Documento API CUENTA

Juan Sebastián Parada Celis

Noviembre 2018

1 Descripción

La API fue construida con Django y la librería Django Rest Framework. Permite crear cuentas, a las cuales se les puede asociar perfiles. Un perfil se refiere al rol que tiene un cliente respecto a una cuenta. Por ejemplo, la cuenta C, es compartida por los clientes A y B, dónde el cliente A es el propietario de la cuenta y el B, es beneficiario. Por lo tanto ambos clientes A y B, pueden realizar transacciones con una misma cuenta.

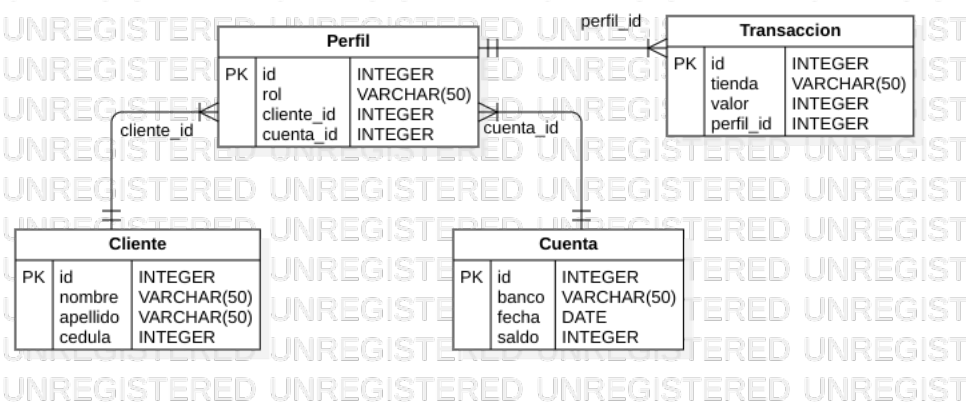

Figure 1: Modelo de Base de Datos

CLIENTE				
Campo	**Tipo**	**PK**	**FK**	**Descripción**
id	integer	Sí	No	Identificador de cada registro
nombre	varchar(50)	No	No	Nombre del cliente
apellido	varchar(50)	No	No	Apellido del cliente
cedula	integer	No	No	Cedula del cliente

CUENTA				
Campo	**Tipo**	**PK**	**FK**	**Descripción**
id	integer	Sí	No	Identificador de cada registro
banco	varchar(50)	No	No	Banco al qué está asociada la cuenta
fecha	date	No	No	Fecha de creación de la cuenta
saldo	integer	No	No	Saldo de la cuenta

PERFIL				
Campo	**Tipo**	**PK**	**FK**	**Descripción**
id	integer	Sí	No	Identificador de cada registro
rol	varchar(50)	No	No	Rol del cliente en la cuenta
cliente_id	integer	No	Sí	Llave foránea relacionada con la tabla cliente
cuenta_id	integer	No	Sí	Llave foránea relacionada con la tabla cuenta

TRANSACCIÓN				
Campo	**Tipo**	**PK**	**FK**	**Descripción**
id	integer	Sí	No	Identificador de cada registro
tienda	varchar(50)	No	No	Tienda en la cual se realizó la transacción
valor	integer	No	Sí	Valor de la transacción
perfil_id	integer	No	Sí	Llave foránea relacionada con la tabla perfil. El perfil qué realizó la transacción

2 Despliegue

Existen dos maneras para realizar pruebas de la API. La primera consiste en clonar el repositorio disponible en esta URL:

https://github.com/jsparadacelis/api_cuenta

Para esto es necesario tener Git y Docker instalados. Para la realización de este proyecto se utilizaron git version 2.17.1 y Docker version 18.06.1-ce, build e68fc7a. Para realizar pruebas se deben realizar estos cuatro pasos:

1. git clone https://github.com/jsparadacelis/api_cuenta.git
2. cd api_cuenta
3. sudo docker-compose up
4. entrar a http://0.0.0.0:8000/ desde el navegador

La otra opción es entrar a la siguente URL:

https://api-cuenta-django.herokuapp.com/

En esta URL está desplegado el API sobre Heroku.

3 EndPoints

3.1 List y Detail

La API está conformada posee dos tipos de endpoints los cuales son de tipo **List** y **Detail**. El primero permite hacer solicitudes de tipo **GET** a la Api, obteniendo cómo resultado una lista de los objetos solicitados. Por ejemplo, haciendo una solicitud GET con la herramienta *cURL*:
curl https://api-cuenta-django.herokuapp.com/cliente/
retorna:


```
[
  {
    "id": 1,
    "nombre": "juan",
    "apellido": "parada",
    "cedula": 1018483756
  },
  {
    "id": 2,
    "nombre": "jorge",
    "apellido": "gomez",
    "cedula": 17116253
  }
]
```

Los endpoints de tipo Detail, permiten realizar operaciones sobre un objeto en particular. La especificación del objeto sobre el que se quiere trabajar, se hace indicando su ID en la URL de la solicitud. Estos endpoints soportan solicitudes de tipo GET, POST, PUT y DELETE. Ejemplo:
Realizando una solicitud GET cómo:
curl https://api-cuenta-django.herokuapp.com/cliente/1
retorna:

```
{
  "id": 1,
  "nombre": "juan",
  "apellido": "parada",
  "cedula": 1018483756
}
```

Realizando una solicitud POST cómo:
curl –header "Content-Type: application/json" –request POST –data '"nombre":"diego","apellido":"perez","cedula":41700353' https://api-cuenta-django.herokuapp.com/cliente/
retorna:

```
{
  "id": 3,
  "nombre": "diego",
  "apellido": "perez",
  "cedula": 41700353
}
```

Realizando una solicitud PUT cómo:
curl –header "Content-Type: application/json" –request PUT –data '"nombre":"Pablo","apellido":"Cardona","cedula":41700353' https://api-cuenta-django.herokuapp.com/cliente/1
retorna:

```
{
  "id": 1,
  "nombre": "Pablo",
  "apellido": "Cardona",
  "cedula": 41700353
}
```

Finalmente, realizando una solicitud DELETE cómo:
curl –header "Content-Type: application/json" –request DELETE https://api-cuenta-django.herokuapp.com/cliente/1
retorna:

```
status=status.HTTP_204_NO_CONTENT
```

Estos dos tipos de endpoints, List y Detail, se pueden aplicar a cualquiera de las cuatro tablas del modelo indicado anteriormente: Perfil, Transacción, Cliente y

Cuenta. La URL cambia según sea el caso, por ejemplo, en el caso en el qué se quiera utilizarlos (los endpoints) para transacción, la URL sería

https://api-cuenta-django.herokuapp.com/transaccion/ y **https://api-cuenta-django.herokuapp.com/transaccion/{id}**

Tipo	URL	Descripción
GET	/cliente/	Lista los clientes
GET/POST/PUT/DELETE	/cliente/{id}	Lista el cliente del id al que corresponde. Permite hacer crear, modificar y eliminar clientes
GET	/cuenta/	Lista las cuentas
GET/POST/PUT/DELETE	/cuenta/{id}	Lista el cliente del id correspondiente. Permite hacer crear, modificar y eliminar cuentas

Tipo	URL	Descripción
GET	/transaccion/	Lista las transacciones
GET/POST/PUT/DELETE	/transaccion/{id}	Lista la transacción del id al que corresponde. Permite hacer crear, modificar y eliminar transacciones
GET	/perfil/	Lista los perfiles
GET/POST/PUT/DELETE	/perfil/{id}	Lista el perfil del id correspondiente. Permite hacer crear, modificar y eliminar perfiles

3.2 Realizar depósito Inicializar saldo en cero y Listar transacciones

Existen, además de los endpoints expuestos anteriormente, otros tres qué permiten realizar modificaciones sobre las cuentas, además de listar las transacciones relacionadas a una cuenta. Estos son:

- add_money
- set_zero
- list_tran

El endpoint de *add_money*, permite realizar depositos o retiros sobre una cuenta. Recibe valores positivos o negativos, lo que permite modificar. La estructura de la solicitud a este endpoint es la siguiente:

curl –header "Content-Type: application/json" –request PUT –data '"id":"4","deposit":45000' https://api-cuenta-django.herokuapp.com/add_money/

retorna:

```
{
  "id": 4,
  "banco": "Banco de Bogot ",
  "fecha": "2018-11-16",
  "saldo": 390600
}
```


curl –header "Content-Type: application/json" –request PUT –data '"id":"4","deposit":-90600' https://api-cuenta-django.herokuapp.com/add_money/

retorna:


```
{
  "id": 4,
  "banco": "Banco de Bogot ",
  "fecha": "2018-11-16",
  "saldo": 300000
}
```


La solicitud recibe el id de la cuenta a modificar, y el valor que se le quiere sumar o restar. Este valor corresponde a la clave *deposit*.

El siguiente endpoint es *set_zero*. Este endpoint permite inicializar el valor de la cuenta en cero. La estructura de la solicitud es la siguiente:

curl –header "Content-Type: application/json" –request PUT –data '"id":"4"' https://api-cuenta-django.herokuapp.com/set_zero/

retorna:


```
{
  "id": 4,
  "banco": "Banco de Bogot ",
  "fecha": "2018-11-16",
  "saldo": 0
}
```


curl –header "Content-Type: application/json" –request PUT –data '"id":"2"' https://api-cuenta-django.herokuapp.com/set_zero/

retorna:


```
{
  "id": 2,
  "banco": "Banco ABC",
  "fecha": "2018-11-16",
  "saldo": 0
}
```


Esta solicitud recibe el id de la cuenta que se quiere modificar a través de la url.

Por ultimo, el endpoint *list_tran* permite listar las transacciones realizadas con una cuenta en específico. La estructura de la solicitud es la siguiente:

curl –header "Content-Type: application/json" –request GET https://api-cuenta-django.herokuapp.com/list_tran/3

retorna:


```
[
  [
    {
      "id": 3,
      "tienda": "Papeleria",
      "perfil": 3,
      "valor": 40000
    },
    {
      "id": 4,
      "tienda": "Carniceria",
      "perfil": 3,
      "valor": 10000
    }
  ],
  {
    "id": 3,
    "banco": "Av Villas",
    "fecha": "2018-11-16",
    "saldo": 1200000
  }
]
```


Esta solicitud recibe el id de la cuenta a la cual se le quiere listar sus transacciones, a través de la url. Regresa las transacciones realizadas con la cuenta, además de los datos de la cuenta.

Tipo	URL	Descripción
PUT	/add_money/	Suma o resta valores al saldo de la cuenta
PUT	/set_zero/	Inicializa el saldo de la cuenta en cero
GET	/list_tran/{id}	Lista las transacciones de la cuenta relacionada con el id qué se le pasa por la url

4 Resolviendo el Double Spending

El problema de double spending se refiere, en este contexto, a la concurrencia de operaciones sobre una misma cuenta, es decir, a realizar más de una transacción al mismo tiempo utilizando la misma cuenta para realizarlas. La solución que se implementó para resolver este problema corresponde a un enfoque pesimista, lo que implica un bloqueo sobre la base de datos, en este caso, sobre una cuenta.

Por lo tanto, si dos perfiles van a realizar una cada uno una transacción utilizando la misma cuenta, aquel qué acceda primero al recurso, bloqueará la cuenta hasta qué se complete la transacción. Por ejemplo:

1. El usuario A va a realizar una transacción con la cuenta 1. Bloquea la cuenta.
2. El usuario B va a realizar una transacción con la cuenta 1. No puede acceder a la cuenta, dado qué el usuario A la bloqueó.
3. El usuario A realiza la transacción y libera a la cuenta 1 del bloqueo del paso 1.
4. El usuario A va a realizar una transacción con la cuenta 1. Bloquea la cuenta y realiza la transacción

La implementación de lo expuesto anteriormente, se llevó a cabo utilizando **select_for_update**, una función perteneciente a la API de QuerySet de Django. Esta función permite realizar el bloqueo de un objeto hasta que se termine la transacción que se está efectuando sobre este. En este caso, la función fue implementada en el metodo *get_object* de la vista CuentaDetail.

```
def get_object(self, pk):
    try:
        #Locking object
        return Cuenta.objects.select_for_update().get(pk=pk)
    except Cuenta.DoesNotExist:
        raise Http404
```

Figure 2: Implementación de select_for_update

De acuerdo con lo anterior, para realizar el bloqueo,es necesario qué la operación cumpla con el prinicipio de Atomicidad de las bases de datos; lo que significa que todas las operaciones dentro de una transacción deben completarse para que la transacción este completa. Para lograr eso, se utilizó un bloque **with transaction.atomic():** dispuesto por Django para el control de transacciones. Este bloque se implementó en el metodo *create* del serializador para el modelo de Transacción (*TransaccionSerializer*).

```python
def create(self, validated_data):
    with transaction.atomic():
        perfil = validated_data["perfil"]
        cuenta = perfil.cuenta
        if cuenta.saldo <= 0:
            raise serializers.ValidationError("Fondos insuficientes")


        cuenta.saldo = cuenta.saldo - validated_data["valor"]
        cuenta.save()
        transaccion = Transaccion(
            tienda = validated_data['tienda'],
            perfil = perfil,
            valor = validated_data["valor"]
        )
        transaccion.save()
    return transaccion
```

Figure 3: Implementación del bloque transaction.atomic()