**Documento de Pruebas Unitarias**

**“SKIPUR”**

**Integrantes:**

Chavez Oscullo Klever Enrique

Guacan Rivera Alexander David

Trejo Duque Alex Fernando

**Fecha:** 2025-07-29

**1. Introducción y Metodología de Pruebas Unitarias**

El presente documento detalla los casos de prueba unitarios diseñados e implementados para el backend del proyecto SKIPUR. El objetivo de estas pruebas es verificar la correctitud lógica de los componentes individuales del sistema de forma aislada, garantizando que cada pieza de la lógica de negocio funcione según lo especificado.

Se ha utilizado el framework de pruebas **Vitest**, en conjunto con la librería **vitest-mock-extended**, para llevar a cabo las pruebas en un entorno de Node.js con TypeScript. La metodología se centra en la **capa de Aplicación (Casos de Uso)**, ya que esta contiene la lógica de negocio más pura y crítica del sistema.

Cada prueba sigue rigurosamente el patrón **Arrange-Act-Assert (AAA)**:

* **Arrange (Preparar):** Se inicializa el entorno de la prueba, creando mocks (simulaciones) de las dependencias externas (como los repositorios de base de datos o los servicios de correo) para aislar el componente bajo prueba.
* **Act (Actuar):** Se ejecuta el método o función que se está probando, pasándole las entradas de prueba definidas.
* **Assert (Verificar):** Se comprueba que el resultado obtenido (el valor de retorno o un error lanzado) coincide con el resultado esperado.

**2. Casos de Prueba Unitarios por Módulo**

A continuación, se documentan los casos de prueba para cada módulo funcional.

**Módulo: Especialidades**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Caso de Prueba | Entrada de Prueba (Arrange) | Resultado Esperado (Assert) | Resultado Obtenido | Estado |
| 1. Creación exitosa | - DTO con name válido (ej. "Cardiología").<br>- Mock de specialtyRepository.findByName devuelve null. | El caso de uso debe llamar a specialtyRepository.create y devolver el nuevo objeto de especialidad. | Coincide con el resultado esperado. | Aprobada |
| 2. Falla por nombre duplicado | - DTO con un name que ya existe.<br>- Mock de specialtyRepository.findByName devuelve una especialidad existente. | El caso de uso debe lanzar un Error con el mensaje "Ya existe una especialidad con este nombre.". El método create no debe ser llamado. | Coincide con el resultado esperado. | Aprobada |
| Caso de Prueba | **Entrada de Prueba (Arrange)** | **Resultado Esperado (Assert)** | **Resultado Obtenido** | **Estado** |
| 3. Desactivación exitosa | - id de una especialidad existente y activa.<br>- Mock de specialtyRepository.findById devuelve una especialidad con is\_active: true. | El caso de uso debe llamar a specialtyRepository.deactivate y devolver el objeto actualizado con is\_active: false. | Coincide con el resultado esperado. | Aprobada |
| 4. Falla por no encontrar | - id de una especialidad que no existe.<br>- Mock de specialtyRepository.findById devuelve null. | El caso de uso debe lanzar un Error con el mensaje "Especialidad no encontrada.". | Coincide con el resultado esperado. | Aprobada |
| 5. Falla por estar ya inactiva | - id de una especialidad ya inactiva.<br>- Mock de specialtyRepository.findById devuelve una espec+ialidad con is\_active: false. | El caso de uso debe lanzar un Error con el mensaje "Esta especialidad ya se encuentra inactiva.". | Coincide con el resultado esperado. | Aprobada |