

Prueba Unitaria

Trimestre VII 2025

ADSO

Pruebas de Software

Erick Ladino Martinez
Danna Segura Contreras

Servicio Nacional de Aprendizaje (SENA)

Programa de Análisis y Desarrollo de Sistemas de Información (ADSO)

Trimestre VII

Septiembre 18, 2025

1 Introducción

Esta guía de aprendizaje corresponde al módulo de Pruebas de Software en el programa ADSO, Trimestre VII 2025. El enfoque principal es comprobar que los datos del usuario se guarden perfectamente en la base de datos al registrarse.

2 Método

2.1 Enfoque

El módulo está enfocado en comprobar que los datos del usuario se guarden perfectamente en la base de datos al registrarse.

2.2 Historia de Usuario

“Como artista, quiero registrar mis datos personales y especialidad musical, para tener un perfil visible a otros usuarios.”

2.3 Descripción

Se crearon pruebas unitarias para los endpoints /registro y /login de tu API Flask, simulando el comportamiento de la base de datos para no afectar datos reales.

2.4 ¿Qué herramientas se usaron?

- Unit test: Framework estándar de Python para pruebas unitarias.
- unittest.mock: Permite simular (mockear) funciones y objetos, en este caso la conexión y el cursor de la base de datos.
- Flask test cliente: Herramienta de Flask para simular peticiones HTTP a la aplicación sin necesidad de levantar un servidor real.

2.5 ¿Cómo se estructuró la prueba?

- Clase de prueba: Se creó la clase TestA piAna que hereda de unittest.TestCase. setUp:
- Se inicializa el cliente de pruebas de Flash para simular peticiones.

- Mock de la base de datos: Se usó `@patch('api_aona.get_db_connection')` para reemplazar la función real de

2.6 Pruebas de endpoints

- Registro exitoso: Simula un registro correcto y verifica que el mensaje y el código de estado sean los esperados.
- Login exitoso: Simula un login correcto y verifica que el mensaje, el usuario y el código de estado sean los esperados.
- Login fallido: Simula un login con datos incorrectos y verifica que el mensaje y el código de estado sean los esperados.

2.7 ¿Por qué se usó mock?

El mock permite simular la respuesta de la base de datos, así:

- No se necesita una base de datos real.
- No se alteran datos reales.
- Las pruebas son rápidas y seguras.

3 Resultados

3.1 Código utilizado para la prueba

```
test_api_aona.py:from unittest.mock import patch, MagicMock

from api_aona import app

class TestApiAona(unittest.TestCase):
    def setUp(self):
        self.app = app.test_client()
        self.app.testing = True

    @patch('api_aona.get_db_connection')
    def test_registro_usuario_exitoso(self, mock_db_conn):
```

```
mock_conn = MagicMock()
mock_cursor = MagicMock()
mock_db_conn.return_value = mock_conn
mock_conn.cursor.return_value = mock_cursor

mock_conn.commit.return_value = None
mock_cursor.close.return_value = None
mock_conn.close.return_value = None

payload = {
    'nombre': 'Test User',
    'contacto': '123456789',
    'documento': '999999',
    'contrasena': 'testpass'
}

response = self.app.post('/registro', json=payload)
self.assertEqual(response.status_code, 201)
data = response.get_json()
self.assertEqual(data["message"], "Artista registrado con éxito")
self.assertTrue(data["success"])

@patch('api_aona.get_db_connection')
def test_login_usuario_exitoso(self, mock_db_conn):
    mock_conn = MagicMock()
    mock_cursor = MagicMock()
    mock_db_conn.return_value = mock_conn
    mock_conn.cursor.return_value = mock_cursor

    mock_cursor.fetchone.return_value = {
        'id': 1,
        'nombre': 'Test User',
        'documento': '999999',
        'contrasena': 'testpass'
    }

    mock_cursor.close.return_value = None
    mock_conn.close.return_value = None

    payload = {
        'documento': '999999',
        'contrasena': 'testpass'
```

```
}  
response = self.app.post('/login', json=payload)  
self.assertEqual(response.status_code, 200)  
data = response.get_json()  
self.assertEqual(data["message"], "Inicio de sesión exitoso")  
self.assertTrue(data["success"])  
self.assertIn("usuario", data)  
self.assertEqual(data["usuario"]["nombre"], "Test User")  
  
@patch('api_aona.get_db_connection')  
def test_login_usuario_fallido(self, mock_db_conn):  
    mock_conn = MagicMock()  
    mock_cursor = MagicMock()  
    mock_db_conn.return_value = mock_conn  
    mock_conn.cursor.return_value = mock_cursor  
  
    mock_cursor.fetchone.return_value = None  
    mock_cursor.close.return_value = None  
    mock_conn.close.return_value = None  
  
    payload = {  
        'documento': 'noexiste',  
        'contrasena': 'incorrecta'  
    }  
  
    response = self.app.post('/login', json=payload)  
    self.assertEqual(response.status_code, 401)  
    data = response.get_json()  
    self.assertEqual(data["message"], "Documento o contraseña incorrectos")  
    self.assertFalse(data["success"])  
  
if __name__ == '__main__':  
    unittest.main()
```

Esta prueba fue ejecutada en la terminal de la siguiente manera:

python -m unittest test_api_aona.py

3.2 Resultado

The screenshot shows a Visual Studio Code editor with a project named 'AONA'. The file explorer on the left shows the project structure, including a 'backend' folder containing 'api_aona.py' and 'test_api_aona.py'. The main editor window displays the content of 'test_api_aona.py', which is a unittest test case for the 'api_aona' module. The test case includes a setUp method, a patch for 'api_aona.get_db_connection', and a test method 'test_registro_usuario_exitoso' that uses MagicMock to simulate database interactions and a payload for a user registration request.

```

1 import unittest
2 from unittest.mock import patch, MagicMock
3 from api_aona import app
4
5
6 class TestApiAona(unittest.TestCase):
7     def setUp(self):
8         self.app = app.test_client()
9         self.app.testing = True
10
11     @patch('api_aona.get_db_connection')
12     def test_registro_usuario_exitoso(self, mock_db_conn):
13         mock_conn = MagicMock()
14         mock_cursor = MagicMock()
15         mock_db_conn.return_value = mock_conn
16         mock_conn.cursor.return_value = mock_cursor
17
18         mock_conn.commit.return_value = None
19         mock_cursor.close.return_value = None
20         mock_conn.close.return_value = None
21
22         payload = {
23             'nombre': 'Test User',
24             'contacto': '123456789',
25             'documento': '9999999',
26             'contrasena': 'testpass'

```

The terminal at the bottom shows the execution of the test command: `python -m unittest test_api_aona.py`. The output indicates that 3 tests were run successfully in 0.023 seconds.

```

File "C:\Users\Aprendiz\AppData\Local\Programs\Git\AONA\backend\test_api_aona.py", line 31, in TestApiAona
data = response.get_json()
^^^^^^^^
NameError: name 'response' is not defined
PS C:\Users\Aprendiz\AppData\Local\Programs\Git\AONA\backend> ^C
PS C:\Users\Aprendiz\AppData\Local\Programs\Git\AONA\backend> python -m unittest test_api_aona.py
...
Ran 3 tests in 0.023s

OK
PS C:\Users\Aprendiz\AppData\Local\Programs\Git\AONA\backend>

```

4 Conclusión

Las pruebas unitarias implementadas demuestran que los endpoints de registro y login de la API Flask funcionan correctamente bajo los escenarios simulados. El uso de mocks ha permitido realizar pruebas rápidas, seguras y sin alterar datos reales, cumpliendo con el objetivo de verificar el guardado y manejo de datos de usuarios. Este enfoque asegura una base sólida para futuros desarrollos y ajustes en el sistema, destacando la importancia de las pruebas en el ciclo de desarrollo de software.

5 Referencias

Referencias

- [1] Servicio Nacional de Aprendizaje. (2025). *Guía de aprendizaje: Pruebas de software*. SENA.
- [2] Python Software Foundation. (2023). *Unittest | Unit testing framework*.
<https://docs.python.org/3/library/unittest.html>
- [3] Pallets Projects. (2023). *Testing Flask applications*. <https://flask.palletsprojects.com/en/3.0.x/testing/>