El futuro digital es de todos

MinTIC

Ciclo 3:

Desarrollo de Software

UNIVERSIDAD NACIONAL DE COLOMBIA

Misión TIC 2022

VERSIÓN 1.0

Unidad de educación continua y permanente
Facultad de Ingeniería

Unidad Camilo Torres
Calle 44 # 45-67
Bloque B5 piso 1

(57) + 316 5000
uec_fibog@unal.edu.co

Pruebas Unitarias

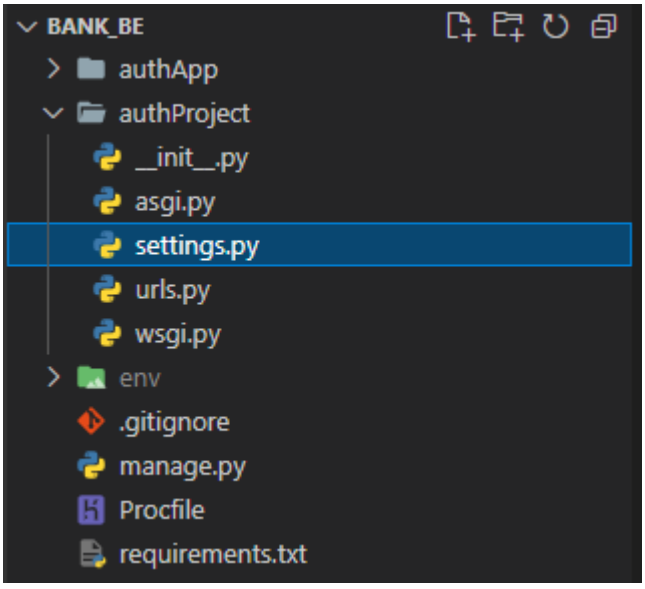
Actividad Práctica

El concepto de *prueba unitaria* es agnóstico a cualquier tecnología, lenguaje o *framework*, y esto genera que de acuerdo con el entorno de trabajo se manejen ciertos parámetros y conceptos propios en ese escenario. El objetivo de esta guía será identificar las características y particularidades de una prueba unitaria en *Django Rest*. Para ello se crearán y ejecutarán una serie de *pruebas* para los *endpoints* de los servicios del componente de back-end *bank_be* (correspondiente a la capa lógica).

Configuraciones Preliminares

Una de las grandes ventajas que ofrece *Django Rest* es la posibilidad de ejecutar pruebas sin afectar los *recursos* disponibles, es decir que los cambios realizados en las pruebas no tienen efecto sobre la aplicación o sobre los recursos que usa. Dentro de dichos recursos se encuentran las *bases de datos* principalmente. Esto permite que se puedan ejecutar las pruebas en cualquier momento sin modificar el estado de los datos del sistema.

Para hacer posible la ejecución de las pruebas, sin modificar el estado, es necesario realizar una serie de cambios sobre las configuraciones del proyecto, es decir sobre el archivo *authProject/settings.py*.



A continuación, se presentan los cambios que se deben realizar.

Conexión con la Base de Datos

Cuando se hace uso de la aplicación, *Django Rest* establece una conexión robusta con la *base de datos real*, con el objetivo de mantener el estado de los recursos intacto, es decir que las pruebas unitarias no tengan efecto sobre los datos reales. *Django Rest* permite crear una *réplica* de la base de datos real, y sobre esta replica realiza las pruebas. La réplica es creada y desechada cada vez que se realiza una prueba. Para establecer dicha replica se debe agregar a la siguiente instrucción *'TEST': {'MIRROR': 'default',},* en la sección de *DATABASES*:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'mydatabasename',
        'USER': 'myuser',
        'PASSWORD': 'mypassword',
        'HOST': 'myhost',
        'PORT': '5432',
        'TEST': {'MIRROR': 'default',},
    }
}
```

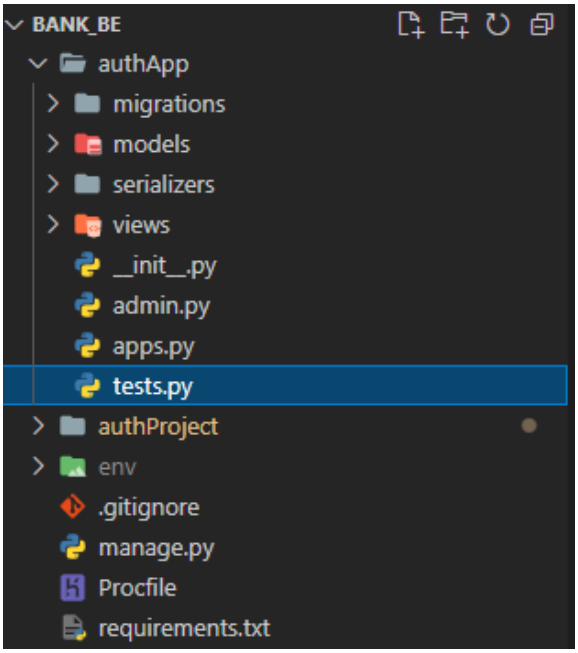
Establecer Formato

Django Rest es un *framework* flexible capaz de trabajar con distintos formatos, pero en este caso en específico se usará por defecto el formato *JSON*, esto permitirá tener salidas y entradas en un formato ya conocido. Para establecer el formato *JSON* como el formato por defecto se debe agregar la instrucción, *'TEST_REQUEST_DEFAULT_FORMAT': 'json'* a la sección de *REST_FRAMEWORK*:

```
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': (
        'rest_framework.permissions.AllowAny',
    ),
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework_simplejwt.authentication.JWTAuthentication',
    ),
    'TEST_REQUEST_DEFAULT_FORMAT': 'json'
}
```

Creación de la Prueba

Al momento de crear una aplicación (*authApp*, por ejemplo) dentro de un proyecto, *Django Rest* crea un archivo llamado *tests.py* para la definición de las pruebas unitarias. Si bien en un proyecto de mayor dimensión se debe hacer uso de un *módulo*, para el tamaño de la aplicación será ideal utilizar únicamente un *archivo*.



Dentro del archivo se definirá una clase llamada *TestAPI* que extenderá de la clase base *TestCase*. Esta clase es provista por *Django* y permite definir una serie de *métodos* que como tal serán los *tests* o pruebas unitarias. El objetivo será crear pruebas unitarias para los *endpoints* de *registro*, *autenticación* y *obtener la información del usuario*. A continuación, se detallarán cada uno de los fragmentos de código que deberán ir dentro del archivo *authApp/tests.py*.

Importaciones

A lo largo de las distintas pruebas se usarán algunos paquetes de *Python* y *Django Rest*. Estos paquetes serán importados con las siguientes líneas de código:

```
import json
import jwt

from django.test import TestCase
```

```
from rest_framework import status
from rest_framework.test import APIClient
```

Las librerías *json* y *jwt* son provistas por *Python* y facilitan el manejo de algunos objetos usados en los *tests*. La librería *status* y *APIClient* permitirán manejar las pruebas sobre los endpoints, es decir, *Django Rest* permitirá crear un cliente muy ligero con *APIClient*, este realizará peticiones sobre los *endpoints* que se desean probar y sus resultados podrán ser comparados usando *status*. Por último, la librería *TestCase* permitirá agrupar los *tests* creados.

Prueba #1

El primer *endpoint* o servicio del componente *bank_be* que se probará es el *registro*. Para esto se solicitará la creación de un usuario y luego se verificará si la solicitud fue realizada de manera correcta. El fragmento de código referente a esta prueba (archivo *authApp/tests.py*) es el siguiente:

```
class TestAPI(TestCase):

    def test_signUp(self):
        client = APIClient()
        response = client.post(
            '/user/',
            {
                "username": "user_prueba_1",
                "password": "password_prueba_1",
                "name": "user prueba",
                "email": "user_prueba_1@misionTIC.com",
                "account": {
                    "lastChangeDate": "2021-09-23T10:25:43.511Z",
                    "balance": 20000,
                    "isActive": "true"
                }
            },
            format='json')

        self.assertEqual(response.status_code, status.HTTP_201_CREATED)
        self.assertEqual('refresh' in  response.data.keys(), True)
        self.assertEqual('access' in  response.data.keys(), True)
```

Lo primero que se realiza es la definición de la clase *TestAPI*, extendida de *TestCase*. Esta permitirá agrupar

los *tests*. Luego se define el método *test_signUp()* para el *test*. Primero se crea un cliente ligero a partir de la clase *APIClient*, luego con este cliente definido se realiza una petición *post* sobre la ruta *'/user/'*, esta petición debe recibir un cuerpo de tipo *json*. Luego de que sea procesada la petición, se recibirá una respuesta la cual contendrá mucha información relevante, entre ella el *status* de la petición y la *data* recibida. Con ayuda de un *assertEqual* se verificará si el *estatus* es el esperado (*201* indica que el recurso fue creado) y si la información recibida es la *esperada* (los tokens *refresh* y *access*).

Una *observación* interesante sobre este primer *test* es que los datos ingresados en el cuerpo de la petición deben corresponder a un *usuario* que *no exista*, en caso contrario se generará un *error* y la prueba *fallará*. La prueba se puede ejecutar las veces que se desee, ya que cuando se crea el usuario durante la prueba, el usuario no queda registrado en la base de datos real, sino en la *replica*, la cual es desechada al finalizar la prueba.

Prueba #2

La segunda prueba buscará probar el *endpoint* correspondiente al servicio de *autenticación LogIn*. En este test se realizará una solicitud para poder autenticar con las credenciales de un *usuario previamente creado* (no es válido el usuario creado en la prueba anterior, este usuario no es registrado en los datos reales). El fragmento de código referente a esta prueba es el siguiente:

```
def test_login(self):
    client = APIClient()
    response = client.post(
        '/login/',
        {
            "username":"user_existente",
            "password":"password_existente"
        },
        format='json')

    self.assertEqual(response.status_code, status.HTTP_200_OK)
    self.assertEqual('refresh' in  response.data.keys(), True)
    self.assertEqual('access' in  response.data.keys(), True)
```

Es importante mantener la *indentación* en este fragmento de código ya que *Python* es sensible a los espacios, la estructura de la prueba es muy similar al anterior, se crea un cliente con *APIClient* con el cual se realiza una petición de tipo *get* al endpoint *'/login/'*, con su correspondiente cuerpo. Luego con ayuda de *assertEquals*, se evalúa si la información recibida es la esperada.

Prueba #3

Por último, se creará una prueba para comprobar el funcionamiento del servicio que permite obtener la información del usuario. Esta prueba es un poco compleja, ya que, para obtener la *información del usuario*, se necesita obtener un *token* de *acceso*, el cual se agrega en los *headers* de la petición. Además, se debe construir la ruta de la petición de acuerdo con el *user_id*, el cual está contenido en el *payload* del *token*, por lo cual se deberá realizar un proceso de *descifrado* del token.

Para el funcionamiento de esta prueba es necesario tener un *token de acceso*, el cual se consigue usando el servicio de *autenticación LogIn* (servicio probado en la prueba anterior). En este punto se debe tener en cuenta que los *tests* se ejecutan de manera *secuencial*, es decir, si el primer *test* falla, los demás no se ejecutan. En este caso, primero se ejecuta el *test* de *SignUp*, luego el test de *LogIn* y por último el *test* actual, esto permite deducir que, si se llega a ejecutar el *test* actual, es porque los *tests* anteriores se ejecutaron de manera correcta. Esto permite usar con total confianza las peticiones probadas en los *tests* anteriores.

El fragmento de código correspondiente a la tercera prueba es el siguiente:

```
def test_user(self):
    client = APIClient()

    token_access =  client.post(
                        '/login/',
                        {"username":"user_existente", "password":"password_existente"},
                        format='json'
                    ).data["access"]


    secret = 'django-insecure-a$sbxk7p)#ok=yf+%_$^_xfo=9ogzaccn#hgadkiyzjpik4(33'
    user_id =   jwt.decode( token_access,  secret,  algorithms=["HS256"] )["user_id"]


    url = '/user/'+ str(user_id) + '/'
    auth_headers = {'HTTP_AUTHORIZATION': 'Bearer ' + token_access,}


    response = client.get(url,  **auth_headers)

    self.assertEqual(response.status_code, status.HTTP_200_OK)
    self.assertEqual(response.data["username"], "user_existente")
```

Es importante mantener la *indentación* en este fragmento de código ya que *Python* es sensible a los espacios. Existen varios detalles importantes en este fragmento de código:

- Dado que el servicio de *LogIn* ya fue probado, se realiza una petición con el fin de obtener únicamente el *token de acceso*, el cual se agrega a los *headers* de la petición con ayuda del objeto *auth_headers*.

- Una vez se tiene el *token de acceso* se realiza el proceso de *decodificación* para obtener el *id* del *usuario* (*user_id*), con este *user_id* se puede construir la *ruta* de la petición.

- El *secret* es un valor necesario para el proceso de *cifrado* y *descifrado*, este también es definido en los *settings* del proyecto.

- Una vez se tiene la *ruta* y los *headers* se construye la *petición* y se evalúa la respuesta al igual que en los otros *test*.

Posible Error

En muchos casos las referencias a la librería de *jwt* se encuentran *desactualizadas*, por ello desde la terminal que ejecuta el *entorno de Python*, se debe ejecutar el siguiente *comando* para evitar errores futuros:

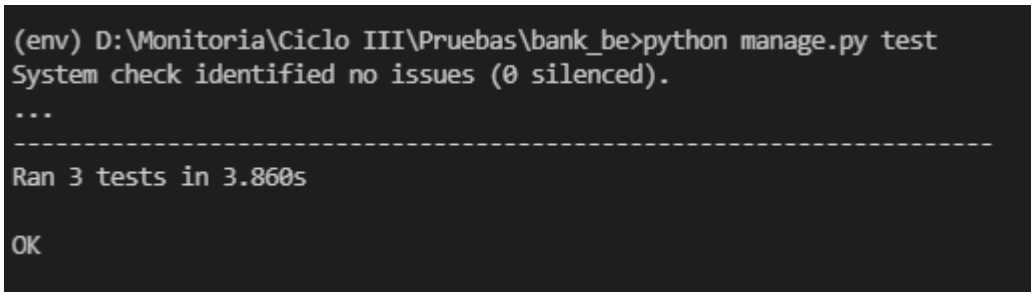
```
pip install PyJWT==2.1.0
```

Run Test

Una vez se han creado los distintos *tests*, estos se pueden ejecutar desde una terminal (con el entorno de Python activo) con el siguiente *comando*:

```
python manage.py test
```

Esto iniciará el proceso de *testing*, es posible que tarde un poco. Al finalizar se obtendrá lo siguiente:



En este caso se *aprobaron* todos los *tests*, por lo cual se ha asegurado que la *aplicación* funciona

correctamente. Si alguno de los tests *falla*, se detendrá el testing y se mostrarán los motivos del fallo.

Nota: de ser necesario, en el material de la clase se encuentra un archivo *C3.AP.18. bank_be.rar*, con el desarrollo de las pruebas realizadas en esta guía.