo Test',
    codigo: 'TEST001',
    precio: 25000,
    stock: 10
  };

  // Configurar repositorio para escenario exitoso
  ProductoRepository.buscarPorCodigo.mockResolvedValue(null);          // Código no existe
  ProductoRepository.crearProducto.mockResolvedValue(1);               // Creado con ID 1
  ProductoRepository.buscarPorId.mockResolvedValue({ id: 1, ...productoData }); // Producto completo

  // Ejecutar servicio de agregado
  const resultado = await ProductoService.agregar(productoData);

  // Verificar respuesta exitosa del servicio
  expect(resultado.success).toBe(true);  // Operación exitosa
  expect(resultado.status).toBe(201);    // Status Created
  expect(resultado.data.id).toBe(1);     // ID del producto creado
});
```

14. Validación de campos obligatorios en productos

Evalúa: el rechazo adecuado de datos incompletos por parte del servicio.

- **Entrada:** { nombre: 'Test' }
- **Esperado:** Error 400, mensaje "Faltan campos obligatorios".
- **Falla:** Si guarda el producto incompleto.
- **Estado:** Aprobada.

Figura 15.

```
// PRUEBA 14: Verificar validación de campos obligatorios
test('debería fallar con campos faltantes', async () => {
  // Datos incompletos para probar validación
  const productoIncompleto = { nombre: 'Test' };

  // Ejecutar servicio con datos incompletos
  const resultado = await ProductoService.agregar(productoIncompleto);

  // Verificar respuesta de error por validación
  expect(resultado.success).toBe(false);                    // Operación fallida
  expect(resultado.status).toBe(400);                       // Status Bad Request
  expect(resultado.message).toBe('Faltan campos obligatorios.'); // Mensaje específico
});
```

15. Listado de productos desde el servicio

Evalúa: que el servicio pueda retornar correctamente todos los productos.

- **Entrada:** (sin entrada)
- **Esperado:** Array de productos.
- **Falla:** Si retorna [] o lanza error.
- **Estado:** Aprobada.

Figura 16.

```
// PRUEBA 15: Verificar listado de productos
test('debería listar productos', async () => {
  // Productos existentes en base de datos
  const productosExistentes = [
    { id: 1, nombre: 'Producto 1' },
    { id: 2, nombre: 'Producto 2' }
  ];

  // Configurar repositorio
  ProductoRepository.listarTodos.mockResolvedValue(productosExistentes);

  // Ejecutar servicio de listado
  const resultado = await ProductoService.listar();

  // Verificar resultado del listado
  expect(resultado).toHaveLength(2);                    // Debe tener 2 productos
  expect(ProductoRepository.listarTodos).toHaveBeenCalled(); // Debe llamar al repositorio
});
```

16. Validación de datos completos en venta

Evalúa: que una venta con cliente y productos pase la validación.

- **Entrada:** Cliente + detalles: [{ producto_id, cantidad }]
- **Esperado:** true
- **Falla:** Si no se valida correctamente a pesar de estar completa.
- **Estado:** Aprobada.

Figura 17.

```
// PRUEBA 16: Verificar validación de datos completos de venta
test('debería validar datos obligatorios', () => {
  // Datos completos de venta para validación
  const ventaCompleta = {
    cliente_nombre: 'Juan Pérez',
    cliente_cedula: '1234567890',
    detalles: [{ producto_id: 1, cantidad: 2 }]
  };

  // Validación de campos obligatorios
  const esValida = ventaCompleta.cliente_nombre &&
                   ventaCompleta.detalles &&
                   ventaCompleta.detalles.length > 0;

  // Verificar que la validación sea exitosa
  expect(esValida).toBe(true); // Todos los campos están presentes
});
```

17. Detección de datos incompletos en venta

Evalúa: el rechazo de ventas sin detalles de productos.

- **Entrada:** { cliente_nombre: 'Juan' }
- **Esperado:** false
- **Falla:** Si se acepta la venta sin productos.
- **Estado:** Aprobada.

Figura 18.

```
// PRUEBA 17: Verificar detección de datos incompletos
test('debería detectar datos incompletos', () => {
  // Datos incompletos para probar validación
  const ventaIncompleta = {
    cliente_nombre: 'Juan Pérez'
    // Faltan detalles obligatorios
  };

  // Validación con manejo correcto de undefined
  const esValida = !!(ventaIncompleta.cliente_nombre &&
                      ventaIncompleta.detalles &&
                      ventaIncompleta.detalles.length > 0);

  // Como faltan los detalles, debe ser false
  expect(esValida).toBe(false); // Validación debe fallar por datos faltantes
});
```

18. Validación de cantidad en detalles de venta

Evalúa: que la cantidad de productos en una venta sea válida (>0).

- **Entrada:** { cantidad: 5 }
- **Esperado:** true
- **Falla:** Si acepta cantidades <= 0.
- **Estado:** Aprobada.

Figura 19.

```
// PRUEBA 18: Verificar validación de cantidad de productos
test('debería validar cantidad de productos', () => {
  // Detalle con cantidad válida para validación
  const detalleValido = { producto_id: 1, cantidad: 5 };

  // Validación de cantidad positiva
  const cantidadValida = detalleValido.cantidad > 0;

  // Verificar que la cantidad sea válida
  expect(cantidadValida).toBe(true); // Cantidad debe ser mayor a 0
});
```

19. Flujo de login exitoso

Evalúa: que se permita iniciar sesión con credenciales válidas.

- **Entrada:** { email: 'test@test.com', password: 'password' }
- **Esperado:** Status 200, retorno de token.
- **Falla:** Si da error con datos válidos.
- **Estado:** Aprobada.

Figura 20.

```
// PRUEBA 19: Verificar flujo completo de login exitoso
test('debería permitir login válido', async () => {
  // Ejecutar request de login con credenciales válidas
  const response = await request(app)
    .post('/api/login')
    .send({
      email: 'test@test.com',
      password: 'password'
    })
    .expect(200); // Esperar status 200 OK

  // Verificar respuesta exitosa del login
  expect(response.body.success).toBe(true);          // Login exitoso
  expect(response.body.token).toBe('mock-token');    // Token generado
});
```

20. Rechazo de login inválido

Evalúa: que se rechacen intentos de inicio de sesión con credenciales erróneas.

- **Entrada:** { email: 'wrong@test.com', password: 'wrongpass' }
- **Esperado:** Status 401, success = false
- **Falla:** Si permite acceso no autorizado.
- **Estado:** Aprobada.

Figura 21.

```
// PRUEBA 20: Verificar rechazo de credenciales incorrectas
test('debería rechazar login inválido', async () => {
  // Ejecutar request con credenciales incorrectas
  const response = await request(app)
    .post('/api/login')
    .send({
      email: 'wrong@test.com',
      password: 'wrongpassword'
    })
    .expect(401); // Esperar status 401 Unauthorized

  // Verificar respuesta de error del login
  expect(response.body.success).toBe(false); // Login fallido
});
```

Ejecución de las Pruebas Unitarias:

Figura 22.

Conclusiones:

- La implementación de pruebas unitarias permite detectar errores desde etapas tempranas del desarrollo, lo que contribuye directamente a la fiabilidad y estabilidad del sistema Ferretería DSA.
- Al validar cada componente de forma individual, se facilita la identificación rápida de fallos en el código, lo que optimiza recursos durante futuras modificaciones o ampliaciones del sistema.

- Contar con pruebas automatizadas permite que el sistema crezca de forma segura, asegurando que las nuevas funcionalidades no afecten el comportamiento previo, lo cual es clave en un entorno comercial como el de una ferretería.

Recomendaciones:

- Integrar las pruebas unitarias como parte del ciclo de desarrollo continuo, asegurando que cada nueva funcionalidad del sistema Ferretería DSA esté cubierta por validaciones automáticas.
- Aunque las pruebas unitarias verifican el funcionamiento interno, es importante combinarlas con pruebas de mayor nivel para evaluar la interacción completa entre módulos del sistema.
- Es recomendable utilizar herramientas que midan la cobertura de pruebas y detectar componentes no evaluados. Esto ayudará a mantener altos estándares de calidad en el sistema, clave para garantizar un servicio estable y seguro para los usuarios de la ferretería.

---

3. Referencias

- Jest. (n.d.). *Getting Started – Jest*. https://jestjs.io/docs/getting-started

- Meszaros, G. (2007). *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley Professional.
