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) 1 316 5000
uec_fibog@unaleduco

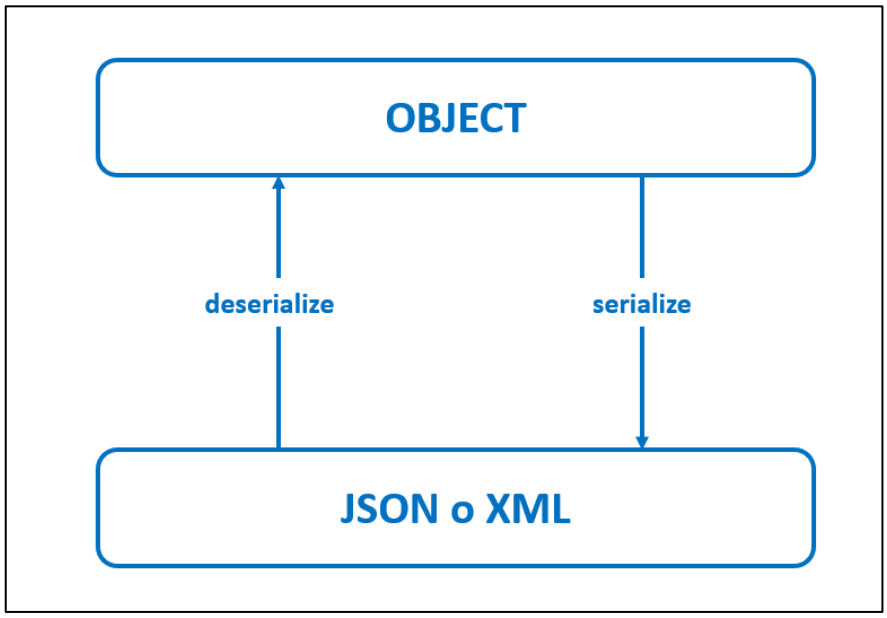
Componente Lógico

(Serializadores)

Actividad Práctica

El componente lógico: *bank_be*, esta siendo desarrollado bajo el patrón arquitectónico MVC (Modelo, Vista y Controlador). En la guía anterior se desarrollaron una serie de modelos que permitieron definir una estructura que interactúa con el componente de datos *bank_db* para brindar persistencia al sistema. En la guía de la próxima sesión se definirán una serie de vistas que permitirán al usuario o cliente interactuar con el componente. Si bien los modelos permiten al componente comunicarse con la base de datos y las vistas permiten al cliente comunicarse con el componente, no existe una interacción entre los modelos y las vistas. Es por ello por lo que estas dos partes necesitan de una tercera parte llamada *Controlador*. Este hará las veces de intermediario, pues utilizará los modelos para responder las peticiones del usuario que llegan a través de las vistas.

En muchas ocasiones el concepto de *Controlador* puede ser poco preciso en cuanto a su implementación se refiere, es por ello por lo que dentro de DjangoREST el concepto de *Controlador* es representado a través de los Serializadores (*Serializers*). Un *Serializer* es un tipo de controlador cuya función es transformar información de un formato X a un formato Y y viceversa. Por una parte, el componente le brinda persistencia a la información usando una serie de modelos (que en el fondo son objetos instanciados), y por otra parte, el usuario se comunica con el componente a través de las vistas usando un formato como JSON o XML. En este punto se puede comprender fácilmente la función del Serializer, por un lado, recibe un objeto creado a partir de la información almacenada en la base de datos y lo debe transformar en un formato comprensible para el usuario como JSON o XML, y por otro lado, debe realizar el proceso contrario, es decir, recibir cierta información por parte del usuario en formato JSON o XML y a partir de esta, crear un objeto con los modelos definidos.

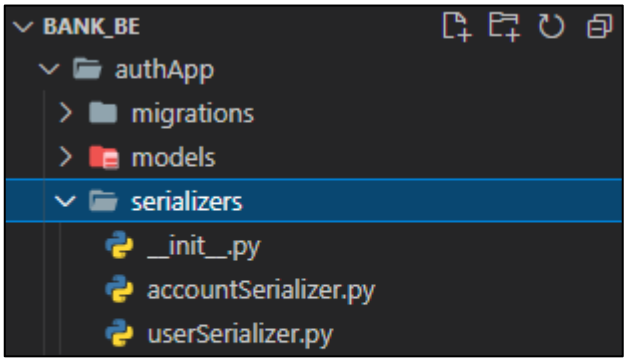


Si bien los *Serializers* se pueden construir desde cero, Django REST ofrece herramientas que facilitan la creación de estos. Una de las herramientas más útiles es la creación de un *Serializer* a partir de un Modelo, esto ahorra gran parte del trabajo ya que con solo indicarle unos pocos parámetros Django REST se encarga del trabajo difícil. En algunas ocasiones el trabajo que realiza Django REST en la creación automática de elementos como *Serializers* puedes tener imprecisiones que no se adaptan a las necesidades del componente, Django REST entiende esta problemática y ofrece la posibilidad de sobre escribir métodos y atributos para poder adaptar los elementos al componente según se requiera.

Dado que Django REST permite crear *Serializers* a partir de modelos, en muchas ocasiones se puede presentar que dichos modelos tengan relaciones entre sí. Como es de esperarse, en estos casos Django REST ofrece facilidades para realizar el proceso de serialización, por ejemplo, si se tienen dos modelos Artista y Canciones con una relación entre sí, Django REST permite que al realizar la serialización del artista se incluyan o excluyan las canciones con las que tiene relación.

Un aspecto a tener en cuenta en el desarrollo de los *Serializers* es la naturaleza del modelo *Account*, si bien a nivel de los modelos se tiene al usuario y la cuenta como dos entidades distintas, de cara al cliente final (el cliente que consumirá los servicios del componente), únicamente existe la entidad usuario y la cuenta es una parte de dicha entidad. Este patrón de diseño es bastante recurrente en el desarrollo de software, es útil cuando de cara al usuario final se quiere mantener una estructura única y completa, pero para al manejo y persistencia de datos se busca una estructura desacoplada que facilite el procesamiento de los datos. En el caso del componente se optó por dividir la entidad cuenta del usuario debido a que en muchos escenarios sería ineficiente consultar los datos personales del usuario (email, name, entre otros), cuando únicamente se necesita manipular la información financiera (balance, ultimo cambio, entre otros).

Antes de iniciar el proceso de desarrollo se debe tener la siguiente estructura de archivos:



Se debe tener un módulo llamado *serializers* con su respectivo __init__.py, y se debe tener dos archivos, uno para cada serializer.

AccountSerializer

A partir del modelo *Account* se creará el respectivo *AccountSerializer*, su objetivo principal es servir de auxiliar al *UserSerializer*. Más adelante se comprenderá el papel de *AccountSerializer* en el *UserSerializer*.

El código correspondiente al *AccountSerializer* que debe ir en el archivo *serializers/accountSerializer.py* es el siguiente:

```
from authApp.models.account import Account
from rest_framework import serializers

class AccountSerializer(serializers.ModelSerializer):
    class Meta:
        model = Account
        fields = ['balance', 'lastChangeDate', 'isActive']
```

Este código es bastante sencillo, y se puede evidenciar claramente que con solo indicar en la subclase *Meta*, el modelo y los atributos o campos que se quieren incluir, Django REST crea un *Serializer* totalmente funcional. Se puede notar además que el campo que hace de referencia al *User* en el modelo *Account* fue omitido, pues esta relación será controlada en el *UserSerializer*.

UserSerializer

El *UserSerializer* es bastante similar al *Serializer* creado anteriormente, ya que también se creará a partir de su respectivo modelo y definiendo los campos que se quieren incluir. Sin embargo, la naturaleza de *Account* previamente explicada y su relación con *User*, obliga a realizar algunas sobrescrituras de métodos para lograr el comportamiento adecuado.

El código correspondiente al *UserSerializer* que debe ir en el archivo *serializers/userSerializer.py* es el siguiente:

```
from rest_framework import serializers
from authApp.models.user import User
from authApp.models.account import Account
from authApp.serializers.accountSerializer import AccountSerializer


class UserSerializer(serializers.ModelSerializer):
    account = AccountSerializer()
    class Meta:
        model = User
        fields = ['id', 'username', 'password', 'name', 'email', 'account']

    def create(self, validated_data):
        accountData = validated_data.pop('account')
        userInstance = User.objects.create(**validated_data)
        Account.objects.create(user=userInstance, **accountData)
        return userInstance

    def to_representation(self, obj):
        user = User.objects.get(id=obj.id)
        account = Account.objects.get(user=obj.id)
        return {
                    'id': user.id,
                    'username': user.username,
                    'name': user.name,
                    'email': user.email,
                    'account': {
                        'id': account.id,
                        'balance': account.balance,
                        'lastChangeDate': account.lastChangeDate,
```

```
                'isActive': account.isActive
            }
        }
```

La primera parte de la definición del modelo y los campos a utilizar en la subclase *Meta*, es similar al caso de *AccountSerializer*, salvo que en este caso no se excluyó ningún campo, sino que además se incluye uno nuevo que es *Account*. La inclusión de este campo y su definición, con ayuda de *AccountSerializer*, permite manejar la relación entra ambas entidades.

Al tratarse de un caso bastante especifico en el que Django REST no puede automatizar el proceso de serialización y deserialización, se deben sobrescribir dos métodos para definir el comportamiento adecuado. El primer método a sobrescribir es *create*, este método crea el usuario con su respectiva cuenta simultáneamente, esto permite que a nivel del usuario final del componente se tenga únicamente el usuario, pero a nivel de modelos se tenga la división deseada (usuarios con datos personales y cuenta con datos financieros). Algo similar sucede en el segundo método a sobrescribir, el cual es *to_representantion*, en este método se obtienen dos objetos: *User* y *Account* (usando los modelos y las función *get* provista por el ORM), pero se convierten en un único objeto totalmente natural para el usuario.

Importar Serializers

Por último, cuando se hayan implementado ambos *Serializers*, se deben agregar al módulo para que puedan ser importados más adelante. Para esto solo se deben agregar las siguientes líneas de código en el archivo, *serializers/__init__.py*:

```
from .accountSerializer import AccountSerializer
from .userSerializer import UserSerializer
```

Nota: de ser necesario, en el material de la clase se encuentra una archivo *C3.AP.09. bank_be.rar*, con todos los avances en el desarrollo del componente realizado en esta guía.