Informe testing

Materia:

ingeniería de software I

Presentado por:

Adrian Ramirez Gonzalez

Adrian Alexander Benavides

Brayan Alejandro Muñoz Pérez

Profesor:

Oscar Eduardo Álvarez Rodríguez

Universidad Nacional de Colombia

Facultad de ingeniería

Jueves 27 de febrero del 2025

Introducción de la aplicación

Boxy es una aplicación que facilita la gestión de inventarios al interior de las organizaciones la cual, además de permitir el almacenamiento, modificación, eliminación y consulta de productos, también ofrece funciones adicionales como la generación de informes sobre el inventario actual, la visualización de productos con stock bajo y la gestión de movimientos de entradas y salidas de productos. Adicionalmente, también existen funcionalidades asociadas a la gestión de usuarios, como registro de usuarios, inicio de sesión e incluso un sistema de recuperación de contraseña por medio del correo electrónico registrado.

Resumen de los tests

Adrian Ramirez Gonzalez

- **Tipo de prueba:** Unitaria
- **Descripción del componente que se probará**

  El servicio al que se le realizará la prueba unitaria es el **inicio de sesión**. Al usar este servicio, si la contraseña es correcta, se arrojará un código 200, mientras que si la contraseña es incorrecta, se arrojará un código 400.

  El objetivo de este proceso de testing será confirmar que el servicio arroja los códigos adecuados para cada caso, por lo cual se harán pruebas usando tanto contraseñas correctas como incorrectas. Además, debido a que dentro del software se contemplan 2 tipos de roles (administrador y usuario), las pruebas se realizarán para ambos roles.

- **Herramienta o framework usado**

  Ya que se está haciendo uso de django REST Framework para la API, se utilizará el módulo **test** de la librería **rest_framework.** En particular, se importarán APITestCase (clase diseñada para la realización de tests) y APIClient (herramienta que permite simular peticiones HTTP).

- **Screenshot del código del test**

  Para definir las pruebas, se crea un clase llamada TestLogin, la cual hereda de APITestCase. Dentro de esta clase se define el método setUp (heredado de APITestCase), el cual permite realizar algunas configuraciones antes de ejecutar las pruebas.

  En este caso, ya que se realiza un proceso de login, es necesario registrar previamente a los usuarios dentro de la base de datos, por lo cual se define un conjunto de datos para el usuario y para el administrador los cuales serán registrados en la base de datos. Además, se crea una organización a la cual estarán asociados tanto el usuario como el administrador (todos los usuarios y

administradores dentro de la aplicación web deben estar asociados a una organización).

```python
from rest_framework import status
from rest_framework.test import APITestCase, APIClient
from inventario.models import Usuario, Organizacion

class TestLogin(APITestCase):

    def setUp(self):

        self.loginURL = '/inventario/login'
        self.client = APIClient()

        self.datosAdministrador = {
            "nombre": "Jaime",
            "rol": "Administrador",
            "organizacion": "Organizacion1",
            "correo_electronico": "jaime@example.com",
            "password": "123456789"
        }

        self.datosUsuario = {
            "nombre": "Juan",
            "rol": "Usuario",
            "organizacion": "Organizacion1",
            "correo_electronico": "Juan@example.com",
            "password": "123456789"
        }

        self.nombreOrganizacion = "Organizacion1"

        self.organizacion = Organizacion.objects.create(nombre_organizacion=self.nombreOrganizacion)

        self.administrador = Usuario.objects.create_user(
            nombre=self.datosAdministrador['nombre'],
            rol=self.datosAdministrador['rol'],
            id_organizacion=self.organizacion,
            correo_electronico=self.datosAdministrador['correo_electronico'],
            password=self.datosAdministrador['password']
        )

        self.usuario = Usuario.objects.create_user(
            nombre=self.datosUsuario['nombre'],
            rol=self.datosUsuario['rol'],
            id_organizacion=self.organizacion,
            correo_electronico=self.datosUsuario['correo_electronico'],
            password=self.datosUsuario['password']
        )
```

Por último, se definen las siguientes 4 pruebas:

1. **Test para el correcto inicio de sesion del administrador**

```python
def test_login_correcto_administrador(self):

        response = self.client.post(self.loginURL,
                                    {
                                        "correo_electronico": self.datosAdministrador['correo_electronico'],
                                        "password": self.datosAdministrador['password']
                                    },
                                    format='json')

        self.assertEqual(response.status_code, status.HTTP_200_OK)
```

2. **Test para el correcto inicio de sesión del usuario**

```
def test_login_correcto_usuario(self):

        response = self.client.post(self.loginURL,
                                    {
                                        "correo_electronico": self.datosUsuario['correo_electronico'],
                                        "password": self.datosUsuario['password']
                                    },
                                    format='json')

        self.assertEqual(response.status_code, status.HTTP_200_OK)
```

3. **Test para el fallo en el inicio de sesion del administrador cuando la contraseña es incorrecta**

```
def test_login_incorrecto_administrador(self):

        response = self.client.post(self.loginURL,
                                    {
                                        "correo_electronico": self.datosAdministrador['correo_electronico'],
                                        "password": "passwordIncorrecta"
                                    },
                                    format='json')

        self.assertEqual(response.status_code, status.HTTP_400_BAD_REQUEST)
```

4. **Test para el fallo en el inicio de sesión del usuario cuando la contraseña es incorrecta**

```
def test_login_incorrecto_usuario(self):

        response = self.client.post(self.loginURL,
                                    {
                                        "correo_electronico": self.datosUsuario['correo_electronico'],
                                        "password": "passwordIncorrecta"
                                    },
                                    format='json')

        self.assertEqual(response.status_code, status.HTTP_400_BAD_REQUEST)
```

- **Resultado de la ejecución**

Para ejecutar los tests se hace uso del comando **python manage.py test inventario.tests.TestLogin** el cual se encargará la ejecución y nos dará los resultados. En este caso, los resultados fueron los siguientes:

```
(venv) C:\Users\adria\Desktop\ingenieria_de_software_1\Proyecto\backend_djangoREST>python manage.py test inventario.tests.TestLogin
Found 4 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
....
----------------------------------------------------------------------
Ran 4 tests in 3.911s

OK
Destroying test database for alias 'default'...
```

Los cuatro puntos de la quinta línea indican que los cuatro tests fueron superados satisfactoriamente, por lo cual se concluye que el servicio de inicio de sesión está funcionando correctamente.

Adrian Alexander Benavides

- **Tipo de prueba:** Unitaria
- **Descripción del componente que se probara**

  El módulo de gestión de inventario permite a los usuarios autenticados visualizar los productos disponibles, registrar nuevos productos, editar sus atributos y eliminarlos cuando sea necesario. Para ello, se diseñaron pruebas automatizadas utilizando el framework de pruebas de Django.

  Las pruebas implementadas fueron:

  - Visualización de inventario: Verifica que se puedan listar los productos almacenados.
  - Registro de producto: Comprueba que se pueda crear un nuevo producto correctamente.
  - Modificación de producto: Evalúa la capacidad de actualizar los datos de un producto existente.
  - Eliminación de producto: Confirma que se pueda eliminar un producto de la base de datos.

- **Herramienta o framework usado**

  Para la ejecución de las pruebas de este módulo de gestión de inventario, se utilizó las siguientes herramientas teniendo en cuenta como hemos venido manejando la estructura del proyecto:

  - Django REST framework (DRF): Para el desarrollo de la API.
  - TestCase de Django: Para la implementación de las pruebas unitarias.
  - APIClient de DRF: Para realizar solicitudes HTTP a la API durante los tests.
  - JWT (JSON Web Tokens): Para la autenticación de las solicitudes en las pruebas.

- **Screenshot del código del test**

  Para validar la funcionalidad del módulo de gestión de inventario, se implementaron pruebas automatizadas utilizando el framework de pruebas de Django. Estas pruebas verifican la correcta ejecución de las operaciones CRUD en la API. El código de pruebas se encuentra en el archivo tests.py y sigue la siguiente estructura:

  1. Configuración del entorno de pruebas

- Se crea un cliente de pruebas (APIClient) para simular las solicitudes HTTP a la API.
- Se genera un usuario administrador con credenciales para autenticación mediante JWT.
- Se inicializa un producto de prueba en la base de datos.

```python
from django.test import TestCase
from django.urls import reverse
from rest_framework import status
from rest_framework.test import APIClient
from rest_framework_simplejwt.tokens import RefreshToken
from .models import Usuario, Organizacion, Producto


You, 10 hours ago | 1 author (You)
class GestionInventarioTests(TestCase):
    def setUp(self):
        """
        Configuración inicial para los tests.
        """
        self.client = APIClient()
        self.organizacion = Organizacion.objects.create(nombre_organizacion="Org de prueba")

        # Crear un usuario administrador
        self.admin_user = Usuario.objects.create_user(
            nombre="admin",
            rol="Administrador",
            id_organizacion=self.organizacion,
            correo_electronico="admin@test.com",
            password="adminpassword"       You, 10 hours ago • Uncommitted changes
        )

        # Producto de prueba
        self.producto = Producto.objects.create(
            nombre="Producto de prueba",
            categoria="Categoría",
            cantidad=20,
            precio=150.00,
            stock_minimo=5
        )

        self.client.credentials(HTTP_AUTHORIZATION=f'Bearer {self.obtener_token(self.admin_user)}')

    def obtener_token(self, usuario):
        """
        Genera un token de autenticación JWT para el usuario dado.
        """
        refresh = RefreshToken.for_user(usuario)
        return str(refresh.access_token)
```

2. Definición de los casos de prueba
   - Visualización del inventario: Verifica que se pueden listar los productos disponibles.

```python
def test_visualizar_inventario(self):

    url = reverse('producto-list')
    response = self.client.get(url)
    self.assertEqual(response.status_code, status.HTTP_200_OK)
    self.assertGreaterEqual(len(response.data), 1)
```

- Registro de un nuevo producto: Comprueba que se pueda agregar un producto correctamente.

```python
def test_registrar_producto(self):
    """
    Verifica que se puede registrar un nuevo producto.
    """
    url = reverse('producto-list')
    data = {
        "nombre": "Nuevo Producto",
        "categoria": "Nueva Categoría",
        "cantidad": 50,
        "precio": 200.00,
        "stock_minimo": 10
    }
    response = self.client.post(url, data, format='json')
    self.assertEqual(response.status_code, status.HTTP_201_CREATED)
    self.assertTrue(Producto.objects.filter(nombre="Nuevo Producto").exists())
```

- Modificación de un producto: Evalúa si se pueden actualizar los atributos de un producto.

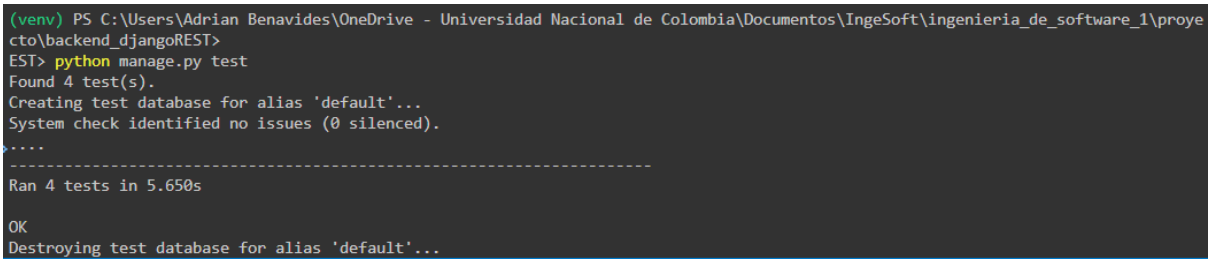
```python
def test_modificar_producto(self):
    """
    Verifica que se puede modificar un producto existente.
    """
    url = reverse('producto-detail', kwargs={'pk': self.producto.id})
    data = {"nombre": "Producto Modificado", "cantidad": 30}
    response = self.client.patch(url, data, format='json')
    self.assertEqual(response.status_code, status.HTTP_200_OK)
    self.producto.refresh_from_db()
    self.assertEqual(self.producto.nombre, "Producto Modificado")
    self.assertEqual(self.producto.cantidad, 30)
```

- Eliminación de un producto: Confirma que se puede borrar un producto de la base de datos

```python
def test_eliminar_producto(self):
    """
    Verifica que se puede eliminar un producto.
    """
    url = reverse('producto-detail', kwargs={'pk': self.producto.id})
    response = self.client.delete(url)
    self.assertEqual(response.status_code, status.HTTP_204_NO_CONTENT)
    self.assertFalse(Producto.objects.filter(id=self.producto.id).exists())
```

- **Resultado de la ejecución**

Ahora ya teniendo definidos los test se hizo la verificación de que todos ellos pasen satisfactoriamente con el comando **python manage.py test** en la terminal para poder visualizar que se ejecuten correctamente.



Finalmente analizando lo que arroja al ejecutar los test podremos observar que efectivamente detecta los 4 test que se hicieron en este caso y que los pasa de manera satisfactoria.

Brayan Alejandro Muñoz Pérez

- **Tipo de prueba:** Unitaria
- **Descripción del componente que se probará:**
  El servicio a probar es el módulo de generación de informes PDF. Este módulo cuenta con dos componentes esenciales:

  Función de generación del PDF:
  La función generate_pdf_report, utiliza ReportLab para crear un documento PDF que incluye un título ("Reporte de Inventario") y un listado de productos (con atributos como nombre, categoría, cantidad y precio).

  - Si el PDF se genera correctamente, se espera que el objeto retornado sea de tipo ContentFile, que contenga datos binarios no vacíos, tenga al menos una página y que el contenido incluya el texto "Reporte de Inventario".

  Vista de exportación del PDF:
  La vista export_pdf, invoca la función anterior y retorna el PDF mediante una respuesta HTTP.

  - Si la operación es exitosa, el endpoint deberá arrojar un código 200 y enviar en los headers el Content-Type como application/pdf y el Content-Disposition configurado para descargar el archivo con el nombre "reporte_inventario.pdf".
- **Herramienta o framework usado**

  Para llevar a cabo estas pruebas se hace uso de:

  Django TestCase y Client: Para configurar y simular el entorno de pruebas, realizando solicitudes HTTP a la vista.

PyPDF2: Para leer y extraer el contenido del PDF, comprobando que el documento tenga al menos una página y que incluya el texto esperado.

ReportLab: Para construir el PDF; se asume su funcionamiento a través de la validación realizada con PyPDF2.

- **Screenshot del código del test**

Se define el método setUp() de la clase de pruebas, que hereda de TestCase. Aquí se crea un objeto Client de Django, el cual se utiliza para simular las solicitudes HTTP que se realizarán en el test de la vista.

```python
from django.test import TestCase, Client
from django.core.files.base import ContentFile
from .services import generate_pdf_report
import io
import PyPDF2

class TestExportPDF(TestCase):
    def setUp(self):
        self.client = Client()
```

Se realiza la prueba unitaria de la función de generación del PDF. Se verifica que:

Se retorna un objeto del tipo ContentFile.

El contenido no es vacío.

El PDF contiene al menos una página (utilizando PyPDF2).

La primera página del PDF contiene el texto "Reporte de Inventario", asegurando que la información se generó correctamente.

```python
    def test_generate_pdf_report_content(self):
        """
        Verifica que la función generate_pdf_report devuelve un ContentFile
        que representa un PDF no vacío y con contenido esperado.
        """
        pdf_file = generate_pdf_report()
        # Verifica que se generó un ContentFile y no está vacío
        self.assertIsInstance(pdf_file, ContentFile)
        self.assertGreater(len(pdf_file.read()), 0, "El PDF está vacío")

        # Reinicia la lectura del archivo
        pdf_file.seek(0)
        # Lee el contenido del PDF usando PyPDF2
        pdf_reader = PyPDF2.PdfReader(io.BytesIO(pdf_file.read()))
        num_pages = len(pdf_reader.pages)
        self.assertGreater(num_pages, 0, "El PDF no tiene páginas")

        # Extrae texto de la primera página y verifica que contenga el título esperado
        extracted_text = pdf_reader.pages[0].extract_text()
        self.assertIn("Reporte de Inventario", extracted_text, "El título no se encontró en el PDF")
```

Se prueba la vista que expone el PDF. Se realiza una solicitud GET al endpoint export_pdf y se valida que:

La respuesta tenga un código de estado 200.

El header Content-Type sea application/pdf.

El header Content-Disposition contenga la instrucción de descarga con el nombre "reporte_inventario.pdf".

```python
def test_export_pdf_view(self):
    """
    Verifica que la vista export_pdf retorna un archivo PDF válido.
    """
    response = self.client.get('/inventario/export_pdf/')
    # Comprueba que la respuesta es exitosa
    self.assertEqual(response.status_code, 200)
    # Verifica que el tipo de contenido es application/pdf
    self.assertEqual(response['Content-Type'], 'application/pdf')
    # Verifica que se envía como attachment con el nombre adecuado
    self.assertIn('attachment; filename="reporte_inventario.pdf"', response['Content-Disposition'])
```

- **Resultado de la ejecución**

  Finalmente se realiza el test con el siguiente comando: python manage.py test inventario.tests_pdf.

```
PS C:\Users\bamp0\OneDrive\Escritorio\Proyecto\backend_djangoREST> python manage.py test inventario.tests_pdf
Found 2 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
..
----------------------------------------------------------------------
Ran 2 tests in 0.021s

OK
Destroying test database for alias 'default'...
PS C:\Users\bamp0\OneDrive\Escritorio\Proyecto\backend_djangoREST>
```

  Esto indica que ambos tests (tanto para la función de generación del PDF como para la vista) se ejecutaron correctamente y cumplen con los requerimientos establecidos. Los resultados confirman que:

  El PDF se genera correctamente, contiene la información esperada y es válido.

  La vista `export_pdf` responde de manera adecuada para permitir la descarga del archivo.

Lecciones aprendidas y dificultades

El proceso de testing realizado nos ayudó a entender con mayor profundidad la importancia de este aspecto dentro del desarrollo de código para garantizar un trabajo de calidad y confiable. En este caso en particular, el testing nos permitió entender mejor cómo debería comportarse cada función o componente ante distintas entradas, incluyendo tanto entradas válidas como inválidas, lo cual ayuda a realizar análisis más profundos del funcionamiento de nuestro código e incluso considerar nuevas soluciones.

También hay que resaltar que el proceso de testing puede llegar a ser bastante extenso e incluso lento debido a la cantidad de escenarios que hay que analizar, volviendo muy importante el hecho de tener claro cómo funciona nuestro código para evitar que este proceso tome más tiempo del necesario.
En conclusión, aunque el proceso de testing puede llegar a ser bastante extenso y complicado, su impacto en el desarrollo de software es inmenso ya que no solo nos ayuda a detectar errores, sino que también nos permite analizar el código de manera más amplia. Además, realizar pruebas de manera constante podría mejorar nuestras habilidades de desarrollo, ya que nos obliga a ser más críticos con el código que realizamos.