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@unaleduco

Componente Lógico

(Vistas y URLs)

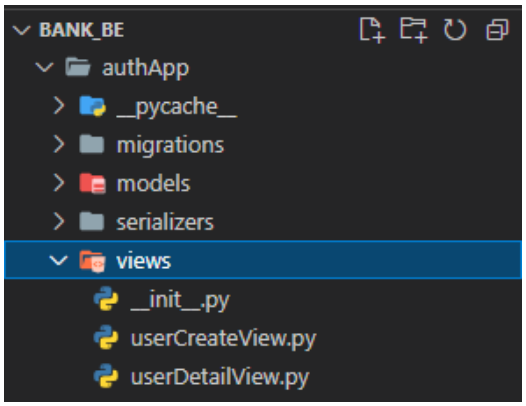
Actividad Práctica

En esta guía se desarrollarán las vistas del componente lógico *bank_be* y se expondrán sus funcionalidades en *URLs* del servidor de Django. Las funcionalidades que se implementarán son las siguientes:

1. **Registrar un usuario en el sistema:** se desarrollará una funcionalidad que reciba la información de un nuevo usuario y cree la cuenta en el sistema. Si la creación se hace correctamente, la funcionalidad debe retornar un *JSON* con el *access token* y el *refresh token* asociado a la cuenta creada.
2. **Iniciar sesión:** se desarrollará una funcionalidad que reciba las credenciales de un usuario y, si este existe en el sistema, se retornará un *JSON* con el *access token* y el *refresh token* asociado a esta cuenta.
3. **Obtener la información de un usuario que ha iniciado sesión:** se desarrollará una funcionalidad que reciba el *access token* y el *id* del usuario, y que retorne la información de dicho usuario. En esta funcionalidad el *access token* se utilizará para verificar que el usuario que realiza la petición esté autenticado, y para controlar que el usuario autenticado pueda consultar únicamente la información de su cuenta, pues un usuario no debería ser capaz de consultar la información de otro.
4. **Renovar el access token:** se desarrollará una funcionalidad que reciba un *refresh token* asociado a una cuenta y que retorne un *access token* asociado a la misma cuenta.

Las vistas correspondientes a las funcionalidades 1 y 4 ya se encuentran implementadas por el paquete *Simple JWT*, lo que implica que sólo es necesario asignar una URL del servidor a cada vista. Sin embargo, las vistas correspondientes a las funcionalidades 2 y 3 deben ser desarrolladas desde cero, por lo cual, para facilitar el desarrollo se utilizarán las vistas implementadas por Django REST Framework.

Para empezar, se deben crear las carpetas y/o archivos necesarios para tener la siguiente estructura:



Dentro de *authApp* se debe tener el módulo *views* con su respectivo *__init__.py*, y con dos archivos donde se implementarán cada una de las vistas.

UserCreateView

Para implementar la funcionalidad de registro de un usuario se creará la clase *UserCreateView*. Esta abstraerá de *APIView*, una clase implementada por Django REST que contiene la configuración inicial necesaria para que Django perciba a cualquier clase como una vista. *APIView* no contiene la implementación de ninguna funcionalidad, por lo cual se utiliza únicamente en las clases en las que se realizará toda la implementación.

El código correspondiente a la *UserCreateView* que debe ir en el archivo *views/userCreateView.py* es el siguiente:

```
from rest_framework import status, views
from rest_framework.response import Response
from rest_framework_simplejwt.serializers import TokenObtainPairSerializer

from authApp.serializers.userSerializer import UserSerializer

class UserCreateView(views.APIView):

    def post(self, request, *args, **kwargs):
        serializer = UserSerializer(data=request.data)
        serializer.is_valid(raise_exception=True)
        serializer.save()

        tokenData = {"username":request.data["username"],
                     "password":request.data["password"]}
        tokenSerializer = TokenObtainPairSerializer(data=tokenData)
        tokenSerializer.is_valid(raise_exception=True)

        return Response(tokenSerializer.validated_data, status=status.HTTP_201_CREATED)
```

Como se puede observar, la implementación consta de la definición de la clase *UserCreateView* (que abstrae de *APIView*), y de la definición del método *post*. En Django, cuando una vista recibe una *petición HTTP*, esta se encarga de ejecutar una función u otra de acuerdo con el *método HTTP* de la petición. Para indicar qué función se debe ejecutar con cada *método HTTP*, se deben nombrar las funciones de acuerdo con el método al que se asocian. Es por ello, que el método definido en el código tiene como nombre *post*, pues de esta forma, esta función se ejecuta cuando una *petición HTTP* llega a la vista con el método *POST*.

Todos los parámetros y la información que contiene la *petición HTTP* llegan a la función por medio de sus argumentos (*self*, *request*, **args*, y ***kwargs*). Cada función ejecuta un código u otro dependiendo de su

propósito, en este caso, la función *post* se encarga de tomar los datos de la *petición HTTP*, verificar que siguen el formato que el *UserSerializer* requiere, y crear el usuario en el sistema. Una vez lo crea, inicia la sesión y retorna el *access token* y el *refresh token* al usuario.

UserDetailView

Para implementar la funcionalidad restante, es decir, la funcionalidad para obtener la información de un usuario, se creará la clase *UserDetailView*. Esta abstraerá de *RetrieveAPIView*, una clase implementada por Django REST que configura e implementa la funcionalidad para la obtención de un registro de una entidad, de acuerdo con su id.

El código correspondiente a la *UserDetailView* que debe ir en el archivo *views/userDetailView.py* es el siguiente:

```
from django.conf import settings
from rest_framework import generics, status
from rest_framework.response import Response
from rest_framework_simplejwt.backends import TokenBackend
from rest_framework.permissions import IsAuthenticated

from authApp.models.user import User
from authApp.serializers.userSerializer import UserSerializer

class UserDetailView(generics.RetrieveAPIView):
    queryset = User.objects.all()
    serializer_class = UserSerializer
    permission_classes = (IsAuthenticated,)

    def get(self, request, *args, **kwargs):

        token = request.META.get('HTTP_AUTHORIZATION')[7:]
        tokenBackend = TokenBackend(algorithm=settings.SIMPLE_JWT['ALGORITHM'])
        valid_data = tokenBackend.decode(token,verify=False)

        if valid_data['user_id'] != kwargs['pk']:
            stringResponse = {'detail':'Unauthorized Request'}
            return Response(stringResponse, status=status.HTTP_401_UNAUTHORIZED)

        return super().get(request, *args, **kwargs)
```

Como se puede observar en el código, en la clase se definen 3 atributos: *queryset*, *serializer_class* y *permission_classes*. Es indispensable definir los primeros 2 atributos (*queryset* y *serializer_class*) si se abstrae de una clase de Django REST que implemente una funcionalidad, pues estos atributos indican a la clase de Django REST, el modelo y el *Serializer* que debe utilizar dicha funcionalidad. En este caso, se le indica que utilice el modelo *Users* y el *Serializer UserSerializer*. Por otro lado, se tiene el atributo opcional *permission_classes*, este se utiliza para indicarle a la clase de DjangoREST la necesidad de verificar que el usuario que hace la *petición HTTP* esté autenticado.

Finalmente, se implementa la función *get.* La implementación original de esta función en la clase *RetrieveAPIView* se encarga de tomar el *id* del usuario, y retornar su información solo si el *access token* es válido. Sin embargo, como se indicó anteriormente, en el sistema no se permitirá a un usuario solicitar la información de otro. Por ello, en el código se añade una verificación adicional, la cual toma el *id* del usuario solicitado, y lo compara con el *id* del usuario al que representa el *access token*. Si no coinciden, se retorna un mensaje que indica al usuario que no tiene acceso. Si coinciden, se ejecuta la función original de la clase *RetrieveAPIView* (ese es el propósito de la última línea de código).

Importar vistas

Una vez se han implementado ambas vistas, estas deben agregarse al módulo para que puedan ser importadas más adelante. Para esto, se deben agregar las siguientes líneas de código en el archivo, *views/__init__.py*:

```
from .userCreateView import UserCreateView
from .userDetailView import UserDetailView
```

URLs

Como se mencionó anteriormente, para exponer las funcionalidades implementadas en las vistas a los usuarios es necesario asignar una *URL* a cada una de ellas. Para ello, se debe reemplazar el código del archivo *authProject/urls.py* con el siguiente código:

```
from django.urls import path
from rest_framework_simplejwt.views import (TokenObtainPairView, TokenRefreshView)
from authApp import views

urlpatterns = [
    path('login/', TokenObtainPairView.as_view()),
    path('refresh/', TokenRefreshView.as_view()),
    path('user/', views.UserCreateView.as_view()),
    path('user/<int:pk>/', views.UserDetailView.as_view()),
]
```

En este código se realizan los *imports* de las vistas desarrolladas y de las vistas de *Simple JWT*, y se asigna una *URL* a cada una de ellas. De esta forma, cada funcionalidad se puede acceder desde las siguientes *URLs*:

1. **Registrar un usuario en el sistema:** *http://localhost:8000/user/*.
2. **Iniciar sesión:** *http://localhost:8000/login/*.
3. **Obtener la información de un usuario:** *http://localhost:8000/user/<int:pk>/*. En esta *URL* se indica adicionalmente un parámetro de tipo entero (*int*), que recibirá la petición y cuyo nombre será *pk*. Normalmente este nombre se puede cambiar, sin embargo, debido a que se está utilizando Django REST en el desarrollo de la vista, es necesario que el nombre sea *pk*.
4. **Renovar el access token:** *http://localhost:8000/refresh/*.

Finalmente, si se ejecuta el servidor con el comando *python manage.py runserver*, no debe aparecer ningún error:

```
(env) D:\bank_be>py manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
September 23, 2021 - 10:07:48
Django version 3.2.7, using settings 'authProject.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

En la próxima sesión se realizará el despliegue y el *testing* de las rutas del servidor, donde se mostrará como se puede comprobar que las *peticiones HTTP* se ejecutan correctamente.

Nota: de ser necesario, en el material de la clase se encuentra una archivo *C3.AP.10. bank_be.rar*, con todos los avances en el desarrollo del componente realizado en esta guía.