***Django REST framework: mi primer REST API***

En este curso **crearemos una REST API CRUD utilizando SQLite3 para el desarrollo y PostgreSQL para la producción, y por último como desplegarlo de forma gratuita en un servicio llamado render.com**

En primer lugar, **Qué es una API?**

Las **API’s (Interfaces de Programación de Aplicaciones)** son utilizadas para **conectar el backend con el frontend de una aplicación**

El backend se refiere a la parte de la aplicación que se ejecuta en el servidor y se encarga de procesar las solicitudes, manejar la lógica de negocio y acceder a la base de datos u otros servicios

Las API’s proporcionan una capa de abstracción entre el backend y el frontend, **permitiendo que ambos se comuniquen y compartan datos de manera estructurada**. El **backend expone una o varias API’s que definen los puntos de acceso y las operaciones disponibles para que el frontend pueda consumirlas**. El frontend, a través de llamadas a estas API’s, puede solicitar datos, enviar información al servidor y realizar otras acciones necesarias para interactuar con la aplicación.

**La API define cómo se debe estructurar la solicitud y qué se espera recibir como respuesta**. Esto puede implicar la especificación de los parámetros de entrada, el formato de los datos y los métodos HTTP utilizados para la comunicación (como GET, POST, PUT, DELETE, entre otros)

**Qué es una REST API CRUD?**

Una **REST API CRUD** **es una interfaz de programación de aplicaciones** (API) que **sigue los principios del estilo arquitectónico REST** (Representational State Transfer) **y permite realizar operaciones CRUD** (Crear, Leer, Actualizar y Eliminar) **en un sistema o servicio web**.

El acrónimo CRUD representa las operaciones básicas que se pueden realizar sobre los datos de una aplicación:

* Crear **(Create)**: Permite **agregar nuevos datos** a un sistema o servicio. En el contexto de una REST API, esto se realiza mediante una **solicitud HTTP de tipo POST enviando los datos que se desean crear**.
* Leer **(Read)**: Permite **recuperar datos existentes** de un sistema o servicio. En una REST API, esto se realiza mediante una **solicitud HTTP de tipo GET para obtener los datos específicos o una colección de datos**.
* Actualizar **(Update)**: Permite **modificar datos existentes** en un sistema o servicio. En una REST API, esto se logra mediante una **solicitud HTTP de tipo PUT o PATCH que envía los datos actualizados**.
* Eliminar **(Delete)**: Permite **eliminar datos existentes** de un sistema o servicio. En una REST API, esto se realiza mediante una **solicitud HTTP de tipo DELETE para eliminar los datos específicos**

**Project Setup (Configuración del Proyecto)**

**Creamos la carpeta del proyecto**, en este caso la llamaremos Django REST API CRUD, **y la pegamos dentro del editor de código** Visual Studio Code

En primer lugar **crearemos un entorno virtual**, y de esta forma poder aislar los paquetes de los demás proyectos que realicemos en otros entornos

**Como crear un entorno virtual**

Para eso **abrimos la consola** (F1 y create new terminal) **e instalamos desde la consola el entorno virtual escribiendo pip install virtualenv**

Luego debemos **ejecutar ese entorno virtual**, para eso **en la consola escribimos:**

**python -m virtualenv venv (tenemos que cambiar el guión y ponerlo nosotros en VSC ya que si no nos tira error)**

Ahora nos falta **activar ese entorno virtual**, para eso:

**F1** y escribimos **select interpreter**, allí **seleccionamos el intérprete que nos salga como recomendado con una estrella**, ya que será el que el editor de código reconozca como un entorno virtual

De esta forma cerramos la terminal actual y **cada vez que abramos una consola estaremos trabajando desde este entorno virtual**

\***En el caso de que esto nos genere problemas**, podemos abrir el entorno virtual cuando querramos usando:

**.\venv\Scripts\activate**

**El backslash (\) lo hacemos con ALT + 92**

Lo que debemos saber es que **cuando abramos una nueva terminal deberemos ejecutar de nuevo ese comando**

**Como instalar los módulos necesarios**

Ahora **instalaremos el módulo de Django** (**pip install django**) **y el de Django REST framework** (**pip install djangorestframework**)

**Como crear el proyecto**

En la consola escribimos **django–admin startproject nombre\_del\_proyecto . (poner nosotros el guión)**

**Debemos poner el punto al final para que el editor de código entienda que debe crear este proyecto en el directorio actual**

**Como crear una app**

En la consola escribimos **python manage.py startapp nombre\_de\_la\_app**

Ahora vamos al archivo **settings.py** **de la carpeta del proyecto**, **y en la lista INSTALLED\_APPS agregamos el nombre de nuestra app. Además debemos agregar el módulo rest\_framework**

En este caso:

INSTALLED\_APPS = [

    'django.contrib.admin',

    'django.contrib.auth',

    'django.contrib.contenttypes',

    'django.contrib.sessions',

    'django.contrib.messages',

    'django.contrib.staticfiles',

    #esta es la app que creamos

    'MiApp',

    #este es el modulo rest\_framework

    'rest\_framework',

]

**Models (Modelos)**

**Como crear un modelo de proyecto**

**Crearemos este modelo para poder generar la tabla en la base de datos**, para poder luego crear una API

Para esto iremos al archivo **Models.py de la carpeta de la app** y **crearemos una clase llamada Project**, la cual contendrá 4 objetos en este caso: title, description, technology y created\_at

#creamos la tabla Project en la base de datos, la misma tendrá 4 columnas

class Project(models.Model) :

    #el título sera un string de máximo 200 caracteres

    title = models.CharField(max\_length=200)

    #la descripción será un texto extenso, por ende es un TextField

    description = models.TextField()

    #para saber que tecnología se usó. es un string de máximo 200 caracteres

    technology = models.CharField(max\_length=200)

    #para saber cuando se creó el proyecto. con esto hacemos que la fecha y hora se ponga automáticamente al crear el proyecto y el parámetro auto\_now\_add=True hace que se guarden automáticamente estos datos

    created\_at = models.DateTimeField(auto\_now\_add=True)

Para **crear la base de datos debemos ejecutar el servidor** usando:

**Python manage.py runserver**

Y luego debemos **cargar los modelos o tablas ejecutando las migraciones correspondientes** a través de los comandos:

**Python manage.py makemigrations**

**Python manage.py migrate**

**Esto permitirá que se actualice la base de datos db.sqlite3 y se guarden las tablas que creamos**

**API (Interfaces de Programación de Aplicaciones)**

Como dijimos al principio, **las API’s son las encargadas de conectar el backend con el frontend**

La documentación de Django REST framework nos pide que cuando ejecutemos el modulo principal creemos un ViewSet y un serializer

**Que se necesita para crear API’s?**

En Django REST framework**, los ViewSets y los serializers son componentes fundamentales para construir APIs**

Los **ViewSets** son una abstracción de alto nivel que **permiten agrupar la lógica relacionada con las vistas en un solo lugar**. Proporcionan una forma conveniente de **definir las operaciones CRUD** (Create, Retrieve, Update, Delete) **en un conjunto de datos determinado**.

Los ViewSets **manejan automáticamente las rutas y las acciones HTTP asociadas a las operaciones CRUD**, como GET, POST, PUT y DELETE. Esto significa que puedes **definir un conjunto de vistas relacionadas en un solo lugar, en lugar de tener que escribir vistas separadas para cada operación**

Los **serializers** en Django REST framework **son responsables de convertir los objetos de Django en formatos de datos legibles, como JSON, y viceversa**. Los serializers se utilizan para **validar los datos de entrada recibidos en las solicitudes HTTP, así como para serializar y deserializar objetos en las respuestas**

Podemos **especificar los campos que deseamos incluir, los campos requeridos, las validaciones y las relaciones con otros modelos**. Además, los serializers también pueden realizar validaciones personalizadas en los datos antes de guardarlos en la base de datos

Esto nos permitirá **convertir los datos de Python en objetos Json que luego podrán ser consultados desde el cliente, y además nos permite administrar quién puede ver estos datos**

**Como crear API’s**

**Crear los serializers**

En primer lugar **creamos un archivo llamado serializers.py en la carpeta de la app**

#en este archivo crearemos los serializers, los cuales junto con el ViewSet creado en api.py nos permitirán crear API's

#importamos el módulo serializers

from rest\_framework import serializers

#importamos el modelo Project

from .models import Project

#DESDE ESTE ARCHIVO, DJANGO SABRÁ QUE RESPONDER CUANDO SE HAGA UNA PETICIÓN POST, GET, Put y Delete

#creamos la clase que nos permita convertir el modelo en datos que puedan ser consultados

class ProjectSerializers(serializers.ModelSerializer) :

    class Meta :

        #en model ponemos el nombre de la app

        model = Project

        #ahora ponemos los campos que serán consultados dentro de una tupla. Los mismos serán los que tenemos dentro de la clase Project en el archivo models.py. También podemos verlos en las migraciones de la app Project

        fields = ('id', 'title', 'description', 'technology', 'created\_at')

        #ahora ponemos los campos de solo lectura, osea los que no queremos que el usuario pueda modificar

       read\_only\_fields = ('created\_at', ) #debemos poner una coma para que no lo lea como string y no nos tire errores al ejecutar la web

**Crear el ViewSet**

Luego en la misma **carpeta de la app creamos un archivo llamado api.py**, **en el cual creamos el ViewSet**

#En este archivo crearemos el ViewSet, que junto con los serializers nos permiten crear API's

#importamos el modelo Project

from .models import Project

#importamos viewsets y permissions

from rest\_framework import viewsets, permissions

#importamos el modelo del serializer

from .serializers import ProjectSerializers

#creamos el ViewSet del modelo Project, el cuál heredará lo que ponemos dentro del parámetro

class ProjectViewSet (viewsets.ModelViewSet) :

    #ponemos las consultas que se podrán hacer

    #hacemos que el conjunto de datos contenga todos los datos de la tabla Project

    queryset = Project.objects.all()

    #ponemos en una lista quienes tendrán permisos (en este caso AllowAny significa que todos tendrán permiso)

    permission\_classes = [permissions.AllowAny]

    #ponemos desde qué serializer convertirá los datos

    serializer\_class = ProjectSerializers

De esta forma **ya tenemos creada nuestra API, solo nos falta utilizarla a través de una ruta URL para que el cliente pueda consultarla**

**Enlazar la API a una ruta URL**

Podemos hacerlo de dos formas: la forma manual (a través de crear un archivo urls.py y agregar las rutas de forma individual a la lista URLPATTERNS) o **mediante un módulo de REST framework llamado routers que se encarga de crear todas las rutas básicas o el CRUD de la API**

Para hacerlo de esta forma, la cual es la forma más óptima, debemos crear un archivo urls.py dentro de la carpeta de la app y poner:

#En este archivo crearemos las rutas de la API de forma sencilla usando el módulo routers de Django REST framework

#importamos el módulo routers

from rest\_framework import routers

#importamos el ViewSet del modelo Project

from .api import ProjectViewSet

#la variable router toma el valor de DefaultRouter(), el cual crea el CRUD

router = routers.DefaultRouter()

#registraremos en primer lugar el nombre de la ruta y luego el nombre del ViewSet (podemos agregarle como en este caso el nombre de esta ruta)

router.register('api/projects', ProjectViewSet, 'projects')

#urlpatterns tomará todas las urls generadas por router

urlpatterns = router.urls

**Enlazar la app con esas rutas**

Para eso vamos a la **carpeta del proyecto, al archivo urls.py**, y lo modificamos de esta forma:

from django.contrib import admin

from django.urls import path, include #importamos el módulo include

#agregamos las URL's que queremos enlazar

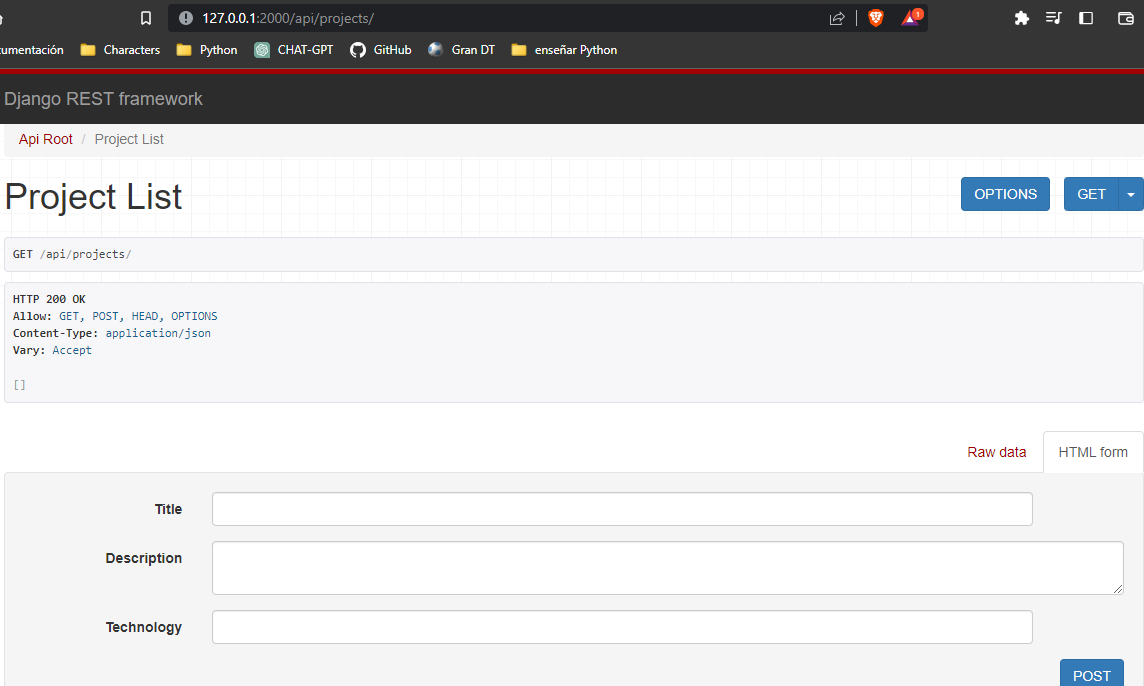
urlpatterns = [

    path('admin/', admin.site.urls),

    path('', include('MyApp.urls')),

]

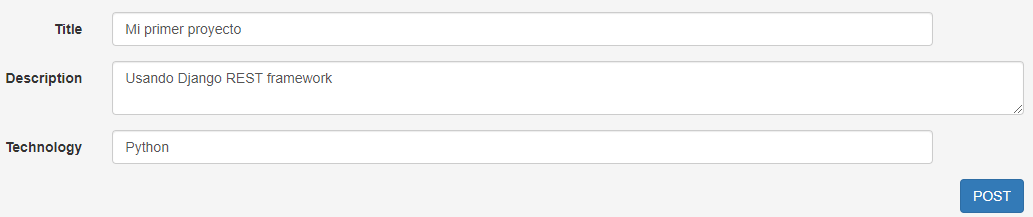
Si visitamos a través del **método GET (desde ingresar las rutas URL escribiéndolas en el buscador)** la ruta api/projects veremos lo siguiente:



**A medida que vayamos enviando peticiones se irá actualizando el array que se muestra en la página** (en este caso aún está vacío ya que no se ha enviado ninguna petición aún)

Ahora veremos dos formas de enviar peticiones POST, PUT y DELETE

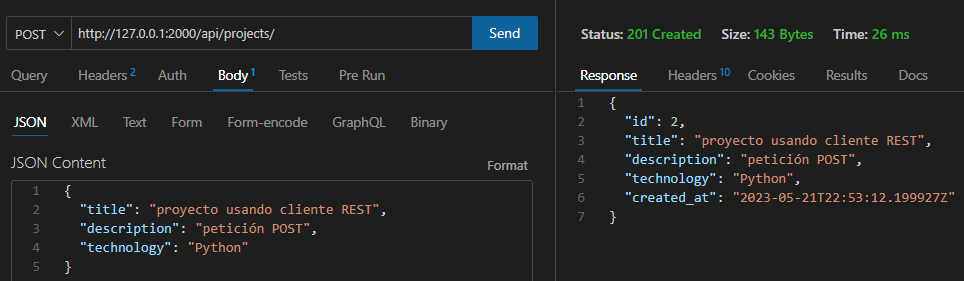
**Formas de enviar peticiones POST, PUT y DELETE**

Una forma es **usar la interfaz del formulario que creamos en la página**

La otra forma es **utilizar un cliente REST**

Para eso **instalaremos la extensión del cliente REST** (en este caso usaremos **Thunder Client**)

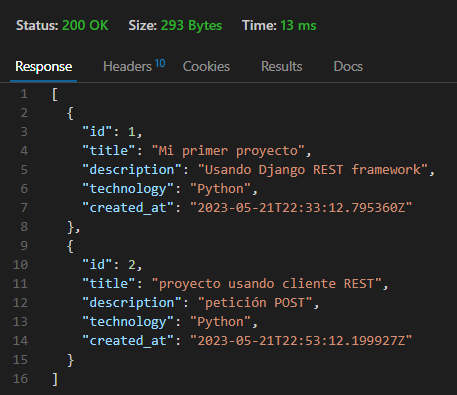
Al instalarlo **se nos creará un ícono** en la parte izquierda del Visual Studio

**Para crear una petición desde allí debemos ingresar a ese ícono, a la parte de new request**, y allí ingresar que el **método será POST, ingresar la URL, y en la sección Body escribir en forma de diccionario los campos que queremos enviar mediante la petición POST**

**Aclaración: a la derecha nos muestra la petición enviada. Vemos que le puso automáticamente un ID y el campo de created\_at se ingresó de forma automática**

**Como ver todas las peticiones que se han hecho**

Si ponemos que la **petición sea GET y le pasamos la URL nos devolverá todas las peticiones que se han hecho hasta ese momento**

En este caso tenemos dos peticiones: una a través de la interfaz del formulario en el navegador y otra a través del cliente REST

**Los ID se pusieron de forma automática en cada petición**

**Si queremos que nos muestre una sola petición debemos**

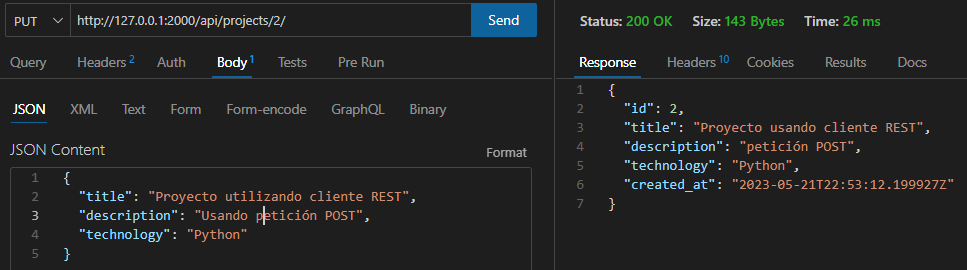
**A agregarle el nro de ID luego de la URL**

**Como borrar peticiones**

**Usamos la opción DELETE si estamos en alguna petición habiendo usado la opción GET y ponemos SEND**

**Como actualizar las peticiones**

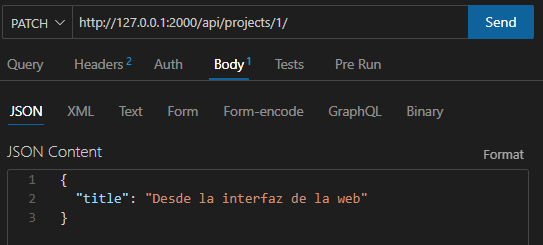
Si queremos **actualizar algún campo de la petición tenemos que seleccionar la opción PUT y buscar a través de la URL el nro de ID de la petición a la que queremos actualizar alguno de sus datos**

Una vez hecho esto **modificamos el valor que querramos actualizar y ponemos SEND**

En este caso actualizamos el title y la description de la petición con ID 2

A la izquierda está como lo actualizaremos y a la derecha como estaba antes de actualizarlo hasta que pongamos SEND

**Si queremos actualizar solo un campo** es mejor utilizar **PATCH**

Ej: queremos actualizar el campo title de la petición con ID 1

**Hasta acá ya tenemos creada nuestra API, pero ya no queremos ejecutarla en el localhost, sino que queremos desplegarla y ejecutarla en un servidor**

**Ahora veremos cómo desplegar nuestra REST API creada con Django REST framework**

**Deploy (Despliegue)**

En este caso usaremos un **servidor gratuito llamado Render.com** (hay otros, como por ejemplo Heroku, etc)

Este servicio permite desplegar sitios estáticos, servicios web, bases de datos con PostgreSQL, aplicaciones en segundo plano o con intervalos de tiempo, etc

**Como desplegar en Render.com**

En nuestro caso, **como hemos creado un Backend a través de Django, usaremos la opción New Web Service** (Nuevo Servicio Web)

**Para poder desplegar nuestra API es obligatorio crear un repositorio de Git para poder subirlo a GitHub**

**Instalar Git**

**Si no lo tenemos instalado aún debemos bajarlo desde su sitio**

Añade Git a la variable de entorno del sistema: Si Git está instalado pero no se encuentra en la variable de entorno del sistema, debemos agregarlo manualmente :

* En Windows:
  + Haz clic derecho en "Este equipo" o "Mi PC" y selecciona "Propiedades".
  + Haz clic en "Configuración avanzada del sistema" y luego en "Variables de entorno".
  + En la sección "Variables del sistema", busca la variable "Path" y haz clic en "Editar".
  + Haz clic en "Nuevo" y agrega la ruta de instalación de Git (por ejemplo, "C:\Program Files\Git\bin").
  + Haz clic en "Aceptar" para guardar los cambios.
  + Reinicia la línea de comandos o terminal para que los cambios surtan efecto.

Una vez que hayamos instalado Git correctamente y lo hayamos añadido a la variable de entorno del sistema, deberíamos poder ejecutar el comando **git init** sin recibir errores

**Inicializar un repositorio de GitHub en la consola de Visual Studio**

**Importante: si al ejecutar este u otro script nos muestra un error llamado “Error: X: El término X no se reconoce como nombre de un cmdlet, función, archivo...” debemos instalar a través del CMD poniendo pip install typescript**

En la consola escribimos **git init** para iniciar el repositorio de GitHub

Esto nos pondrá todos los archivos en verde, ya que por defecto selecciona todos para enviarlos al repositorio

Como nosotros **no queremos enviar todos**, necesitamos crear un archivo en la carpeta global llamado **.gitignore** en el cuál **pondremos los archivos que queremos que el repositorio ignore**

#puede que nos genere problemas o el mismo servidor borre la base de datos dsp de un tiempo, usaremos una database de Postgre

db.sqlite3

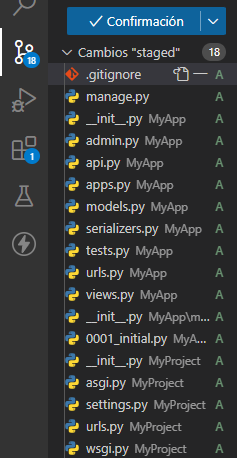
venv

#ya que es solo para funcionamiento interno de Python

\_\_pycache\_\_

#TODO LO QUE NO ESTÉ EN ESTE ARCHIVO SERÁ PARTE DEL REPOSITORIO INICIAL

Ahora en la consola ponemos **git add .** para seleccionar todos los archivos que sigan en verde

Una vez hecho esto, **en la pestaña de Control de Código Fuente veremos una lista con todos los archivos que se van a exportar al repositorio**

Luego ponemos en la consola **git commit -m “first commit”**

al ejecutar el comando **git commit -m “first commit”**, se crea un nuevo commit en el repositorio con el mensaje "first commit". Este mensaje podría indicar que este commit representa la primera versión inicial del proyecto o la primera serie de cambios importantes realizados en el proyecto

Importante: si nos tira el error **"fatal: unable to auto-detect email address"** debemos poner en el CMD los comandos

**git config --global user.email "schonhalsemiliano@gmail.com"**

**git config --global user.name "EmiSchonhals1"**

De esta forma ya **tenemos creado el repositorio, solo falta configurarlo para poder subirlo a Render.com**

**Configurar el repositorio ya creado para subirlo a Render.com**

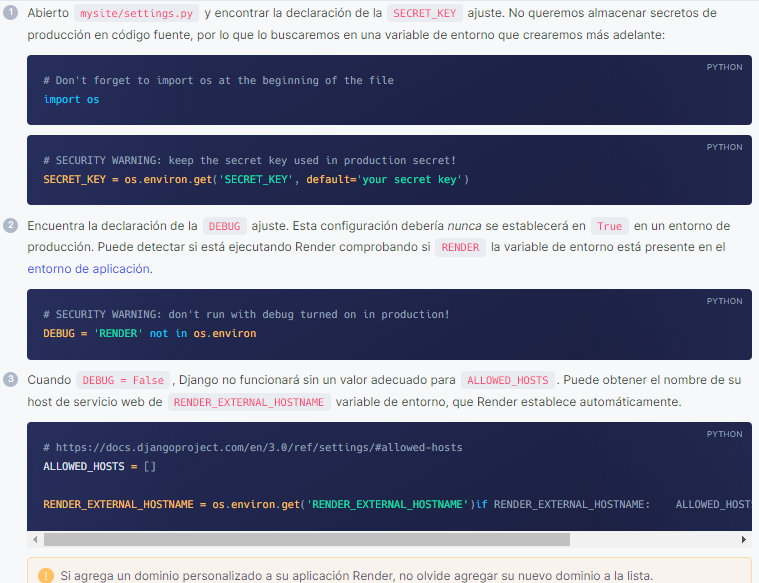
Tenemos una guía si ponemos en el buscador **“Deploy django on render”** en el que nos mostrará el paso a paso de cómo crear y desplegar nuestra API en este servicio

**https://render.com/docs/deploy-django#update-your-app-for-render**

**Actualizar para renderizar**

Como nosotros ya creamos la API y el repositorio nos vamos a centrar en la parte de la página llamada **“Update Your App For Render”**

Lo primero que nos muestra es que debemos ir al archivo **settings.py** de la carpeta del proyecto e **importar os y actualizar la variable secret\_key** (esta es una variable de Django que se encarga de asegurar los datos que envíe el cliente), luego debemos seguir los demás pasos:



Una vez terminados estos cambios pasamos a la siguiente parte llamada:

**Configurar Django para PostgreSQL**

Vamos a la sección Dashboard de render.com y allí ingresamos donde dice **New PostgreSQL**

Por defecto usaremos el plan gratuito que nos brinda la página

Nos pedirá ingresar un name, un nombre para la base de datos y otras cosas más (la sección user podemos dejarla en random si queremos)

Una vez completado ponemos crear base de datos

**Aclaración: en Render.com las bases de datos gratuitas tienen 90 días de validez, si no se actualizan a un paquete pago se eliminan**

Una vez que creamos la base de datos en Render nos mostrará una serie de URL’s que usaremos más adelante, mientras tanto seguiremos con el proceso

Ahora según la documentación de Render debemos hacer lo siguiente:

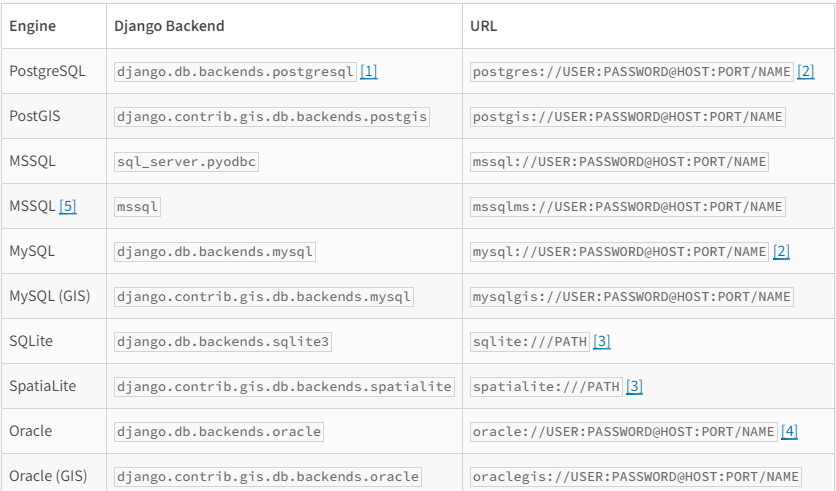


**Aclaración: como estamos trabajando con la consola del entorno virtual, debemos instalar los paquetes del paso 1 usando pip install**, no usando $ poetry add

Una vez instaladas estas dependencias pasamos al paso 2

Durante el paso 2 tendremos que copiar los valores que muestra la página (en la imagen anterior no se llega a ver todo pero hay que copiar todo menos el comentario del medio)

Sin embargo hay que hacerle un retoque, ya que estamos usando **SQLite como base de datos**

Si visitamos la URL dj database url tendremos un cuadro donde nos muestran todas las bases de datos y sus respectivas URL’s para pegar en DATABASES

Al modificar DATABASES quedaría así:

# si por defecto no encuentra la variable Database\_url le asignará lo que le pusimos como parámetros

DATABASES = {

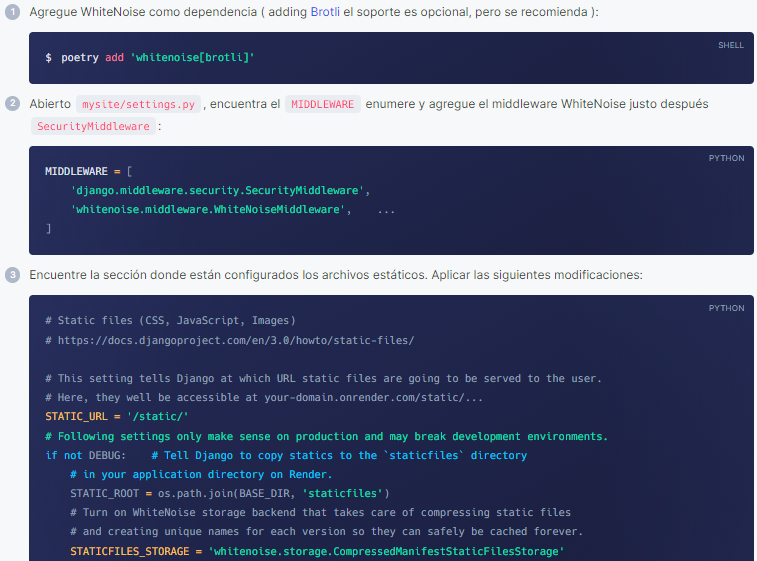
    'default': dj\_database\_url.config(

        default='sqlite:///db.sqlite3',

        conn\_max\_age=600

    )

}

**Archivos Estáticos**

**Aclaración: como estamos trabajando con la consola del entorno virtual, debemos instalar los paquetes usando pip install**, no usando $ poetry add. Y además **copiamos el módulo sin las comillas ‘ ’**

En el paso 2 debemos **pegar el módulo justo debajo del primer módulo que está en la lista MIDDLEWARE**

MIDDLEWARE = [

    'django.middleware.security.SecurityMiddleware',

    'whitenoise.middleware.WhiteNoiseMiddleware',

. . .

. . .

En el paso 3 pegamos lo que está debajo de STATIC\_URL

STATIC\_URL = 'static/'

if not DEBUG:

    STATIC\_ROOT = os.path.join(BASE\_DIR, 'staticfiles')

    STATICFILES\_STORAGE = 'whitenoise.storage.CompressedManifestStaticFilesStorage'

**Crear un Build Script**

Un build script (también conocido como **script de compilación** o script de construcción) es un archivo o conjunto de instrucciones utilizado para **automatizar el proceso de compilación y construcción de un proyecto de software**. Se utiliza comúnmente en proyectos de desarrollo de software para ejecutar una serie de tareas necesarias para convertir el código fuente en un producto final ejecutable.

El build script **contiene una serie de comandos o instrucciones que indican cómo compilar, enlazar y empaquetar el software**

En la **carpeta principal creamos un archivo llamado build.sh**

El archivo "build.sh" es un script utilizado en el desarrollo de software y proyectos de programación. Este archivo suele utilizarse para automatizar el proceso de construcción de una aplicación o proyecto.

El propósito principal del archivo build.sh es compilar, construir y preparar el entorno necesario para ejecutar la aplicación o proyecto. El script puede contener una serie de comandos y configuraciones específicas para llevar a cabo estas tareas, como la compilación de código fuente, la generación de archivos ejecutables, la instalación de dependencias y la configuración del entorno de ejecución

Es importante tener en cuenta que la funcionalidad y el propósito exactos del archivo **build.sh** pueden variar dependiendo del proyecto y del entorno de desarrollo utilizado. Cada proyecto puede personalizar el archivo build.sh para adaptarse a sus necesidades específicas de construcción y configuración

Primero debemos poner en la **consola** **pip freeze > requirements.txt**

Esto **creará en la carpeta principal un archivo llamado requirements.txt en el que dentro tiene todos los módulos que hemos usado**

Ahora, en el archivo **build.sh**, **pegamos los comandos que están en la documentación de render que venimos usando**, lo único que cambiaremos será que al estar en Django no usamos poetry para instalar, **usamos pip install**

#!/usr/bin/env bash

# exit on error

set -o errexit

#para ejecutar requirements.txt

pip install -r requirements.txt

#para que se genere la carpeta por defecto de archivos estáticos

python manage.py collectstatic --no-input

#para ejecutar las migraciones

python manage.py migrate

Luego debemos **darle permisos de ejecutable a este archivo, para eso abriremos una nueva consola y tocaremos en el +, allí elegiremos la consola Git Bash**

En la consola **Git Bash** ponemos **chmod a+x build.sh**

De esta forma el archivo **build.sh ya será un ejecutable**

Ahora necesitamos usar **gunicorn** para servir contenido estático, como imágenes, css, etc

Para eso volvemos a la **consola normal de Python** y ponemos **pip install gunicorn**

Por último debemos **añadir el módulo gunicorn al archivo requirements.txt**

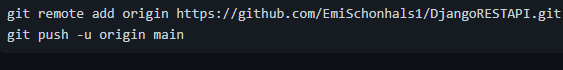
Para eso en la **consola** ponemos **pip freeze > requirements.txt**

En control de código fuente tendremos los 2 archivos que mandaremos al repositorio y el archivo que actualizaremos también

En la **consola** ponemos **git status**, luego **git add .** y por último **git commit –m “Listo para render.com”**

Ahora necesitamos **crear un repositorio en GitHub**

Para eso vamos a GitHub, ponemos crear nuevo repositorio y **lo creamos**

Luego vamos a **pegar en la consola dos comandos** que nos aparecen luego de crear el repositorio en GitHub:

**Aclaración: en Django la rama por defecto se llama master, no main. Por ende hay que modificar eso en el último comando**

Con esto **ya tenemos creado nuestro repositorio en GitHub**

Lo que falta es **conectar nuestro repositorio de GitHub con el servicio de Render.com**

Para eso debemos ir a GitHub desde el link de render, entrar en el repositorio, ir a settings, luego allí vamos a GitHub Apps y allí ponemos configure

Una vez dentro **seleccionamos el repositorio que queremos que se agregue a la vista de la gente** y listo

Importante: Si ya has conectado GitHub a Render, pero aún experimentas el error, es posible que la aplicación de Render en GitHub esté mal configurada. Esto puede ocurrir si la aplicación está configurada para el conjunto incorrecto de repositorios o si un repositorio que antes era público se ha vuelto privado [Fuente 1]. Para solucionar esto, puedes administrar los permisos que has otorgado a la aplicación de Render en GitHub. Visita <https://github.com/apps/render/installations/new> para administrar los permisos. Si la aplicación no está instalada, puedes instalarla

En **Create New Web Service** tendremos agregado de esta forma nuestro repositorio, por lo que nos resta ponerle connect, y de esta forma terminaremos de desplegarlo

Para terminar de desplegarlo debemos completar el formulario que nos sale

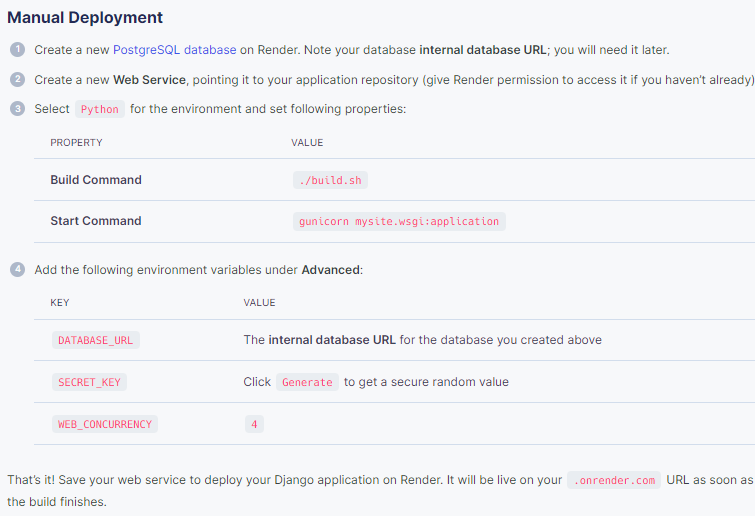
Aclaración: donde dice ***Build Command*** lo modificamos y ponemos **./build.sh**

También, en ***Start Command*** lo modificamos y ponemos **gunicorn nombre\_carpetaproyecto.wsgi**

Luego de eso creamos el web Service

Importante: esto nos va a tirar error ya que nos falta es crear las variables de entorno

Para arreglar esto vamos a ***environment dentro del servicio web*** que creamos y agregamos las variables de entorno necesarias (**add environment variable**)

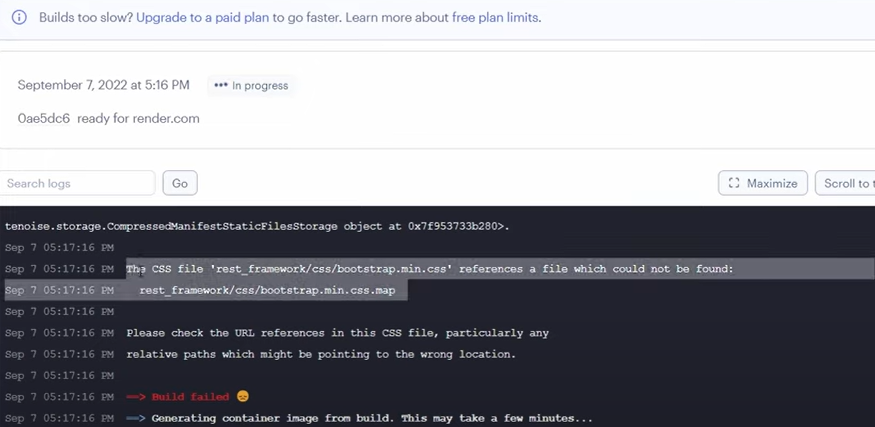
Estas las encontramos en la documentación de render.com

Cuando pongamos la variable **DATABASE\_URL**, el valor que debemos ponerle lo encontramos al entrar a la base de datos que creamos primero en render y copiar **Internal Database URL**. Esta URL nos permite conectarnos desde nuestra app a la base de datos

La variable **SECRET\_KEY** podemos obviarla si queremos, ya que ya tenemos una secret key por defecto en Django. Pero si queremos ponerla igual buscamos cualquier secret key de internet y la agregamos. Las secret key sirven para mantener seguro nuestro sitio

Lo que debemos agregarle (que no aparece en esta documentación de render) es la versión de python que estamos usando. Para eso en la key ponemos **PYTHON\_VERSION** y en el valor ponemos nuestra versión de python (en la consola escribimos **python --version**) y pegamos los números en el value

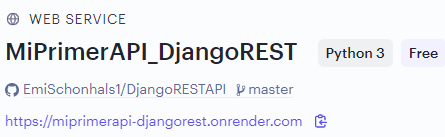
***De esta forma se hace un redespliegue, por ende si tocamos el nombre del web Service que creamos y vamos a Deploy veremos todo el despliegue que se está haciendo***

**Importante: en caso de que el despliegue nos tire este error:**

Para arreglarlo vamos a requirement.txt y modificamos Django para dejarlo en 4.0

Luego de eso subimos el nuevo commit como ya hemos visto anteriormente

***Si este error no nos aparece no tocamos nada de requirements.txt***

**Habiendo terminado todo, tendremos la URL de nuestro app web para que cualquiera pueda ingresar**

Si no nos aparece la página al ingresar a la URL solo basta con actualizar un par de veces, a veces demora un tiempo al ser la version free

**DE ESTA FORMA TENEMOS NUESTRA API TERMINADA JUNTO CON NUESTRA WEB LANZADA EN UN SERVIDOR LISTA PARA INICIAR UN PROYECTO FRONTEND**