Crear carpeta donde se trabajara el backend

En cmd:

1 -INSTALAR ENTORNO VIRTUAL

python -m venv env

2-INICIAR ENTRONO VIRTUAL

.\env\Scripts\activate

# *para desactivar el entorno virtual*

deactivate

3- VER MODULOS INSTALADOS

pip list

4- Instalar Django

pip install Django

5- VERIFICAR SI SE INSTALO DJANGO CON PIP LIST

para desinstalar es pip uninstall nombre\_programa

6- Crear Proyecto. Por norma general se llama backend.

django-admin startproject nombre\_proyecto

7- Una vez creado en settings.py se puede ver un DEBUG = True eso significa que esta en

estado de construccion. Y la secret key.

Templates

Base de datos, no trabajar con el sqlite3

Passwords

Lenguaje, zona horaria

Rutas de archivos(js, html, css)

8-En el archivo urls.py nos muestra como crear importaciones de vistas

9-wsgi.py es para levantar el proyecto

10- manage nos permite administrar el proyecto, a traves de comandos para comenzarlo, pararlo, migraciones

11- Si los archivos nos salen con lineas con corrector amarillo, es porque vsc no encontró los

paquetes instalados. Se debe cambiar la ruta del interprete, ir al margen inferior derecho,

apretar donde dice 3.10.8 64-bit(microsoft store), en la ventana superior que aparece seleccionar

Escriba la ruta de acceso del interprete -> Buscar... -> En la carpeta donde se creó la app -> ir al env

-> scripts -> python ->seleccionar -> y debe quedar en el margen inferior derecho 3.10.8 ('env':venv)

12 - Levantar el proyecto: para ello debo estar parado en la consola en la carpeta que se encuentre el manage.py

python manage.py runserver

Nos devuelve un server de desarrollo con una ip (link), el cual debe abrirse en un navegador.

Para frenar el server utilizo ctrl+c

Crea un archivo nuevo, que es una base de datos db.sqlite3 se puede cambiar la base de datos desde settings.py

Las migaciones es traducir codigo a sentencias de sql y los inpacta en la base de datos

Para ejecutar las migraciones de base de datos se ejecuta el comando:

python manage.py migrate

13 - Admin de DJANGO para poder gestionar la base de datos.

se ejecuta el server y al server devuelto lo copio y lo pego en un navegador donde le coloco /admin

Así voy a poder ingresar. Los datos del perfil no se comparte con nadie.

14- CREACIÓN USUARIO ADMIN DJANGO:

se realiza tirando el siguiente comando:

python manage.py createsuperuser

poner nombre usuario/mail/contra

15 - Panel de Administración:

Se debe volver a correr el servidor, ingresar con el server que me brinda colocarle /admin e ingresar con usuario

y contra creada. Luego me permite entrar y administrar el panel de usuarios.

17 - CONFIGURACIONES SETTINGS

18- Instalar DJANGO REST FRAMEWORK:

comando: pip install djangorestframework

verificar con pip list su instalación.

Luego en la sección de settings en installed apps, se van a dividir los modulos instalados en BASE\_APPS (modulos internos bases de django), LOCAL\_APPS (modulos/apps locales)

y THIRD\_APPS(modulos de 3ros) aquí se copia/registra el 'rest\_framework', para que se instale rest en nuestra aplicación

para luego concatenarlos bajo la variable INSTALLED\_APPS = BASE\_APPS + LOCAL\_APPS + THIRD\_APPS

Levantar el server para controlar que se haya realizado el proceso de manera correcta.

19 - BASES DE DATOS:

Vamos a cambiar la base de datos porque sqlite3 es muy basico para la ejecución de un proyecto.

En settigns vamos a ir al objeto DATABASES.

DATABASES = {

    'default': { ''' Indica que se utiliza por defecto '''

        'ENGINE': 'django.db.backends.sqlite3', '''Indica que motor se utiliza'''

        'NAME': BASE\_DIR / 'db.sqlite3', '''nombre de la base que utiliza'''

    }

}

Para ello accederemos al link que se encuentra por encima del objeto DATABASES.

https://docs.djangoproject.com/en/4.1/ref/settings/#*databases*

y lo vamos a cambiar por otra base de datos de mysql:

DATABASES = {

    'default': {  ''' Indica que se utiliza por defecto '''

        'ENGINE': django.db.backends.mysql', '''Indica que motor se utiliza'''

        'NAME': 'mydatabase', '''nombre de la base que utiliza'''

        'USER': 'mydatabaseuser', '''buscar en el motor de base de datos'''

        'PASSWORD': 'mypassword', '''Igual que user'''

        'HOST': '127.0.0.1', '''Igual que user'''

        'PORT': '3306', '''Se ve en la consola de xampp mysql port'''

    }

}

Buscar mas abajo el motor de base de datos a utilizar en esta ocación se usara mysql

Luego editar los demas campos.

Levantar Xampp:

creamos una nueva base de datos, la dejamos en blanco solo le colocamos el nombre relacionado a la app.

Luego busco busco el user, host y port en phpMyAdmin -> Cuentas de usuarios ->muestra tabla con los datos.

Instalar el driver para realizar la conexión/traducción entre django y mysql: (para evitar errores)

comando: pip install mysqlclient

Posterior debo realizar las migraciones para que se creen automaicamente las tablas en la base de datos

Comando: python manage.py migrate

Y refrescar el phpMyAdmin y verificar si la base de datos contiene las nuevas tablas.

20 - iNDEPENDIZACIÓN DE APARTADOS CON APPS

python manage.py startapp nombreDescriptivo

Se crea una nueva carpeta con sus archivos y secciones. La carpeta se va a llamar como el nombre descriptivo del comando tirado en consola

Podemos manejar todas las funciones de la app. y aquí estan los modelos (patrones de diseño de la app). Modelo Vista Template/Controlador.

Este se basa al modelo de base de datos (tablas)

El controlador va a ser la parte de las funciones.

Models para crear los modelos de base de datos(tablas sql)

Controlador /Views es donde se construye toda la logica, tanto front se capturan los post los get y peticiones a la base de datos.

Views es el front pero al trabajar con rest no se hace. La deolución se hace a traves de formato JSON(archivos que guardan info en el backend para enviarlo al front).

Ir a la carpeta de la apliación :

Models.py -> los modelos son tablas de la base de datos. En su sección se pueden crear nuevas tablas/modelos.

se crea una class que va a heredad de models, luego se coloca un .Model que significa que va a sobreescribir en modelo, que ya tiene su propio constuctor.

a la clase se le dan:

-Opciones: se van a contruir objetos que contengan valores que pueden ser obtenidos por inputs tipo radio, checkbox, desplegables. Y de esta manera simplificar la ontrucción de atributos.

luego se va a conectar con el atributo correspondiente como valor de una de sus cualidades.

-Atributos, a estos se es asigna un models. para darle un tipo de campo x ej. models.charfield.

a cada atibuto luego de darle el tipo, se le asignan cualidades: longitud, que sea unico,

# Create your models here.

class Hero(models.Model):

    #Opciones

    UNIVERSE\_CHOICES = (

        ('1', 'Marvel'),

        ('2', 'DC')

    )

    # Atributos

    name = models.CharField(

        max\_length = 100,

        unique = True,

        verbose\_name = 'Nombre', """Como figura el nombre en la tabla"""

    )

    age = models.IntegerField()

    universe = models.CharField(

        max\_length = 1,

        choices = UNIVERSE\_CHOICES,

        verbose\_name = ' Universo'

    )

 Por último, se debe registrar la app en settigns en el objeto de LOCAL\_APPS

 con el nombre de la app (nombre de la carpeta, en este caso es 'heroe')

Si a la class de mi modelo le paso como arg/param (AbstractBaseUser, PermissionsMixin) --> me permiten usar mi propio modelo con mis propios atributos y usarlos por defecto,  va a heredarde estos dos atributos. Que nos permite crear y decirle a django que vamos a utilizar este modelo como nuevo de usuario.

class Nombre\_Class(AbstractBaseUser, PermissionsMixin):

          prop1 = models.TipoDeDato(max\_length = 255, unique = True, null = True, blanck = true, 'nombre de como figura en tabla', upload\_to='perfil/' -para img-, default = True)

                          ChardField / EmailField / ImageField / BooleanField/

          object = UserManager() """creado para manejar todo lo referente al usuario se debe definir manualmente arriba del modelo, se definen sus atributos, que elementos se van a pedir al generar un nuevo usuario."""

class Meta:

    verbose\_name = 'nombre\_en\_singular'

     verbose\_name\_plural = 'nombre\_en\_plural'

    managed = False """Esta prop impide que django pueda manejar la tabla y de está manera no podrá utilizarla un 3ro para manipularla y plasmarla en su template, o debe ser True o eliminarse."""

    db\_table = 'nombre\_tabla'  """Indica que el modulo apunta a una tabla ya existente, puede estar o no."""

USERNAME\_FIELD = 'username'           """Le pido determinado/s atributos para loggearse"""

REQUIRED\_FIELD = ['email', 'name']     """Datos adicionales para crear un super usuario"""

                               class NombreModelo(models.Model)

Relacionar tablas en models ---> var\_model\_orm = models.OneToOneField(modelo, prop = models.CASCADE)

                                                  models.ManyToManyField()

Tambien se pueden dejar creado un archivo con datos/registros por defecto y que me lo retorne. Para ello:

ejecutar el comando:

debo crear un archivo extra, dentro de la carpeta global con el nombre de datos iniciales y allí mismo le exportamos los datos en un archivo.

python manage.py dumpdata  ---> Importa todos los objetos de la base de datos en formato json. Los importa en la app.

python manage.py dumpdata  carpetaLocal/hija/modulo.Models/tabla ---> Importa los objetos que contiene especificamente ese modulo(base de datos) con esa modelo(tabla).

Se copia el resultado obtenido en el cmd y se crea un archivo dentro de la carpeta inicialdata y allí se copia esta info con formato json.

python manage.py loaddata ---> es para exportar la info.

21 - Volver a realizar Migrate para que se carguen los nuevos modelos.

Comando: python manage.py makemigrations

En la carpeta de la app (heroe), en migrations se puede observar el nuevo archivo

creado con el nombre 0001\_initial.py que muestra la nueva tabla a crear,

los campos que contiene, sus tipos de datos, atributos y opciones.

Se debe volver a ejecutar una migración

Comando: python manage.py migrate

Refrescar el phpMyAdmin y allí debe figurar la nueva tabla dentro de nuestra base de datos.

le comando:

python manage.py showmigrations ---> Muestra los migrates que se han realizado (historial de migrates), se deben corroborar al modficar la base de datos.

python manage.py sqlmigratite nombre\_app/local/hijo 0001  ---> """Devuelve la query que se realizo en la bbdd para crear o la operación que se haya realizado en el numero del migrate (0001) en lenguaje mysql."""

reset migration ---> ir a la carpeta hija y buscar los migrations allí borrar todos menos el \_\_init\_\_.py

y el cache también borrarlo. y volver a ejecutar el makemigrations y el migrate (este ejecuta las querys en la base de datos).

22- Entrar al Django Admin

23 - Debo registrar mi modelo para utilizarlo en el Django Admin.

Voy a la carpeta de la aplicación (heroe), al archivo admin.py y en él voy a registrar mi modelo

# Imports de models

from heroe.models import Hero

    """Ruta"""        """Modelo"""

Register your models here.

admin.site.register(Hero)

Refrescar admin y allí debe aparecer la nuva tabla/modelo.

24 - Creación de servicios APIS.

Crear dentro de la carpeta principal de la app (en este caso heroesApp),

una carpeta que se llame apps, para colocar en ella todas las aplicaciones nuevas que se creen.

En la carpeta hereo (carpeta de la unica app creada hasta el momento), crear una nueva carpeta donde

se almacenaran todos los archivos relacionados al las apis (modulos, rutas, serialazers).

25 - Crear Views:

ir al archivo views.py de la app.

Importar rest, models,

Trabajar en vistas basadas en clases.

se crea una class NombreServicioDescriptivo(DEDONDEVAHeredar)

luego utilizar metodos para solicitar,pedir, modificar,etc.

Los metodos van a recibir argumentos estos van a ser self, request y pueden ser id, etc.

APIView intercepta metodos.

Luego se crea la logica.

Como las views son clases/objetos podemos acceder a sus metodos y propiedades.

Podemos crear diferentes tipos de vistas para trabajar con n cantidad de metodos.

class HeroApiView(APIView):

    def get(self, request):

        """Retorna un listado con todos los heroes almacenados en la base"""

        print(f'REQUEST --->{request}') """1ro se crea un print para poder ver que nos devuelve la request"""

Una vez creada la vista(view) con su servicio de petición(API), se procede a correr la app con un runserver, (activar anteriormente el env)

Como tenemos la base de datos de phpMtAdmin ligada a la app también debe encenderse, para poder correr la app.

Crear una request con un metodo post para llevar información del front y postearlo ingresarlo en la base de datos.

puedo crear/capturar los metodo dentro de la misma clase (usa un solo url) o indipendientes cada uno con una clase (y cada uno con un url)

En la misma clase creada en el archivo views.py voy a crear una función post con los arg/param self(para hacer referencia a si misma, permite instanciar la funcion dentro de si misma), request.

Debajo coloco una descripción de que es lo que realiza.

PRIMER EJMPLO\* realizo un primer print, los print se utilizan para debuggear ( es decir para ir controlando que nos devuelve esa función en consola y si estaá correcta)

Luego le añado un data diccionario que tiene como key un 'mensaje':'el mesaje a mostrar'

Luego dor un return Response(respuesta) que devuelve la data diccio creada anteriormente y un status.

Estos ultimos dos devuelve en el front.

 def post(self, request):

        """crea un nuevo registro/heroes"""

        Primer ejemplo\*

        print("ESTAMOS EN EL METODO POST !!!")

        data = {

            'men': 'todo pk'

        }

        return Response(

            data = data,

            status = status.HTTP\_201\_CREATED

        )

Para hacer uso de este metodo puedo pasarle por postman, creo una nueva request cuyo nombre es descriptivo a la función, agrego el link, selecciono la opción Body - raw - y el formato JSON, luego en formato json

le paso el nuevo objeto(heroe) a crear, guardo la sentencia.

Objeto en formato Json:

{

        "id": 3,

        "name": "Thor",

        "age": 2000,

        "universe": "1"

    }

En el mismo url se van a poder acceder a dos servicios uno es el get y el otro es el post, el primero paa mostrar la lista de heroes y el segundo para crear y agregar un nuevo heroe.

Solo usar de esta manera cuando tenga sentido de utilizar. Para ello deben ser coherentes las urls.

De esta manera estamos pasando desde el front un nuevo Objeto en formato Json, el paso que nos falta es hacer que el backend tome esa info y la guarde en la base de datos.

Para ello:

Utilizamos el serializer, que traduce en ambos sentidos, en este caso toma lo que nos envía el front traducirlo a un objeto que entienda la base y guardarlo en la misma base de datos. Sería el camino inverso que realiza el serializer en el metodo get.

Se pueden enviar 1 x 1 o se puede enviar una fila de objetos, igualmente el serializer introduce 1 x 1 en la base de datos

EJEMPLO 2\* USO DEL SERIALIZER:

creamos una variable dentro del meodo post ya creado, que se le va a asignar el serializer creado en el metodo get que está asociado al modelo, si utilizamos otro modelo se debe crear otro serializer.

va a recibir como arg/param -> data = request.data para poder acceder a la data que trae la request y poder transformarlo con el serializer.

vamos a printear el serializer para debugguearlo.

y un print type -> serializer

 # EJEMPLO 2\*

        serializer = HeroSerializer(data = request.data)

        print(serializer)

        print(type(serializer))

Aún no se guarda en base de datos.

EJEMPLO 3\* VALIDACION DE DATOS:

Comprobación logica para que los datos ingresados en el front sea correcta segun mis parametros(¿/campos de la base de datos.)

Para solo guardar la info si está validada y es correcta. Para ello utilizo una función if.

Hace consultas a la base de datos y si no es correcto nos devuelve un error.

If que utiliza el objeto serializer creado, lo vamos a constratar con su metodo is\_valid() para ver si es valido,

si lo es le vamos a pedir que lo guarde con serializer y su metodo save().

Coloco el objeto data y el return Response dentro del if para que se ejecute si es True.

La contraparte, es decir, el caso contrario si es False, se puede poner un else o directamente un return Response con data que contenga un mensaje dado por el serializer(el error que este objeto encontro) accediendo a su metodo

serializer.errors y luego le brindamos un status 400.

  if serializer.is\_valid():

            serializer.save()

            data = {

                'men': 'todo ok'

            }

            return Response(

                data = data,

                status = status.HTTP\_201\_CREATED

            )

        return Response(

            data = serializer.errors,

            status=status.HTTP\_400\_BAD\_REQUEST

        )

Listo ya se guarda el objeto en la bse de datos.

La funcion post quedaria asi:

def post(self, request):

        """Descripción de que va a registrar en la base de datos"""

      serializer = HeroSerializer(data = request.data) """Uso del serializer para traducir de formato json al formato de la base de datos"""

      if serializer.is\_valid():    """Validacion de datos"""

            serializer.save()

            data = {

                'men': 'todo ok'

            }

            return Response(

                data = data,

                status = status.HTTP\_201\_CREATED

            )

        return Response(

            data = serializer.errors,

            status=status.HTTP\_400\_BAD\_REQUEST

        )

Crear una vista independiente para el metodo Post:

crear una nueva clase dentro del archivo views.py ya utilizado. Y luego el metodo post creado anteriormente.

class NombreDescriptivoApiView(APIView """Es de donde hereda su función""")

class CreateHeroApiView(APIView)

  def post(self, request):

        """Descripción de que va a registrar en la base de datos"""

      serializer = HeroSerializer(data = request.data, many=True) """Uso del serializer para traducir de formato json al formato de la base de datos"""

                                                                  """many=True, me permite ingresar muchos resgistros a la base de datos.""""

      if serializer.is\_valid():    """Validacion de datos"""

            serializer.save()

            data = {

                'men': 'todo ok'

            }

            return Response(

                data = data,

                status = status.HTTP\_201\_CREATED

            )

        return Response(

            data = serializer.errors,

            status=status.HTTP\_400\_BAD\_REQUEST

        )

Luego de crear la vista independiente con su class, debo crearle una url en la carpeta local, independiente.

SE RECOMIENDA UTILIZAR UNA VISTA POR METODO HTTP, es mas facil para relacionarlos con el front.

CREAR UN METODO GET PARA QUE TRAIGA UN SOLO HEROE Y MAS DETALLES SOBRE ÉL:

Creo una nueva class que va a contener varios metodos

Get - Put - Delete

def get(self, request, pk): """El pk es un parametro mas es como usar el id"""

      """Breve descripción de la función del metodo"""

      heroe =  """Guardo dentro de una var que va a ser en infinitivo porque solo quiero un objeto(heroe) en este caso,

              Modelo.ORM.metodo"""

              Hero.objects.get(id=pk)     """En el caso de get trae un solo objeto/valor que es el primero de todos, para encontrar esa coincidencia se utiliza como atributo id=pk para que seleccione los id"""

no usar       Hero.objects.filter() """En cuanto al filter, que es un metodo gral, nos trae todos los registros que cumplan con el paramtero --> se utiliza para filtrar un determinado campo/parametro indicando cual va a ser = al que solicité"""

       heroe\_serializer = HeroSerializer(heroe)

"""Guardo dentro de la var el modelo serializado y como atributo la var creada anteriormente que tiene asignado el ORM """

"""y por último el return:""""

        return Response(

            data = heroe\_serializer.data, """Mensaje a etregar cuando se cumpla la ejecución"""

            status=status.HTTP\_200\_OK     """Estado a entregar cuando se cumpla la ejecución"""

        )

Quedaría finalmente:

class HeroDetailApiView(APIView):

   def get(self, request, pk):

        """Nos devuelve mas info de un heroe en particular"""

      # print('Get')

      # print(f'PK --> {pk}')

        heroe = Hero.objects.get(id = pk)

      # print(f'Get del ORM -> {heroe}')

        heroe\_serializer = HeroSerializer(heroe)

      # print(f'Get del ORM -> {heroe\_serializer.data}')

        return Response(

            data = heroe\_serializer.data,

            status=status.HTTP\_200\_OK

        )

Se pueden realizar prints para debuggear si entra al metodo

Y debo crear el nuevo url local para esta nueva vista.

Este url va a contener en la ruta un atributo <int:pk>/ que permite acceder a un heroe colocando un numero entero despues de la ruta, ya que asocia ese numero entero a un id de la base dedatos.

De esta manera accedo a todos los datos almacenados para ese objeto (en este caso heroe)

CREAR METODO PUT: Permite modificar un registro. Los elementos que lo componen son los mismos que el get y su explicaciónes la misma

 def put(self, request, pk):

         """Modifica el registro"""

          heroe = Hero.objects.get(id = pk) """Obtengo el registro que deseo modificar"""

          heroe\_serializer = HeroSerializer(heroe, data = request.data) """Lo paso a un serialier, y le entrego al serializer el objeto y la nueva data que le quiero modificar"""

          if heroe\_serializer.is\_valid():  """Validación para que se ejecuten los cambios"""

              heroe\_serializer.save()

              data ={

                'mensage':'El hereo fue creado de forma correcta',

              }

          return Response(

              data=data,

              status=status.HTTP\_200\_OK,

          )

CREAR METODO DELETE: Elimina de forma permanente de la base de datos, por lo general no se elimina, sino que a los registros se le asignan un campo status cuyo valor va a identificar si está activo o inactivo.

 def delete(self, request, pk):

         """Elimina un registro"""

          heroe = Hero.objects.get(id = pk)

          heroe.delete()

          data ={

                'mensage':'El hereo fue eliminado de forma correcta',

          }

          return Response(

                data=data,

                status=status.HTTP\_200\_OK,

          )

Listo puedo ver en el mismo url porque dentro de la nueva vista (class/def) contiene los tres metodos. Allí puedo interactuar con los hereos por particular

MANEJO DE ERRORES :

Metodo para devolver un error y que no se caiga el servicio.

TRY - EXCEPT intenta ejecutar un bloque y si no puede o tiene un problema ejecuta otro bloque.

try:

    hereo = Modelo.objects.get(id=pk)

    heroe\_serializer = HeroSerializer(heroe)

    return Response(

        data = heroe\_serializer.data,

        status=status.HTTP\_200\_OK

    )

except:

      data ={

                'mensaje':'Heroe no encontrado'

            }

      return Response(

                data = data,

                status=status.HTTP\_400\_BAD\_REQUEST

            )

Busco con el primer bloque (try) que me devuelva un heroe en especifico, si no lo encuentra me va a ejecutar el segundo bloque y va a saldar el error, devolviendo un mensaje y sin pararme el servidor

El manejo de error no se debe realizar en un solo metodo Http, se debe hacer

Para evitarlo te trabaja dentro de la class creada en la view para que se pueda acceder al mismo try except de manera global.

try:

      var\_orm = Model.objects.get(id=pk)

except:

      return Response(

          {'mensaje:'mensajeaoifjf'}

          status = status.HTTP\_400\_

)

y luego en el codigo de los diferentes metodosHTTP se le pasa como parametro al ModeloSerializer la var\_orm\_model

var\_serializada = ModelSerializer(var\_orm\_model)

o como si fuera un helper(creo una nueva carpeta dentro del local, con el nombre helpers y dentro un archivo con nombre\_errors

y alli poner la función de try-except:)

Crear archivo \_\_init\_\_.py dentro de la carpeta para que python reconozca que es un archivo que contiene codigo funcional.

1ro importar

from carpeta\_local.models import Model

def hayHeroe(pk):

      try:

           var\_orm\_modelo = Model.objects.get(id = pk)

           return [True, var\_modelo]

      except:

              return [False]

Luego en el archivo view dentro del metodo http creado se pone

1ro se importa:

Helpers

from carpeta\_local.helpers.nombre\_archivo\_con\_funcion\_try\_except import nombre\_funcion

def put(self, request, pk):

       resp =

       if hayHeroe(pk)[0]: """Consulta la condicion en una lista"""

            var\_serializada = ModeloSerializer(hayHeroe(pk)[1], data = request.data)

                                               nombre funcion try except, con la lista en posicion 1 y el mensaje que retorna

            var\_serializada = ModeloSerializer(var\_orm\_modelo, data = request.data)           """Creo que esta no es"""

            if var\_serializada.is\_valid():

                var\_serializada.save()

                data ={

                     'mensage':'El hereo fue modificado de forma correcta',

                }

                return Response(

                    data=data,

                    status=status.HTTP\_200\_OK,

                )

#        return Response(

            data = heroe\_serializer.errors,

            status=status.HTTP\_404\_NOT\_FOUND

        )

Tambien al hayHeroe(pk) del if se lo podria asignar a una var y luego pasar esa var en el if y en la var serializada.

Este helper se puede utilizar de igual manera pasandolo por un if en los otros metodos http.

Se usan mas clases pero:

LOS SERVICIOS SE PUEDEN CREAR COMO CLASES COMO LAS QUE HEMOS HECHO O BASADA EN FUNCIONES A TRAVES DE DECORADORES. Es un @ que indica a python que la funcion que tengamos abajo se va a comportar de cierta forma

Para ello se importa:

from rest\_frameworks.decorators import api\_view

Luego se crea la vista:

@api\_view(['GET'])                """Se coloca que comportamiento tendra la vista segun el/los metodos que se le indiquen"""

def nombre\_api\_view(request):     """Parametros que se requieren"""

      heroes = Hero.objects.all()

      heroes\_serializer = HeroSerializer(heroes, many=True)

      return Response(

           data = heroes\_serializer.data,

           status=status.HTTP\_200\_OK

        )

Usa el mismo codigo que clases, para los metodos.

Luego se debe linkear con las urls

Voy al archivo urls local y allí importo la vista

from carpeta\_local.views import nombre\_vista

Igual para todos los metodos y vistas.

Si utilizo mas de un metodoHttp utilizo un if para acceder a cada uno.

@api\_view(['GET','PUT','DELETE'])                """Se coloca que comportamiento tendra la vista segun el/los metodos que se le indiquen"""

def nombre\_api\_view(request, pk):     """Parametros que se requieren"""

 #Servicio que ofrece

      if request.methon == 'GET':

          codigo del metodo get.

 #Servicio que ofrece

      eliif request.methon == 'PUT':

          codigo del metodo put.

 #Servicio que ofrece

      if request.methon == 'DELETE':

          codigo del metodo delete.

26 - Creación de URLS:

Ir a la carpeta de la app/local/hijo (en este caso heroe) y allí crear un archivo urls.py

Aquí se crearan las urls

El archivo URLS que se encuentra en la carpeta heroesApp (carpeta global/padre) contiene los archivos, en este caso urls globales del proyecto.

y allí se coloca el/los urls que comanden todo el proyecto.

En cambio el archivo urls que esta dentro de la carpeta de la app. Comanda las urls de nuestra app (de maera local)

Crear url local(en la carpeta de la app):

1ro hacer import del modulo path:

from django.urls import path

2do importar servicios/views:

from heroe.views import HeroApiView

from carpeta\_local.views import nombre\_clase\_creada\_en\_el\_archivo\_views.py

# si se acuulan muchas vistas en la importación puedo almacenarlas en una tupla.

Como las views son clases/objetos podemos acceder a sus metodos y propiedades.

3ro Crear URLS:

tiene una estructura:

Local: Hay un url por vistas/class creadas (son metodos independientes)

urlpatterns = [

    path('heroes-list/', HeroApiView.as\_view(), name='heroes\_list'),

path --> porque usa el import de django.urls (

Entre 'nombre\_ruta\_deseada/',

nombre\_clase\_creada\_en\_el\_archivo\_views.py.as\_view() --> Indica que usa la vsta creada con la clase en el archivo views,

name = 'nombre\_descriptivo\_de\_la\_pag'

)

Global:

urlpatterns = [

    path('admin/',             admin.site.urls,            name=),

          'nombre-ruta/',   NombreDeLaClase.metodo(),      name= para linkearle un nombre al url es opcional

          direccion de url, cuando se llame se tiene

                            que ejecutar esa vista con su metodo

Una vez creado el url local, se debe lonkear con el url globarl. Para ello iremos al archivo urls. py pero de la carpeta global

 Ese archivo contiene la documentación para realizar el link.

Importamos el paquete include:

 from django.urls import path, include

urlpatterns = [

    path('admin/', admin.site.urls),

    path('heroes/',            include('heroe.urls')),

]        nombre\_ruta\_local/,  metodo('nombre\_carpeta\_app.nombre\_archivo\_donde\_estan\_las\_rutas')

Vista Django en el navegador:

 heroes/              heroes-list/        [ name='heroes\_list']

indica que entró      indica que entró    Nombre del url local

al url global         al url  local       Identificador.

                      y muestra todos

                      urls disponibles

Guardamos los cambios y volvemos a correr el server.

La vista que nos devuelve el navegador es un 404 con debug=True porque estamos desarrollando el proyecto,

 con las dos rutas que creamos en el proyecto.

Si accedo al url completo con heroes-list, me devuelve en pantalla un error y un mensaje de que espera un http response para retornar una view.

Se puede crear un nuevo archivo.py (con nombre urls/disciptivo) en la carpeta local/hija/app y allí colocarle urls especificas, como por ej: almacenar allí, todos los url de las views de los servicios apis.

Este luego debe ser introducido a en el archivo urls.py de la carpeta global

Se les debe brindar una estructura adecuada. Que sea descriptiva y pueda contener a la mayor parte de los urls similares

27- Debo volver a la pestaña views de la carpeta de la app. Allí modificar el metodo get, en el print setear lo que devuelve para identificar mejor lo que retorna.

Luego le voy a crear un Response(respuesta) para que el front (http) pueda devolver algo.

 Para ello importo un modulo:

from rest\_framework.response import Response

from rest\_framework import status

Luego agregarle al metodo un:

 data = {'Heroes':'hola'}

 return Response(

       data = data | data,

       status=status.tipo\_de\_estado)

data= info que quiera devolver de la base de datos, un mensaje, etc al frontend

data estaá formulado como diccionario.

status= es el codigo que nos devuelve 100,200,300,400,500 que nos indican el estado de la pag.

al request que está dentro de la función print, como es un objeto, puedo acceder a sus metodos y prop por ejemplo un

 .data muestra para ver que info nos devuelve

.method para saber que metodos tiene.

28- Django-Rest nos devuelve en el navegador una vista frontend

Dada por la aplicación django rest, que devuelve las consultas de la api.

Nos dice que es una api, el tipo de metodo, la url, el status,  mensaje.

Tambien se puedehacer desde postmas \*ir al archivo de postman

 Luego ir a Postman para ingresarle el url.

Ya tenemos nuestro primer servicio andando.

29- Darle la funcionalidad adecuada a nuestra Api, en view:

 Trabajamos con la base de datos, a traves de un ORM (capa intermedia que realiza consultas a la base de datos, traduce nuestro codigo y se lo manda a la base de datos)

También estoy conectando elementos del archivo de views con elementos del archivo de models (tablas de bases de datos)

heroes =       Hero.                          objects.           all

              le asigno un modelo             Activa el ORM       Metodos de consultas

              Como es una clase puedo         y puedo acceder     debases de datos

              acceder a sus metodos y prop    a los objetos       que existen. En este caso

                      ===                     del modelo Hero     como quiero traer un listado de

creo una      se le va a asignar                                  todos los objetos de este metodo

variable      lo que contenga en

              la base de datos

voy a realizar un print(heroes) para que me devuelva ese listado.

y en data le asigno a la prop Heroes = la var heroes, para luego poder retornar esa var como respuesta del metodo get usado.

Me salta un error, el cual dice que el Objeto de tipo Hero no es un JSON serializado, esto significa que para que se envíe info entre el back y el front esta info tiene que estar en formato JSON.

Para ello se debe serializar.

Si al print(heroes) le coloco un .values(), me va a dolverver un QuerySet con losvalores que contiene el objeto almacenado en la base de datos, pero me da error porque el formato de esa info es QuerySet y no Jason

Se debe traducir/transformar a JSON, se hace a traves de un serializer.

30-Transformar a Json con serializer:

1ro Crear un nuevo archivo en la carpeta local de la app, que se llame serializer.py

Importar funciones de Rest:

from rest\_framework import serializers

Importar el modelo porque el serializador trabaja al mismo tiempo con el modelo de la base y realiza una doble validación.

Por un lado trabaja con lo que nos devuelve el query set y contrasta con con el tipo de dato que le asignamos al modelo.

from heroe.models import Hero

2do Crear el serializer: que es una clase. Se lo nombra con el nombre de modelo seguido de serializer.

class HeroSerializer(serializers.ModelSerializer):

                     Heredo los metodos de los serializadores y trabajo con ellos.

                     .metodo --> el mas completo es ModelSerializer, esto indica que vamos a trabajar con el serializador a traves del modelo.

      class Meta: """ Es un hiperparametro del serializer es la configuración rincipalde como funciona el serializer y lleva el modelo y los campos que queremos traducir."""

          model = Hero

          fields = '\_\_all\_\_'

                    indica que trabajo con todos los campos del modelo.

      """Se le puede agregar funciones para intervenir en los diferentes metodosHTTP, intercepto los metodos para hacerles modificaciones"""

      def nombre\_funcion(self, validated\_data):

          var\_modelo\_orm = modelo(\*\*validated\_data)

          var\_modelo\_orm.set\_password(validated\_data['password'])

          var\_model\_orm.save()

          return var\_model\_orm

      """Esto me permite encriptar passwords, para crearlas"""

31- Voy a views para activar el serializer:

1ro Importo el serializer

from heroe.serializer import HeroSerializer

Voy a crear una variable que indica a los hereos serializados,

les asigno el serializador y le mando datos, que van a ser la var heroes (el modulo) que son los datos de la base.

Luego hago un print a los heroes serializados para ver que nos devuelve,

junto a un mensaje para ver por terminal que nos devuelve.

Nos devuelve en consola como se compone el QuerySet, es la info del model/modelo.

Para devolver en vez de QuerySet, en formato JSON, el serializador tiene un metodo

print(heroes\_serializer.data)

                       .metodo

También a la función HeroSerializer le voy  agregar el parametro Le agrego el parametro many=True

que indica que duevuelva mas de un valor.

Y ya me devuelve el JSON

procedo a eliminar el data diccionario y a la estructura del Response le pongo en data=heroes\_serializers.data,

# Serializers imports

from heroe.serializer import HeroSerializer

Create your views here.

class HeroApiView(APIView):

    def get(self, request):

        """Retorna un listado con todos los heroes almacenados en la base"""

        # print(f'REQUEST --> {request}')

        heroes = Hero.objects.all()

        # print(heroes.values())

        heroes\_serializer = HeroSerializer(heroes, many=True)

        print(heroes.values())

        print(heroes\_serializer)

        print(heroes\_serializer.data)

        # print(heroes.values())

        # data = {

        #      'Heroes':'hola'

        #  }

        # return response(

        #     data=data,

        #     status=status.HTTP\_200\_OK

        #     )

        return Response(

           data = heroes\_serializer.data,

           status=status.HTTP\_200\_OK

        )

En conclusión solo realizo un metodo/servicio que realiza una llamada a la base, uso un serializador y le doy un response

heroes = Hero.objects.all().values('id',''nombre') """Para solo llamar estos valores"""

Recargo la pagina y el front me va a devolver el Json con los datos solicitados a la base de datos.

32 - Crear una View y asignarle un url, para el front.

Primero voy a views.py de la aplicación o carpeta hija/local.

Allí Voy a importar un http

from django.http import HttpResponse

y voy a importar un render

from django.shortcuts impot render

luego creo la vista con una funcion:

def index(request):  """Función y su nombre, en este caso como es la pag principal se llama index y como parametro le pasamos una request o solicitud"""

   return HttpResponse( """Le voy a solicitar que me retorne el HttpResponse importado y dentro de él paso el codigo html deseado"""

   'Esta es mi primer pagina de Django.<br> <strong> Buen inicio de semana</strong> <hr/>' """Código html"""

 )

2do en el archivo de urls.py de la carpeta local/hija:

Realizar un import de:

Los path de django urls para que habilite la ruta

from django.urls import path """Siempre es así"""

Todas las views para poder acceder a las mismas

from . import views """El punto llama a todas las views, el otro es el archivo views.py, siempre así"""

y luego crear la ruta local correspondiente a la view creada en primera instancia

urlpatterns = [

   path('', views.index, name = 'index')

]

el view.index, es para acceder al archivo views y a el nombre de la función de la view que contiene.

el ponerle un name es opcional.

3ro en el archivo urls.py de la carpeta global voy a

importar desde la carpeta global las urls

from nombre\_carpeta\_global import urls

Verificar que el import de django.urls contenga path,include

from django.urls import path, include

 De las urls de la carpeta global/padre

from hotla\_mundo import urls """from carpeta\_global import archivo\_urls"""

Y veriicar que este el import de

from django.contrib import admin

Añadir la url correspondiente a la view recien creada.

Para ello dentro de la urlpatterns le agrego un path ('nombre\_ruta\_de\_la\_view/', include('nombre\_de\_la\_carpeta\_local.urls'))

33 -Crear Template Con url dinamica

En la carpeta local/hijo crear una carpeta con nombre templates, dentro de templates creamos un archivo con el nombre del template(ej lista.html)

Voy al archivo views.py allí modifico la función creada de la vista, donde la función en este caso catalogo retorne un render, con el archivo templates creado, con un diccionario donde la key es === a la variable dinamica del archivo creado dentro del templates

En views --> def catalogo(request):

                  # return HttpResponse('Pagina de catalogo') """Retorna solo el texto en el front"""

                  return render(request,'lista.html',{"comentario":"este es mi comentario variable"}) """Retorna el contenido dinamico, combinando la key del diccio con la var dinamica del template"""

En templates

dentro del archivo

creado ------------> hello mi primer plantilla

                      <br><b>{{ comentario }}</b>

El valor que contenga la key comentario pertenenciente al diccio de la views va a dibujarse en donde figura la var dinamica {{comentario}} del archivo del templates.

34 - Brindarle la ruta al templates:

en el archivo views en la función creada, a la hora de retornar debe llevar como parametro/argumento que es una request una solicitud y el nombre del archivo templates relacionado, solo el nombre porque el path(la ruta ya la tengo en el setting.py de la carpeta global).

Luego en el archivo settings.py de la carpeta global/padre en el objeto TEMPLATES

en la propiedad DIRS : [BASE\_DIR / 'catalogo/templates']

BASE\_DIR/ --> Indica la ruta base que indica donde se encuentra ubicado localmente. Para no poner el PATH super absoluto, si no el de los modulos.

'ruta\_modulo/templates' --> Indica que acceda a determinado modulo/app/carpeta hija/local y luego a templates para finalmente acceder al archivo que contiene templates.

 TEMPLATES = [

    {

        'BACKEND': 'django.template.backends.django.DjangoTemplates',

        'DIRS': [

            BASE\_DIR / 'catalogo/templates'

        ],

        'APP\_DIRS': True,

        'OPTIONS': {

            'context\_processors': [

                'django.template.context\_processors.debug',

                'django.template.context\_processors.request',

                'django.contrib.auth.context\_processors.auth',

                'django.contrib.messages.context\_processors.messages',

            ],

        },

    },

]

35 - Crear contenido en el archivo creado dentro del template.

En la carpeta global creamos la carpeta templates donde vamos a crear un archivo.html, que se va a utilizar en todas las views/vistas particulares/hijas/locales para evitar repetir codigo como el {% load static %} (para cargar el archivo.css), DOCTYPE, HEAD, BODY, HEADER, MAIN solo con {% block content %} y {% endblock%}, y FOOTER

Ir a la carpeta local y abrir el template.

En el template podemos crear el contenido html, el mismo puede ser dinamico si utilizamos tags, bloques, extens, etc.:

{%extends 'archivo.html' %}  """Para que utilicé el archivo que contiene el {% load static %} para cargar el archivo.css, DOCTYPE, el HEAD y el BODY con el header, el main que estará delimitado por

{% block content %} y {% endblock%} allí se cargará todo el contenido dinamico del main y por último el footer"""

entre los {% block content %} y {% endblock%}, se da la lógica dinamica para estructurar el main, se puede utilizar

{% if isLogged %}Hola, {{ userName }}.{% endif %}  --> En el template.

estos {% if %} y {{}} traeran el contenido dinamico dado en la views a las class/def ya definidas.

 En la view: le doy el contenido dinamico.

def home(request):

    return render(request,'home.html', {"isLogged":1,"userName":"Juanito"})

36 - Para anexar el archivo.css, en la carpeta global vamos a crear una carpeta con el nombre static y dentro vamos a guardar el archivo.css. Tambien haremos lo mismo en la carpeta local/hija.

En el settings en la seccion de Statics files (CSS, JS, Img)

Agregar este objeto que contendrá el path que dirije a los archivos staticos que contienen el css, jso img.

STATICFILES\_DIRS = [

       BASE\_DIR / "static"

]

Este indica de donde se leen los archivos que se publican en:

STATIC\_URL = 'static/'

37- EN SETTINGS.PY en el objeto de templates en la key 'DIRS':[Guardar las rutas creadas]

BASE\_DIR/ --> indica el path absoluto(toda la ruta en duro, desde disco local C:)

'ruta de carpetas globales y - o locales' --> 'carpeta:padre/carpeta\_hija'

 'DIRS': [

            BASE\_DIR / 'CafeteriaDjango/templates/base',

            BASE\_DIR / 'home/templates/home',

37- Ingresar dinamicamente informacipon desde la bbdd al template a traves del view.

En el template, en el doc html, se colocaran variables dinamicas con nombres especificos, en los lugares donde querramos tener la infor dinamica.

En el archivo view debe estar importado el modelo especifico a trabajar, luego se creará una variable que almacene el modelo.orm.get(id=1), ejemplo: productos = Productos.objects.get(id\_producto = 1),

Luego en la misma vista (misma class o misma función), por debajo retorno un render (para ello debe estar importado el render, from django.shortcuts import render) y cmo arg/param le paso la request, el\_template y el contexto.

El contexto serían las variables dinamicas colocadas en el template, van entre llaves {} en fromato diccionario.Ejemplo:

  return render(request,

          'lista.html',

            {

              "nombre\_roducto" : productos.nombre,

             }

y se le asigna la variable que contiene el modelo.orm.get(), de manera que podamos acceder a la información almacenada en ese objeto que trajimos por id, desde la base de datos.

accedemos al dato correspondiente colocando el nombre de esa var.key.valor

38 - Colocar multiples elementos en el template pertenencientes a la misma tabla de la bbdd:

en el archivo view:

creo una variable que le asigno el valor del Modelo.objects.all()

El objects es el orm me permite realizar querys con la bbdd

all() me permite seleccionar todos lo que contenga esa tabla/modelo

luego lo retorno en el render(request, nombre\_plantilla.html, {'var\_dinamica':var\_con\_bbdd\_asignada })

Aquí estaré retornando a la solicitud una determinada plantilla, que contendrá una variable dinamica que tiene asignado la var con el orm

En la planilla/templates voy a crear un codigo html deseado que va a estar entre el tags:

{% for i in var\_dinamica %}

{{i.campo}}

{% endfor %}

El for me permite recorrer todo el modelo y la var local creada en el for, en este caso i, acceder a cada valor almacenado según campo.

Se va a dibujar el html por la cantidad de datos almacenados en el modelo.

Personalización del admin:

En el archivo models, cualdo defino el modelo, voy a configurar como quiero que se muestre esta info en el admin.

Vamos a realizar una nueva función

def \_\_str\_\_(self):            """Para que me aparezca en nombre en el admin"""

      return self.name

class Meta:                           """Para indicar que es plural y singular"""

  verbose\_name = 'Heroe'

  verbose\_name\_plural = 'Heroes'

Estructuración de carpeta Django: Pautas Grales.

Carpeta principal contenedora de toda la APP ---> Nombre\_custom

Archivo de requerimientos para saber con que versiones y modulos/paquetes estoy usando.

Archivo README ---> Brinda info

.gitignore ---> Ignora los archivos que no queremos que no sean trackeadas por git.

env ---> Entorno Virtual (debe estar ignorada).

Carpet Global con el nombre de ---> Backend

contiene:

 la carpeta de settings, con los archivos base.py (contiene secret key, aplicaciones definitions, Templates, root, from .base import \* ), \_\_init\_\_.py, local.py (contiene las configuraciones cuando estoy desarrollando en mi maquina, database, debug, statics con un import from .base import \* ) y production.py (contiene from .base import \* )

 Los archivos urls.py, manage.py, wsgi.py, asgi.py, \_\_init\_\_.py

 Carpeta local, contenedora de las apps  ---> Nombre\_custom por lo gral apps. Hay que tener en cuenta que al tener una carpeta contenedora de las apps, en el archivo apps.py de cada aplicación, dentro de la class en la pro name= debe llevar apps.(el nombre de la carpeta)

 Contiene dentro

Migrations

La carpeta api que contiene los archivos nucleos de la app. que son \_\_init\_\_.py, serializer.py, urls.py, views.py.

Filtros orm --> Query ser API reference listado de todos los filtros para obtener datos de la base de datos.

AUTENTICACIÖN PARA MANEJAR METODOS SERVICIOS API HTTP CON REST:

en documentacion adding login to the browsable api

JWT ---> Documentación en ---> coffeebytes.dev/django-rest-framework-y-jwt-para-autentificar

Tenes que ingresar usuario y contraseña, el servicio al llamarlo te solicita un token, si el no podes realizar ninguna acción.

el servicio al recibir la autenticación te responde, por lo gral con un json.

primero en cmd dentro de la carpeta que se ubica el manage.py se debe instalar --->

pip install djangorestframework\_simplejwt

Dentro del settings.py se debe registrar la extención que se instaló, en la sección de thrid apps --->

'djangorestframework\_simplejwt'

Y al fondo del settings se copia este objeto --->

REST\_FRAMEWORK = {

    'DEFAULT\_AUTHENTICATION\_CLASSES': [

        'rest\_framework\_simplejwt.authentication.JWTAuthentication',

    ],

}

Modificar los valores por defecto de los JWT

Para hacerlo vamos al archivo de configuraciones de Django y creamos una variable llamada SIMPLE\_JWT, en la que podemos

sobreescribir los datos que querramos y colocarles la duración que más te convenga.

from datetime import timedelta

SIMPLE\_JWT = {

    'ACCESS\_TOKEN\_LIFETIME': timedelta(minutes=5),

    'REFRESH\_TOKEN\_LIFETIME': timedelta(days=1),

    ...}

 Se deben volver a ejecutar los migrates ---> migrate y makemigrations

y en caso de no tener un super user crear uno.

En el archivo urls.py de la carpeta global/padre

realizar un import --->

from rest\_framework\_simplejwt import views as jwt\_views

registrar estos dos path dentro del urlpatterns--->

path('api/token/', jwt\_views.TokenObtainPairView.as\_view(), name='token\_obtain\_pair'),

path('api/token/refresh/', jwt\_views.TokenRefreshView.as\_view(), name='token\_refresh'),

La primera vista nos devolverá un par de tokens; uno de acceso y otro para refrescar el primero.

La segunda vista nos servirá para refrescar o actualizar el token de acceso.

Debemos crear la vistas que estarán protegidas con el token  que sea accesible únicamente a los usuarios autenticados.

Se crea una vista de ejemplo dentro del archivo urls.py de la carpeta global. para comprobar el correcto funcionamiento.

Verificar que estén todos los imports

 from django.contrib import admin

from django.urls import path

from rest\_framework\_simplejwt import views as jwt\_views

from rest\_framework.response import Response

from rest\_framework.views import APIView

from rest\_framework.permissions import IsAuthenticated

"""Vista basada en clases que recibe la autentificación por el permision\_classes y luego define por funcion el metodo servicio http get"""

class Protegida(APIView):

    permission\_classes = [IsAuthenticated]

        def get(self, request):

        return Response({"content": "Esta vista está protegida"})

"""Se colocan los dos path nuevos y el nuevo path correspondiente a la vista protegida creada."""

urlpatterns = [

    path('admin/', admin.site.urls),

    path('api/token/', jwt\_views.TokenObtainPairView.as\_view(), name='token\_obtain\_pair'),

    path('api/token/refresh/', jwt\_views.TokenRefreshView.as\_view(), name='token\_refresh'),

    path('protegida/', Protegida.as\_view(), name='protegida')

]

En el caso que ya tenga una vista creada en el archivo views, defino la vista por class o @ def y coloco en la linea siguiente --->

permission\_classes = [IsAuthenticated]

luego continúo con la lógica del servicio/vista.

Mediante postman podemos conseguir el token --->

primero se debe crear un metodo post pasarle el url del token --->

http://127.0.0.1:8000/api/token/

luego pasarle los datos mediante el body con form-data --->

username -->

password -->

y enviar el metodo. Nos devuelve el token acces y el token refresh.

Lo vamos a copiar al token acces y pegarlo en el/los metodos/servicios HTTP APi que me lo solicite -->

para ello lo debo copiar en la pestaña de authorizations y seleccionar la opcion del combo Bearer Token y

en el input de la derecha reemplazar el token viejo por el nuevo. De esa manera consigo la autorización para obtener la respuesta del metodo/servicio http api.