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

(Modelos y Migraciones)

Actividad Práctica

Capa Lógica: Manejo de Datos

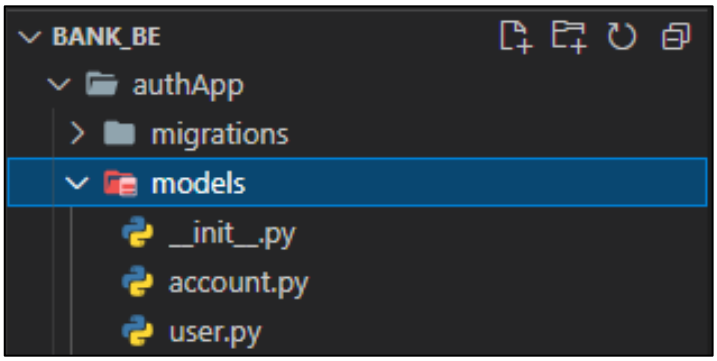
El componente *bank_be*, correspondiente a la capa lógica o capa de back-end, será el encargado de procesar y manejar los datos ingresados y retornados al usuario. Sin embargo, para cumplir su función es necesario que exista una persistencia de los datos, característica que no posee por su cuenta dicho componente. Por esta razón, para lograr la persistencia es necesario utilizar el componente de la capa de datos *bank_db*, lo que implica la interacción entre la capa lógica, y la capa de datos. De manera técnica y entrando en detalles de la implementación, el componente *bank_be* deberá establecer una conexión con la base de datos *bank_db*, pero además de esta conexión se deberá establecer un esquema que permita a ambos componentes definir la manera en la que se almacenaran los datos.

En general, tanto el framework *Django REST*, como la mayoría de frameworks, usan el concepto de *ORM*. Sin entrar en detalles, un *ORM* es una herramienta que, a través de Modelos (Clases con atributos y métodos especiales), permite definir en la capa lógica la estructura con la que se almacenarán los datos. Sin embargo, un *ORM* no solo define la estructura en la capa lógica, sino que, además con ayuda de un proceso llamado migraciones, permite crear el esquema en la base de datos correspondiente a la estructura ya definida. Los *ORM* suelen definir una serie de métodos que facilitan la comunicación con la base de datos, pues eliminan el trabajo de crear consultas *SQL* para guardar, obtener, eliminar, o actualizar datos de la base de datos.

Otro aspecto que se debe tener en cuenta durante el desarrollo de la capa lógica es el sistema de *autenticación*. Este sistema de autenticación es transversal a todo el componente, incluyendo el manejo de datos. Si bien se podría implementar un sistema propio desde cero, *Django* ofrece algunos modelos base que facilitan la adecuación de este sistema a las necesidades de cualquier proyecto.

Modelos

Para el propósito del sistema, se tendrán dos entidades, la entidad usuario (*User*) y la entidad cuenta (*Account*), la entidad usuario representará los datos personales del usuario, y la entidad cuenta representará la información bancaria del usuario. Estas dos entidades tendrán una relación de 1 a 1, es decir, un usuario tiene una única cuenta y una cuenta solamente pertenece a un usuario. Ambas entidades serán representadas con sus respectivos modelos, el modelo de la entidad cuenta será bastante simple, mientras que el modelo de la entidad usuario necesitará algunas configuraciones especiales para adecuarse al sistema de autenticación de *Django*. Los modelos serán agrupados en un módulo por ello antes de iniciar el desarrollo se debe tener la siguiente estructura de archivos:



Dentro de la aplicación (*authApp*) se debe crear un módulo llamado "*models*" con su respectivo archivo *__init__.py*, y además se deben crear otros dos archivos para cada uno de los modelos.

Modelo User

El modelo correspondiente al usuario puede parecer un poco complejo ya que requiere algunas configuraciones adicionales, pero esto simplemente son componentes extras. No se debe perder de vista el concepto de modelo, el cual abarca un conjunto de datos que representan la información de una entidad.

El código correspondiente al modelo user (*models/user.py*) es el siguiente:

```
from django.db import models
from django.contrib.auth.models import AbstractBaseUser, PermissionsMixin, BaseUserManager
from django.contrib.auth.hashers import make_password

class UserManager(BaseUserManager):
    def create_user(self, username, password=None):
        """
        Creates and saves a user with the given username and password.
        """
        if not username:
            raise ValueError('Users must have an username')
        user = self.model(username=username)
        user.set_password(password)
        user.save(using=self._db)
        return user

    def create_superuser(self, username, password):
        """
        Creates and saves a superuser with the given username and password.
        """
        user = self.create_user(
            username=username,
```

```
            password=password,
        )
        user.is_admin = True
        user.save(using=self._db)
        return user

class User(AbstractBaseUser, PermissionsMixin):
    id = models.BigAutoField(primary_key=True)
    username = models.CharField('Username', max_length = 15, unique=True)
    password = models.CharField('Password', max_length = 256)
    name = models.CharField('Name', max_length = 30)
    email = models.EmailField('Email', max_length = 100)

    def save(self, **kwargs):
        some_salt = 'mMUj0DrIK6vgtdIYepkIxN'
        self.password = make_password(self.password, some_salt)
        super().save(**kwargs)

    objects = UserManager()
    USERNAME_FIELD = 'username'
```

El anterior código puede parecer un poco complejo, pero realmente no lo es, en primer lugar, se tiene la clase *UserManager*, esta clase es un mecanismo que provee *Django REST* para administrar la manera en la que se crean los usuarios en el sistema de autenticación, debido a la estructura del *framework* se debe definir los métodos para la creación de *usuarios* y *super-usuarios*, pero para este caso el componente únicamente manejara usuarios. El manager también le permite a *Djang oREST* identificar las credenciales de los usuarios y brindarles un mecanismo de seguridad adecuado.

Una vez se tiene el manager, se procede a crear el modelo User. Dado que este es un modelo que requiere un mecanismo de autenticación, este se crea a partir de dos clases que provee *Django REST*: *AbstractBaseUser* y *PermissionsMixin*, estas clases permiten crear un modelo de usuario básico con autenticación. Una vez se tiene esta estructura inicial, se procede a la parte principal del modelo: la definición de sus atributos (*id*, *username*, *password*, *name*, *email*), estos se definen utilizando de manera normal usando la clase *models* provista por *Django REST*. Debido a que se está trabajando con un campo sensible como es el *password*, se debe sobrescribir el método de guardado (*save*) y realizar un proceso de *hashing* para brindar mayor seguridad. Por último, se debe asociar el *UserManager* al modelo *User*, esto se logra con las últimas líneas de código.

Modelo Account

Ahora se definirá el modelo correspondiente a la entidad de cuenta, este es mucho más simple que el modelo de la entidad usuario, ya que al no necesitar autenticación se puede definir con las clases bases o estándares establecidas por *Django REST*. Sin embargo, tiene un pequeño detalle y es que en este modelo se debe definir la relación entre las entidades usuario y cuenta, la relación entre estas es 1 a 1 y si bien dicha relación puede presentar un nivel de abstracción alto en la implementación, simplemente

se define con una referencia en un atributo.

El código correspondiente al modelo account (*models/account.py*) es el siguiente:

```
from django.db import models
from .user import User

class Account(models.Model):
    id = models.AutoField(primary_key=True)
    user = models.ForeignKey(User, related_name='account', on_delete=models.CASCADE)
    balance = models.IntegerField(default=0)
    lastChangeDate = models.DateTimeField()
    isActive = models.BooleanField(default=True)
```

Como es de esperarse, el modelo account es mucho más sencillo ya que solamente se debe definir la parte principal del modelo que son los atributos (*id*, *user*, *balance*, *lastChangeDate*, *isActivate*). El atributo *user* es un atributo de tipo *ForeignKey* y permite realizar una referencia al modelo *User*, lo cual permite establecer la relación.

Exportar y Registrar modelos

Ahora que se ha finalizado la implementación de los modelos, se debe realizar una serie de pasos que permitan integrarlos al módulo y a la aplicación. Primero se debe exportar los modelos en el módulo *models*, para esto bastará con llamarlos en el archivo *__init__.py* de la carpeta *authApp/models*, esto se logra con las siguientes líneas de código:

```
from .account import Account
from .user import User
```

Una vez los modelos han sido exportados en el módulo, se pueden registrar en la aplicación. Para esto bastará con escribir las siguientes líneas en el archivo *authApp/admin.py*:

```
from django.contrib import admin
from .models.user import User
from .models.account import Account

admin.site.register(User)
admin.site.register(Account)
```

Ahora solo falta un último detalle y es indicarle al proyecto cual será el modelo que se utilizará para realizar la autenticación. Para esto solo se debe agregar la siguiente variable en el archivo *authProject/settings.py*:

```
AUTH_USER_MODEL = 'authApp.User'
```

Esta variable se puede ubicar en cualquier parte del archivo, sin embargo, para mantener el orden de las configuraciones, se ubicará bajo la variable *REST_FRAMEWORK*:

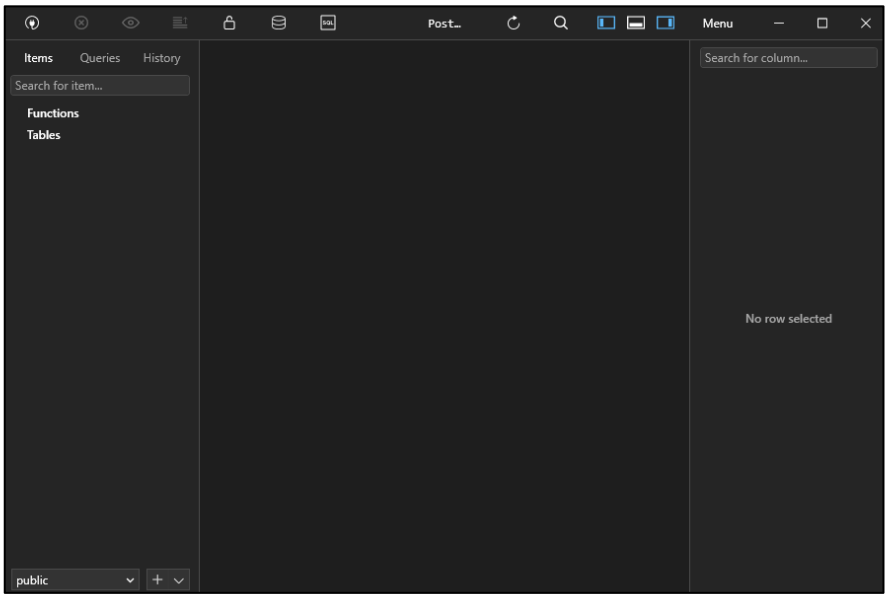
settings.py

authProject > settings.py > ...

```
67  REST_FRAMEWORK = {
68      'DEFAULT_PERMISSION_CLASSES': (
69          'rest_framework.permissions.AllowAny',
70      ),
71      'DEFAULT_AUTHENTICATION_CLASSES': (
72          'rest_framework_simplejwt.authentication.JWTAuthentication',
73      )
74  }
75
76  AUTH_USER_MODEL = 'authApp.User'
77
```

Migraciones

En este punto ya se ha definido por completo la estructura para el manejo y persistencia de datos en el componente de *backend*, pero el componente de *datos* aun no sabe cómo se almacenarán los datos. Si se ingresa a la base de datos a través del cliente *TablePlus*, se evidenciará que no existe ninguna tabla:

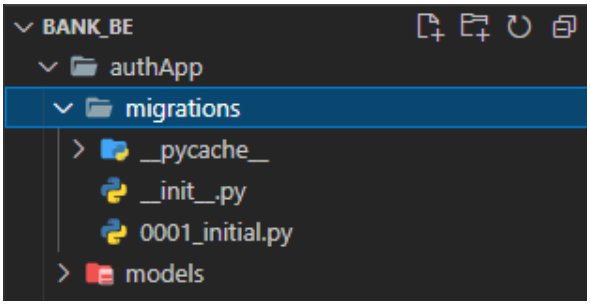


Para establecer un esquema en la base de datos, que permita a ambos componentes interactuar, se debe realizar un proceso llamado *migración*, en este proceso el *ORM* del framework (en este caso Django REST) toma los modelos definidos, y a partir de estos crea un *esquema* equivalente en la base de datos.

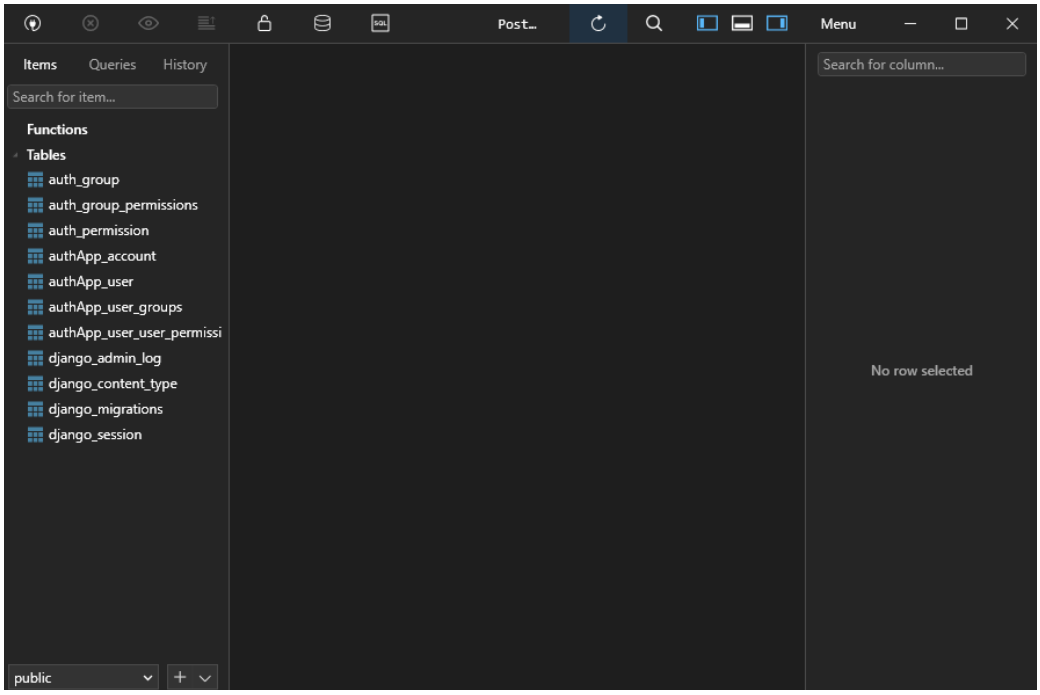
Un proceso de migración consta de dos pasos, primero se crea una serie de archivos en la que se recolecta información de los modelos actuales y se definen las instrucciones de la migración, y un segundo paso en el que, con ayuda de los archivos previamente definidos, se crea el esquema en la base de datos. Para realizar los dos pasos correspondientes a la *migración*, se deben ejecutar los siguientes comandos en la raíz del componente dentro del *entorno virtual*:

```
python manage.py makemigrations authApp
python manage.py migrate
```

Para comprobar que el primer comando se ejecutó de manera correcta se debe verificar que se creó la carpeta *migrations* en la aplicación (*authApp*). Si esta carpeta ya estaba creada antes de ejecutar los comandos, entonces se puede verificar que se ejecutó el comando correctamente al verificar que se creó un archivo adicional en esta carpeta, en este caso el archivo *0001_initial.py*:



Para comprobar que el segundo comando se ejecutó de manera correcta se debe revisar el estado de la base de datos con ayuda del cliente *TablePlus*:



Si el comando se ejecutó correctamente, se debe evidenciar la creación de una serie de *tablas* en la base de datos, que representan el esquema creado por el *ORM*. Muchas de estas tablas son propias del sistema de autenticación provisto por *DjangoREST*. Sin embargo, se ha de notar que las tablas *authApp_user* y *authApp_account* son las tablas que representan los modelos creados anteriormente.

Ejecución del Componente

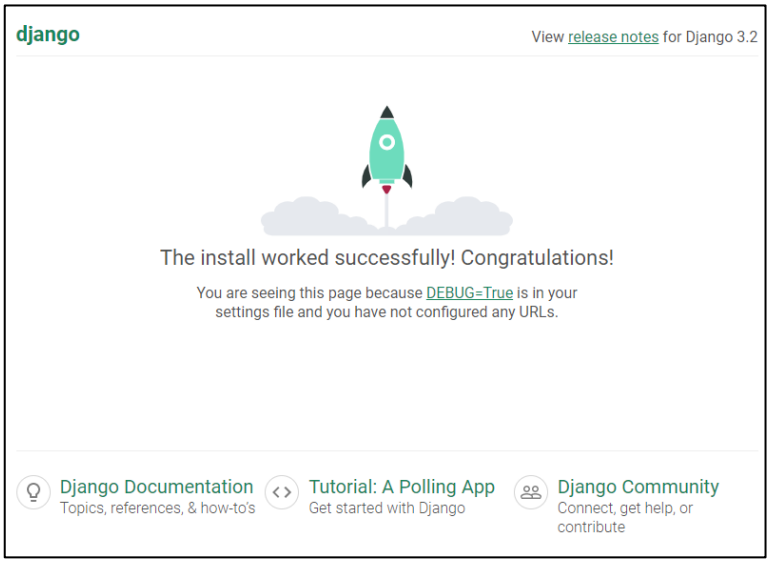
En la sesión anterior se indicó que no se debía ejecutar el servidor hasta finalizar el desarrollo de esta guía. Una vez finalizado lo anterior, ya se puede realizar la ejecución del servidor con el comando *python manage.py runserver*. Si todos los pasos de esta y de la guía anterior se realizaron adecuadamente, no aparecerán errores en la terminal:

```
PROBLEMS  TERMINAL  OUTPUT  DEBUG CONSOLE

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

System check identified no issues (0 silenced).
September 21, 2021 - 12:02:51
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.
```

Y se ejecutará el servidor localmente en la dirección *http://127.0.0.1:8000/*:



Nota: de ser necesario, en el material de la clase se encuentra una archivo *C3.AP.08. bank_be.rar*, con todos los avances en el desarrollo del componente realizado en esta guía.