



UNIVERSIDAD
NACIONAL
DE COLOMBIA

Universidad Nacional de Colombia - sede Bogotá
Facultad de Ingeniería
Departamento de Sistemas e Industrial
Curso: Ingeniería de Software 1 (2016701)

Profesor: Oscar Eduardo Alvarez Rodriguez

Estudiante(s):

Nelson Ivan Castellanos Betancourt

Angel Santiago Avendaño Cañon

David Sebastian Hurtado Sanchez

Nicolas Zuluaga Galindo

Introducción

El testeo de software es un proceso esencial en el desarrollo de aplicaciones, ya que garantiza que cada componente funcione de manera correcta y segura. Probar el sistema permite identificar errores de manera oportuna, reducir fallos futuros y asegurar que el producto final cumpla con los estándares de calidad. Además, facilita el mantenimiento y brinda mayor confianza en el rendimiento de la aplicación.

En este proyecto, el backend se desarrolla utilizando Django, un framework de Python conocido por su robustez y eficiencia en la creación de aplicaciones web. Django, junto con Django REST Framework (DRF), proporciona herramientas integradas que facilitan la creación de pruebas automatizadas, permitiendo validar el comportamiento de las vistas, modelos y APIs.

Solo a través de un proceso de testeo adecuado es posible garantizar el desarrollo de proyectos de calidad. Las pruebas aseguran que el sistema sea estable, funcional y preparado para su implementación en un entorno real.

Testing

Módulo/ Componente	PreRegistrationCompleteSerializer
Integrante	Nelson Ivan Castellanos Betancourt
Tipo de prueba realizada	Unit Tests
Descripción breve del componente probado	<p>Este serializer, parte de la aplicación de PreRegistrations, se encarga de la creación y validación de objetos PreRegistration en conjunto con datos anidados de <code>patient</code>, <code>medical_info</code> y, de manera opcional, <code>third_party</code>. Implementa lógica personalizada en el método <code>to_internal_value</code> para garantizar que se incluyan los campos requeridos y, posteriormente, en el método <code>create</code> para:</p> <ol style="list-style-type: none">1. Determinar si se reutiliza un paciente existente (si se envía su <code>id</code>) o si se crea un paciente nuevo.2. Verificar que la información del paciente (<code>patient</code>) e información médica (<code>medical_info</code>) contenga todos los campos necesarios.3. Crear (y asociar) la información de un tercero (<code>third_party</code>) en caso de proporcionarse, validando también sus campos obligatorios. <p>En caso de detectar datos incompletos, el serializer devolverá errores de validación, asegurando la integridad de la información al momento de crear un Pre-registro.</p>
Framework usado	Django Rest Framework
Código del test	Se crearon en total 8 unit tests que validan cada uno de los posibles caminos del serializer. El screenshot a continuación contiene los primeros. Ver los tests completos en test.py

```

1 from django.test import TestCase
2 from rest_framework.exceptions import ValidationError
3
4 from common.serializers import PreRegistrationCompleteSerializer
5
6 from pre_registrations.models import PreRegistration
7 from patients.models import Patient
8
9
10 class PreRegistrationCompleteSerializerTests(TestCase):
11     def setUp(self):
12         self.existing_patient = Patient.objects.create(
13             first_name="Paciente",
14             last_name="Existente",
15             birth_date="1980-01-01",
16             id_type="CC",
17             id_number="11111111",
18             contact_number="123456789",
19             email="existente@example.com",
20         )
21
22     def test_create_without_patient_data(self):
23         """
24         If no patient data is sent,
25         it should throw validation error.
26         """
27         data = {"status": PreRegistration.PENDING}
28         serializer = PreRegistrationCompleteSerializer(data=data)
29         with self.assertRaises(ValidationError) as context:
30             serializer.is_valid(raise_exception=True)
31         self.assertIn("patient", context.exception.detail)
32         self.assertEqual(
33             context.exception.detail["patient"], "Patient data is required."
34         )
35
36     def test_invalid_patient_missing_required_fields_without_id(self):
37         """
38         If send `patient` data without `id`, and missing other required fields to create a `patient`
39         it should throw validation errors.
40         """
41         data = {
42             "patient": {"first_name": "Juan"}, # Missing other required fields
43             "status": PreRegistration.PENDING,
44         }
45         serializer = PreRegistrationCompleteSerializer(data=data)
46         with self.assertRaises(ValidationError) as context:
47             serializer.is_valid(raise_exception=True)
48         self.assertIn("patient", context.exception.detail)
49         self.assertIn(
50             "Missing required fields", str(context.exception.detail["patient"])
51         )
52
53     def test_invalid_third_party_missing_required_fields(self):
54         """
55         If send `third_party` data, but missing fields,

```

Resultado de la ejecución

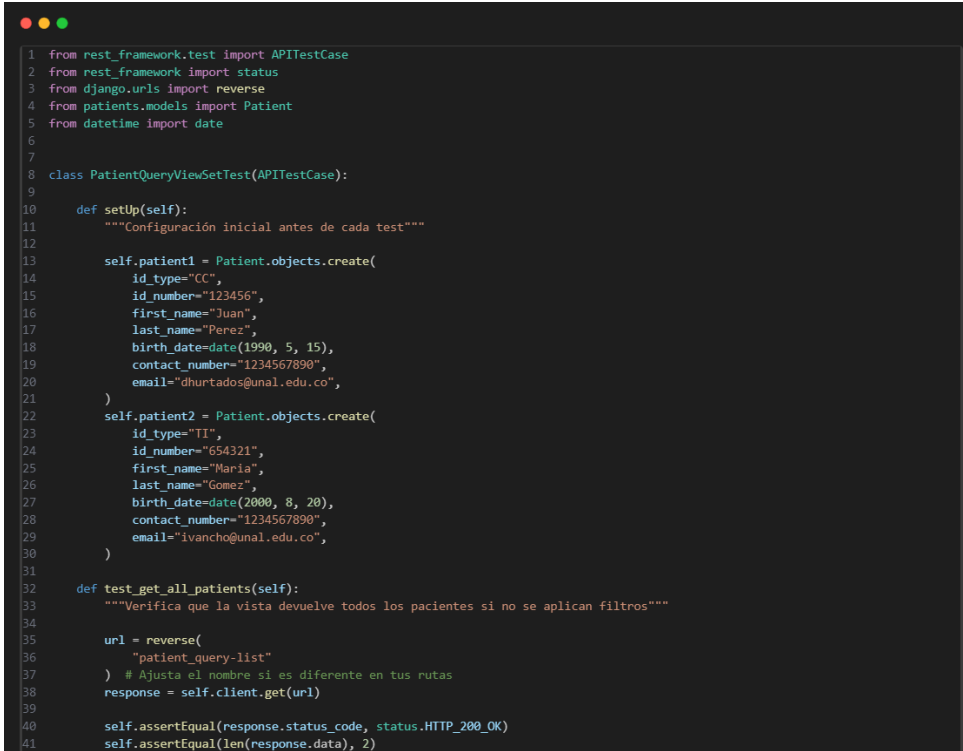
```

.env ~/dev/IngenieriaSoftwareI/Proyecto/django-backend/centro_medico git:(53-taller-3-testing---individual)±3 (0.459s)
python manage.py test
Found 8 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.....
Ran 8 tests in 0.004s

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

.env ~/dev/IngenieriaSoftwareI/Proyecto/django-backend/centro_medico git:(53-taller-3-testing---individual)±2
python manage.py test ↵

```

Módulo/ Componente	PatientQueryViewSet
Integrante	David Sebastian Hurtado Sanchez
Tipo de prueba realizada	Unit Tests
Descripción breve del componente probado	<p>El componente PatientQueryViewSet gestiona la consulta y filtrado de pacientes, extendiendo ListModelMixin y GenericViewSet para manejar listados con opciones de búsqueda y filtrado. Permite recuperar todos los pacientes registrados en la base de datos y utiliza el PatientQuerySerializer para transformar los datos en respuestas JSON. Implementa filtros específicos basados en id_type y id_number, permitiendo encontrar pacientes con su tipo y número de identificación. Además, habilita la búsqueda por first_name y last_name para facilitar la identificación cuando no se tiene el número exacto. El método <code>get_queryset(self)</code> sobrescribe la consulta para aplicar los filtros si se proporcionan en la URL, devolviendo un paciente específico o la lista completa según los parámetros recibidos. Este componente es crucial en la API de pacientes, ya que optimiza la búsqueda y recuperación de información.</p>
Framework usado	Django Rest Framework
Código del test	<p>Se crearon en total 3 unit tests que validan cada uno de los posibles caminos del serializer. El screenshot a continuación contiene los primeros. Ver los tests completos en tests.py</p>  <pre> 1 from rest_framework.test import APITestCase 2 from rest_framework import status 3 from django.urls import reverse 4 from patients.models import Patient 5 from datetime import date 6 7 8 class PatientQueryViewSetTest(APITestCase): 9 10 def setUp(self): 11 """Configuración inicial antes de cada test""" 12 13 self.patient1 = Patient.objects.create(14 id_type="CC", 15 id_number="123456", 16 first_name="Juan", 17 last_name="Perez", 18 birth_date=date(1990, 5, 15), 19 contact_number="1234567890", 20 email="dhurtados@unal.edu.co", 21) 22 self.patient2 = Patient.objects.create(23 id_type="TI", 24 id_number="654321", 25 first_name="Maria", 26 last_name="Gomez", 27 birth_date=date(2000, 8, 20), 28 contact_number="1234567890", 29 email="ivancho@unal.edu.co", 30) 31 32 def test_get_all_patients(self): 33 """Verifica que la vista devuelve todos los pacientes si no se aplican filtros""" 34 35 url = reverse(36 "patient_query-list" 37) # Ajusta el nombre si es diferente en tus rutas 38 response = self.client.get(url) 39 40 self.assertEqual(response.status_code, status.HTTP_200_OK) 41 self.assertEqual(len(response.data), 2) </pre>

Resultado de la ejecución

```
(venv) PS C:\Users\LENOVO\Desktop\materias actuales\ing de software\Proyecto Final\IngenieriaSoftware1\Proyecto\django-backend\centro_medico> python manage.py test
Found 11 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.....
Ran 11 tests in 0.029s

OK
Destroying test database for alias 'default'...
(venv) PS C:\Users\LENOVO\Desktop\materias actuales\ing de software\Proyecto Final\IngenieriaSoftware1\Proyecto\django-backend\centro_medico>
```

Módulo/ Componente	LoginAPIView
Integrante	Nicolás Zuluaga Galindo
Tipo de prueba realizada	Prueba unitaria
Descripción breve del componente probado	<p>La lógica de autenticación se basa en el uso de LoginAPIView, que procesa solicitudes de tipo POST, verifica las credenciales del usuario utilizando el método authenticate() de Django y maneja el inicio de sesión con la función login(). Las respuestas están estructuradas en formato JSON, siguiendo las mejores prácticas del framework Django REST Framework (DRF).</p> <p>Teniendo en cuenta lo anterior, se implementaron y ejecutaron pruebas unitarias para el componente de autenticación de usuarios. El objetivo fue validar el correcto funcionamiento del proceso de inicio de sesión, evaluando diferentes escenarios: credenciales válidas, credenciales incorrectas, campos incompletos (sin email o contraseña) y validación del estado de inactividad del usuario.</p>
Framework usado	Django Rest Framework

Código del test

```

1  from rest_framework.test import APITestCase
2  from django.urls import reverse
3  from django.contrib.auth.models import User
4  from rest_framework import status
5
6
7  class UserLoginTests(APITestCase):
8      def setUp(self):
9          # Crear un usuario de prueba con credenciales válidas
10         self.user = User.objects.create_user(
11             username="testuser",
12             email="testuser@example.com",
13             password="testpass123",
14             is_active=True # Marcado como activo
15         )
16         self.inactive_user = User.objects.create_user( # 🛑 Usuario inactivo
17             username="inactiveuser",
18             email="inactiveuser@example.com",
19             password="inactivepass123",
20             is_active=False # Marcado como inactivo
21         )
22         self.login_url = reverse('login')
23
24     # 🔑 Inicio de sesión exitoso con credenciales válidas
25     def test_login_successful(self):
26         data = {
27             "email": "testuser@example.com",
28             "password": "testpass123",
29         }
30         response = self.client.post(self.login_url, data)
31         self.assertEqual(response.status_code, status.HTTP_200_OK)
32         self.assertIn("email", response.data)
33         self.assertEqual(response.data["email"], "testuser@example.com")
34
35     # ❌ Inicio de sesión fallido con credenciales incorrectas
36     def test_login_wrong_credentials(self):
37         data = {
38             "email": "testuser@example.com",
39             "password": "wrongpassword",
40         }
41         response = self.client.post(self.login_url, data)
42         self.assertEqual(response.status_code, status.HTTP_400_BAD_REQUEST)
43         self.assertEqual(response.data["detail"], "Bad credentials.")
44
45     # 🚫 Login con campos incompletos (sin contraseña)
46     def test_login_missing_password(self):
47         data = {
48             "email": "testuser@example.com",
49         }
50         response = self.client.post(self.login_url, data)
51         self.assertEqual(response.status_code, status.HTTP_400_BAD_REQUEST)
52         self.assertIn("password", response.data)
53
54     # 🚫 Login con campos incompletos (sin email)
55     def test_login_missing_email(self):
56         data = {
57             "password": "testpass123",
58         }
59         response = self.client.post(self.login_url, data)
60         self.assertEqual(response.status_code, status.HTTP_400_BAD_REQUEST)
61         self.assertIn("email", response.data)
62
63     # 🛑 Login fallido con usuario inactivo
64     def test_login_inactive_user(self):
65         data = {
66             "email": "inactiveuser@example.com",
67             "password": "inactivepass123",
68         }
69         response = self.client.post(self.login_url, data)
70         self.assertEqual(response.status_code, status.HTTP_400_BAD_REQUEST)
71         self.assertEqual(response.data["detail"], "Bad credentials.")
72

```

Resultado de la ejecución	<pre>(venv) D:\Nicolás Zuluaga\99. U\Semestre 16\IngeSoft\Proyecto\IngenieriaSoftwareI\Proyecto\django-backend\cent(venv) D:\Nicolás Zuluaga\99. U\Semestre 16\IngeSoft\Proyecto\IngenieriaSoftwareI\Proyecto\django-backend\centro_medico>python manage.py test Found 5 test(s). Creating test database for alias 'default'... System check identified no issues (0 silenced). Ran 5 tests in 8.153s OK Destroying test database for alias 'default'...</pre>
----------------------------------	--

Módulo/ Componente	OccupancyTestCase
Integrante	Angel Avendaño
Tipo de prueba realizada	Unit Tests
Descripción breve del componente probado	El conjunto de pruebas valida la creación y asociación correcta de registros en los modelos de ocupación y registros médicos, incluyendo ResourceUsage, MedicalCenterCapacity, OccupancyHistory y ResourceType. Se asegura de que los objeto se creen con los valores esperados y las relaciones adecuadas
Framework usado	Django Rest Framework
Código del test	Se implementaron pruebas unitarias para la verificación de la creación de diferentes modelos relacionados con la ocupación de recursos en el centro médico

```

from django.test import TestCase
from django.utils import timezone
from occupancy.models import ResourceType, MedicalCenterCapacity, ResourceUsage, OccupancyHistory
from patients.models import PatientAdmission, Patient
from pre_registrations.models import PreRegistration, PreRegistrationMedicalInfo, ThirdParty

class OccupancyTestCase(TestCase):
    def setUp(self):
        # Create a ResourceType
        self.resource_type = ResourceType.objects.create(resource_name="ICU Bed")

        # Create a Patient
        self.patient = Patient.objects.create(
            first_name="John",
            last_name="Doe",
            birth_date="1990-01-01",
            id_type="Passport",
            id_number="A123456789",
            contact_number="1234567890",
            email="john.doe@example.com",
        )

        self.medical_info = PreRegistrationMedicalInfo.objects.create(
            reason="Routine Checkup",
        )

        self.third_party = ThirdParty.objects.create(
            relationship="Spouse",
            first_name="Jane",
            last_name="Doe",
            contact_number="0987654321"
        )

        # Create PreRegistration with medical_info
        self.pre_registration = PreRegistration.objects.create(
            patient=self.patient,
            medical_info=self.medical_info,
            third_party=self.third_party,
            status="pending"
        )

        # Create a PatientAdmission
        self.admission = PatientAdmission.objects.create(
            pre_registration=self.pre_registration,
            admission_type="hospitalization",
            triage_level=5,
            admission_date=timezone.now()
        )

    def test_create_resource_usage(self):
        """Test creating a ResourceUsage record"""
        usage = ResourceUsage.objects.create(
            admission=self.admission,
            resource_type=self.resource_type,
            resource_quantity=3,
            start_time=timezone.now()
        )
        self.assertEqual(usage.resource_quantity, 3)
        self.assertEqual(usage.admission, self.admission)
        self.assertEqual(usage.resource_type, self.resource_type)

    def test_create_medical_center_capacity(self):
        """Test creating a MedicalCenterCapacity record"""
        capacity = MedicalCenterCapacity.objects.create(
            resource_type=self.resource_type,
            total_quantity=10
        )
        self.assertEqual(capacity.total_quantity, 10)
        self.assertEqual(capacity.resource_type, self.resource_type)

    def test_create_occupancy_history(self):
        """Test creating an OccupancyHistory record"""
        history = OccupancyHistory.objects.create(
            resource_type=self.resource_type,
            occupied_quantity=5,
            occupancy_percentage=50.00,
            created_at=timezone.now()
        )
        self.assertEqual(history.occupied_quantity, 5)
        self.assertEqual(float(history.occupancy_percentage), 50.00)
        self.assertEqual(history.resource_type, self.resource_type)

    def test_create_resource_type(self):
        """Test creating a ResourceType record"""
        resource = ResourceType.objects.create(resource_name="Ventilator")
        self.assertEqual(resource.resource_name, "Ventilator")

```

Resultado de la ejecución	<pre>(Ingesoft) ~\WebstormProjects\IngenieriaSoftwareI git:[53-taller-3-testing---individual] Found 15 test(s). Creating test database for alias 'default'... System check identified no issues (0 silenced). ----- Ran 15 tests in 0.053s OK Destroying test database for alias 'default'...</pre>

Lecciones aprendidas y dificultades

La ejecución de estos tests nos ha permitido mejorar la lógica del Serializer, haciéndolo más robusto y consistente. Concretamente los unit tests nos ha permitido darnos cuenta de algunos fallos en las validaciones, otras inexistentes y una mejora en la lógica en la creación del pre-registro. Nos ayudaron a encontrar errores que no eran manejados por el framework y terminarían siendo HTTP 500.

Estos tests aportan una alta robustez al Serializer, asegurando la calidad y consistencia de los datos procesados. Además, nos permitió refactorizar partes del código para mejorar su mantenibilidad y reducir las redundancias. También nos dimos cuenta de que algunas validaciones no estaban cubriendo todos los edge cases

Por otro lado, la ejecución de estos tests nos ha permitido verificar el correcto funcionamiento del **PatientQueryViewSet**, asegurando que los filtros por tipo y número de identificación operen de manera adecuada. Durante el proceso, los unit tests nos ayudaron a identificar problemas en la lógica de filtrado, como la necesidad de manejar casos en los que los parámetros proporcionados no coincidían con datos en la base de datos. Además, nos permitieron comprobar que las respuestas de la API fueran consistentes y cumplieran con los estándares esperados, evitando respuestas incorrectas o vacías cuando no correspondía.

Uno de los desafíos principales fue garantizar que el endpoint respondiera correctamente a diferentes combinaciones de filtros y que los errores en las consultas no generaran fallos inesperados. También nos aseguramos de que la API devolviera códigos de estado HTTP correctos, evitando errores como **500 Internal Server Error** cuando se proporcionaban datos inválidos.

Por último, aprendimos que los tests también sirven como documentación viva del sistema, permitiendo a todo el equipo rápidamente como se espera que funcionen los diferentes módulos sin necesidad de leer todo el código.

Conclusión: Los tests son fundamentales para mejorar la estabilidad y precisión de nuestras funcionalidades. No solo nos ayudan a detectar errores antes de que lleguen a producción, sino que también nos permiten optimizar el código, mejorar su rendimiento y asegurar que los cambios futuros no rompan funcionalidades existentes. La automatización de pruebas debe ser una práctica continua dentro del desarrollo.

NOTA: Este documento fue elaborado con el apoyo de herramientas de inteligencia artificial, incluyendo ChatGPT, para optimizar la claridad y la precisión de la información presentada. Las ideas y contenido reflejan un trabajo colaborativo entre las capacidades tecnológicas y el criterio de los autores.