NauTrade - Taller 3 - Testing

1. Introducción

NauTrade es un asistente virtual diseñado para facilitar y optimizar el proceso de importaciones hacia México. La aplicación nace ante la necesidad detectada por las dificultades que enfrentó un compañero al planificar la importación de computadores de distintas marcas y gamas, donde la complejidad de reunir información precisa y actualizada representaba un reto importante.

Utilizando inteligencia artificial, **NauTrade** proporciona de manera automatizada y en tiempo real datos esenciales para el proceso de importación, tales como:

- **Códigos arancelarios**: Identificación de los códigos tarifarios correspondientes.
- **Cálculo de impuestos:** Información detallada sobre el Impuesto General de Importación (IGI) y el Impuesto al Valor Agregado (IVA).
- **Otros costos y cargos:** Estimaciones de gastos adicionales como el Derecho de Trámite Aduanero (DTA) y las Normas Oficiales Mexicanas (NOMs) aplicables.
- **Requisitos adicionales:** Detalles sobre certificados, etiquetado y otros aspectos regulatorios según el tipo de producto.

La idea es que, con solo una consulta, el usuario pueda obtener toda la información necesaria para planificar su importación de forma precisa, reduciendo la dependencia de expertos externos, abaratando costos y aumentando la escalabilidad del proceso. Además, el proyecto se concibe como una valiosa oportunidad de aprendizaje y desarrollo profesional para el equipo, permitiendo la práctica de habilidades en documentación, planificación y gestión de requerimientos en el ámbito del desarrollo de software.

2. Resumen de los tests

A continuación se muestra un listado con la asignación de pruebas, el componente probado y las herramientas utilizadas, acompañado de imágenes (simuladas con screenshots) del código y del resultado de la ejecución.

- Sergio Nicolas Rey Mora

Tipo de prueba: Prueba unitaria

Componente probado:

> **Endpoint *'/google_login/'***: Verifica la autenticación y registro del usuario en la base de datos.

Framework: Pytest, FastAPI TestClient

Código del test:

```python
def test_google_login(fake_db):
    payload = {"email": "test@example.com"}
    response = client.post("/google_login/", json=payload)
    assert response.status_code == 200
    assert response.json()["message"] == "User logged in successfully"
```

Resultado de la ejecución:

- Jose Julian Pira Naranjo

Tipo de prueba: Prueba unitaria

Componente probado:

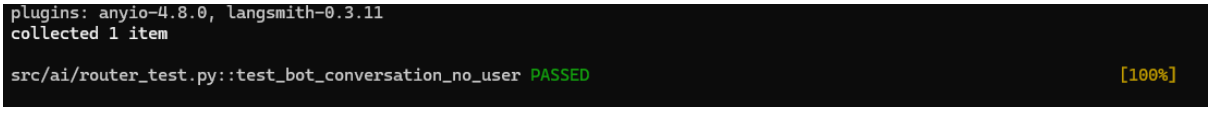
Endpoint *'/bot_conversation/{user_email}'*: Verifica la recuperación del historial de conversación del usuario.

Framework: Pytest, FastAPI TestClient

Código del test:

```python
def test_bot_conversation_no_user(fake_db):
    response =
client.get("/bot_conversation/nonexistent@example.com")
    assert response.status_code == 404
```

Resultado de la ejecución:



- Fabian Espitia

Tipo de prueba: Prueba unitaria

Componente probado:

Endpoint '/get_excel/': Válida la generación y retorno del archivo Excel.

Framework: Pytest, FastAPI TestClient

Código del test:

```python
def test_get_excel_no_user(fake_db):
    response =
client.get("/get_excel/?user_email=nonexistent@example.com")
    assert response.status_code == 404
```

Resultado de la ejecución:

- Javier Esteban Martinez Giron

Tipo de prueba: Prueba unitaria

Componente probado:

Endpoint *'/importation-bot/'*: Evalúa el manejo de payload inválido, comprobando que al no enviarse ni user_email ni user_id se retorne correctamente un error (HTTP 400) y se gestione la excepción adecuadamente.

Framework: Pytest, FastAPI TestClient

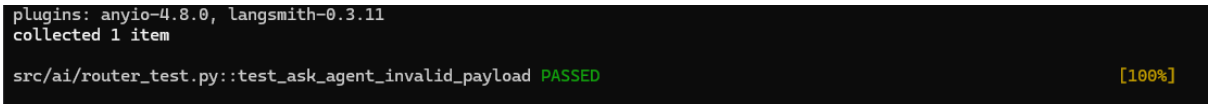
Código del test:

```python
def test_ask_agent_invalid_payload(fake_db):

    payload = {
        "prompt": "Test prompt",
        "user_email": None,
        "user_id": None
    }
    response = client.post("/importation-bot/", json=payload)

    # Se espera un error (HTTP 400) al no poder identificar al
usuario
    assert response.status_code == 400
```

Resultado de la ejecución:



3. Resultado de la ejecución:

A continuación se muestra un ejemplo global del resultado obtenido al ejecutar las pruebas con pytest:

En la salida se evidencia que todos los tests han pasado exitosamente, lo que confirma el correcto funcionamiento de cada uno de los endpoints de la aplicación.

---

4. Lecciones aprendidas y dificultades

Reflexión Grupal

Integrantes:

- Sergio Rey
- Jose Julian Pira
- Fabian Espitia
- Javi Esteban

Durante el desarrollo de los tests unitarios, el equipo aprendió y reafirmó la importancia de:

- **Simulación de dependencias:**
  Se destacó la necesidad de simular componentes externos, como la base de datos y las respuestas de servicios externos (por ejemplo, el agente de IA). Esto permite obtener resultados deterministas y facilita la detección de errores sin depender de servicios reales.

- **Uso de FastAPI TestClient:**
  La utilización de TestClient de FastAPI simplifica la emulación de peticiones HTTP y la verificación de respuestas, lo cual es fundamental para probar endpoints de manera aislada.

- **Inyección de dependencias:**
  La arquitectura basada en dependencias en FastAPI facilitó la sobrescritura de funciones y recursos (como la sesión de base de datos), permitiendo realizar pruebas unitarias de forma independiente y controlada.

- **Cobertura de casos de éxito y error:**
  Se reforzó la importancia de cubrir tanto escenarios exitosos (donde la operación se realiza correctamente) como casos en los que se espera un error (por ejemplo, cuando el usuario no existe), garantizando así una robusta gestión de excepciones.

Dificultades Encontradas

- **Simulación de la base de datos:**
  La necesidad de crear clases "fake" que replican el comportamiento de SQLAlchemy fue uno de los retos más significativos, ya que se debían simular consultas, filtrados y el orden de los mensajes sin una base de datos real.

- **Integración del agente de IA:**
  La lógica para simular el flujo del agente IA y su respuesta implicó sobrescribir métodos y asegurarse de que la estructura de mensajes se mantuviera coherente, lo cual fue un proceso de prueba y error.

- **Manejo de dependencias y excepciones:**
  Garantizar que cada test cubriera adecuadamente tanto el flujo normal como los

escenarios de error (por ejemplo, usuarios no encontrados) requirió un diseño cuidadoso de las pruebas.

En ***conclusión***, la experiencia permitió mejorar la calidad y confiabilidad del software desarrollado, consolidando buenas prácticas en el desarrollo de tests unitarios y la simulación de entornos complejos.