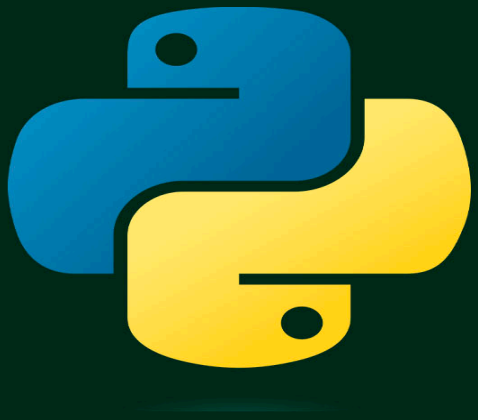
Volumen 1

Introducción a Django 4

Aprende a Crear Aplicaciones Web con Python



&

Marco Mendoza

Introducción a Django 4

Aprende a Crear Aplicaciones Web con Python

Marco Mendoza

Copyright

Primera publicación: Diciembre de 2022

Acerca del autor

Saludos, mi nombre es **Marco Mendoza**, soy ingeniero en sistemas, técnico en informática y autodidacta, a lo largo de mi vida he tenido la oportunidad de trabajar en diferentes campos como la administración de sistemas, diseño y programación web, desarrollo de aplicaciones móviles y también como maestro. hoy en día con el conocimiento que gané por mis años de trabajo, más muchas horas de estudio autodidacta, he llegado a desempeñarme en el área de seguridad informática realizando auditorías de seguridad en diferentes empresas e instituciones de gobierno.

También soy el creador de una pequeña comunidad en **YouTube** llamada **Hacking y Más** donde imparto algunos cursos de seguridad y hacking ético, asimismo también imparto cursos en otras plataformas de enseñanza online como **Udemy** donde cuento con más de **80,000 estudiantes**, mayormente en **Udemy** imparto cursos de programación y hacking ético.

Mi perfil de instructor: https://www.udemy.com/user/mendoza-limon-marco-antonio/

Dedicatoria

Este libro está dedicado a mi familia que siempre está dispuesta a apoyarme en cualquier proyecto que realice, también me gustaría hacer una mención especial en este libro a mi padre que falleció hace poco, gracias papa por tus enseñanzas.

Contenido

Contenido

Contenido

Introducción

Este libro trata sobre Django, un framework de desarrollo web que le ahorrará mucho tiempo y hace que el desarrollo web sea muy placentero. Con Django, puede crear y mantener aplicaciones web de alta calidad con un mínimo esfuerzo.

En el mejor de los casos, el desarrollo web es un acto emocionante y creativo; en el peor de los casos, puede ser una molestia repetitiva y frustrante. Django le permite concentrarse en las cosas divertidas, mientras alivia el dolor de los bits repetitivos. Al hacerlo, proporciona abstracciones de alto nivel de patrones comunes de desarrollo web, atajos para tareas de programación frecuentes y convenciones claras sobre cómo resolver problemas. Al mismo tiempo, Django trata de mantenerse fuera de su camino, lo que le permite trabajar fuera del alcance del framework según sea necesario.

¿Qué es Django?

Django es el framework de Python más popular que existe. Es de código abierto y de uso gratuito. También sigue la arquitectura Model-View-Controller (MVC), que ahora es la arquitectura de facto utilizada en el desarrollo de aplicaciones web. Django brilla en su arquitectura no modular. Puede ayudar al fácil desarrollo de sitios web basados en bases de datos que son de naturaleza compleja. Además, está listo para la reutilización, y un entorno reutilizable puede permitir a los desarrolladores realizar un desarrollo rápido. También sigue la filosofía DRY, lo que permite escribir un código mínimo, ahorrando tiempo. Al igual que cualquier otro marco web, ofrece una operación CRUD básica y un panel de administración fácil de usar para una administración fácil.

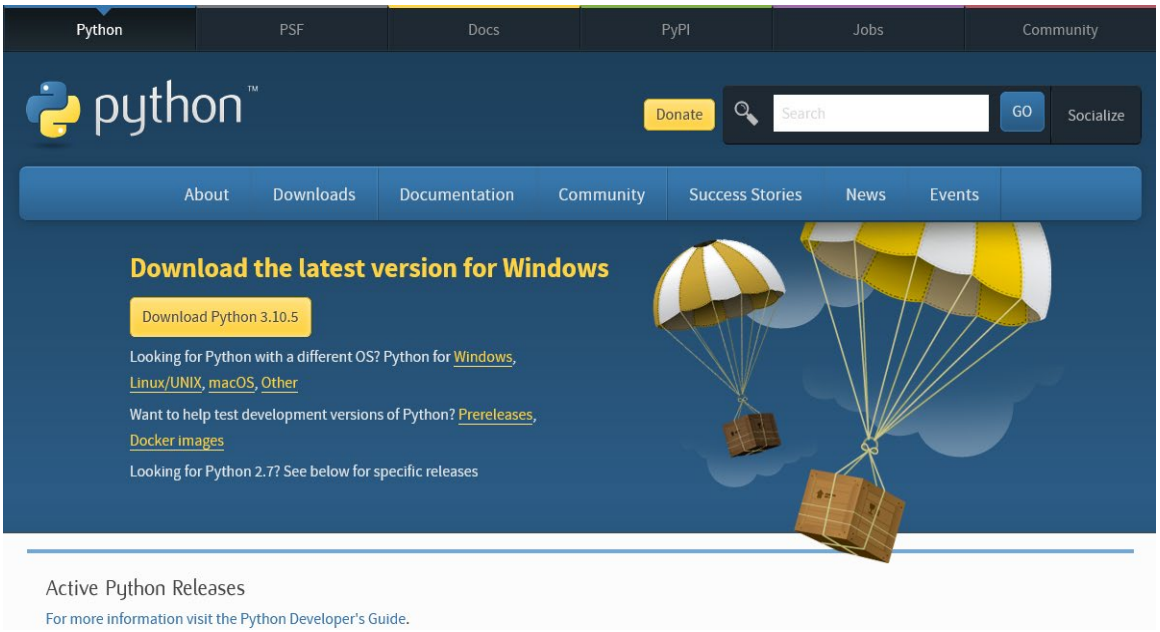
Toda la filosofía detrás de Django es el desarrollo rápido y hacer que los proyectos complejos funcionen para el desarrollador.

Configuración del Entorno de Trabajo

Para poder trabajar con este libro y sus ejercicios, necesitaremos instalar algunas herramientas antes de pasar a escribir código. Necesitaremos instalar **Python**, **Django** y **Visual Studio Code**. Se recomienda descargar la última versión de Python y Django, en mi caso para escribir este libro usé la versión de **Python 3.10.5** y la versión de **Django 4.1**, también se recomienda usar la versión más resiente de **Visual Studio Code** o cualquier editor de código de tu preferencia. Algo a tener en cuenta es que soló mostraré la instalación de estas herramientas ya mencionadas en un entorno **Windows** para no ser tan redundante, en el caso de **Linux** y **Mac** es muy parecida la instalación, solo que la mayoría de pasos se ejecutan desde la terminal.

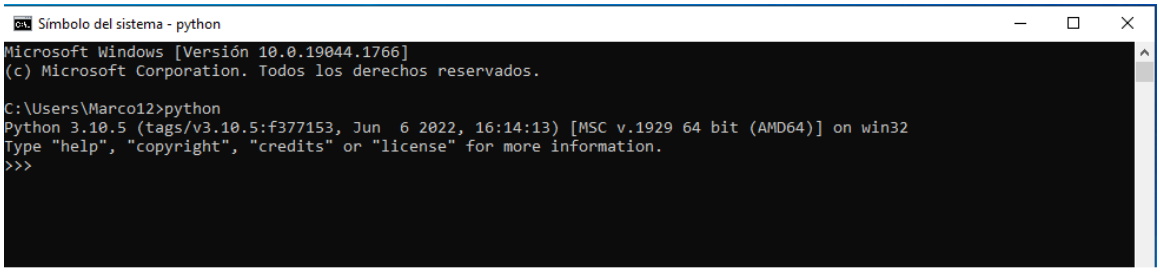
Instalación de Python

Lo primero que tenemos que hacer, es ir a **www.python.org** y descargar la versión más reciente de Python, en mi caso es la versión 3.10.5 para 64 bits.



Una vez descargado el instalador de Python basta con hacer doble clic en el archivo para iniciar la instalación, no te olvides de marcar la opción "**Add Python 3.10 to PATH**", es importante ya que de esta forma se agregará Python al **PATH** de Windows y podremos ejecutar código de una manera más práctica.

Hacemos clic en "**Install Now**" para iniciar la instalación básica de Python, y una vez terminada la instalación, abrimos un **CMD** y ejecutamos el comando *python*, si la instalación fue correcta nos mostrará una conexión directa a la **SHELL** de Python, como en la siguiente imagen.



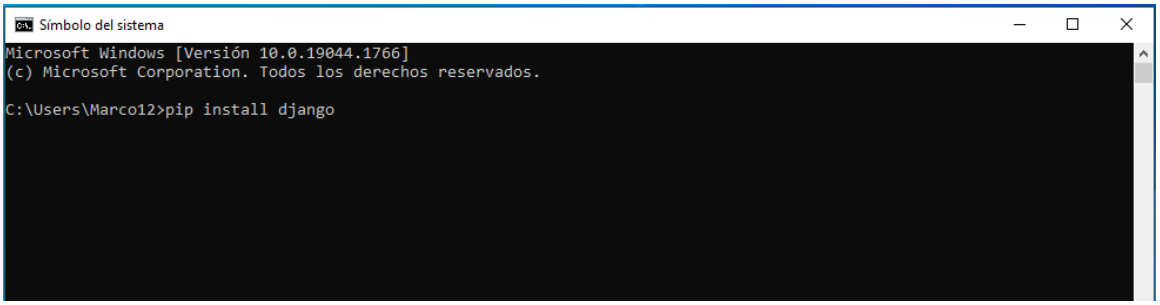
Instalación de Django

Después de instalar Python, podemos instalar fácilmente Django usando PIP. PIP significa "**Preferred Installer Program**" o "**Pip Installs Packages**", PIP es un administrador de paquetes para Python, y se usa principalmente para instalar, actualizar y desinstalar paquetes. Y viene por defecto con la versión 3.4 o versiones posteriores de Python.

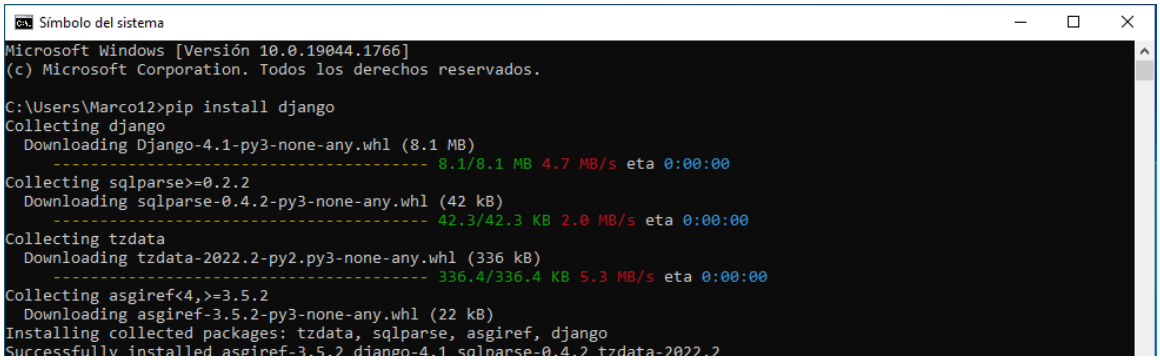
Tengo que hacer una pequeña aclaración con respecto a cómo vamos a trabajar con los proyectos de Django, en este caso vamos a instalar Django de forma global, no vamos a usar **entornos virtuales** o **virtual environments**, trabajaremos con solo una versión de Django en todos los proyectos por eso no veo conveniente usar **entornos virtuales**, también creo que es mejor enfocarse en Django por el momento, ya que este libro está dirigido a principiantes, obviamente sé que usar **entornos virtuales** es una mejor opción pero no quiero sobrecargarlos con muchos temas si son principiantes, y prefiero enfocarme en la estructura del Framework para darles una mejor idea de cómo usarlo.

Para las personas muy nuevas en Django voy a dar una pequeña explicación de lo que son los **entornos virtuales**, un **entorno virtual** es una forma de trabajar con diferentes versiones de paquetes para un sólo proyecto en específico, esto nos permite tener un mejor control de versiones dependiendo el proyecto que estemos desarrollando y también nos simplifica la forma de trabajar con otros desarrolladores, ya que hay menos posibilidades de errores al desplegar el proyecto en otro entorno. El trabajar con entornos virtuales es muy parecido a usar contenedores como en **Docker**.

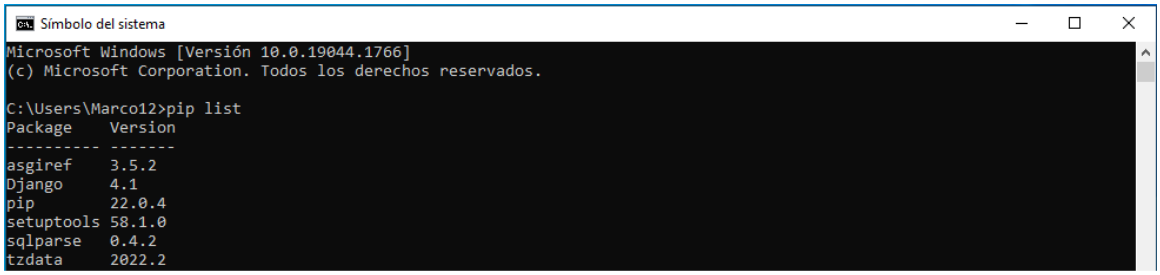
Ahora sí, procedamos a instalar Django con el siguiente comando *pip install django*.



Al usar el comando anterior instalaremos la versión más reciente de Django disponible, la instalación solo tomará unos minutos, y si todo salió bien y no tuvimos ningún error tendremos el siguiente resultado.



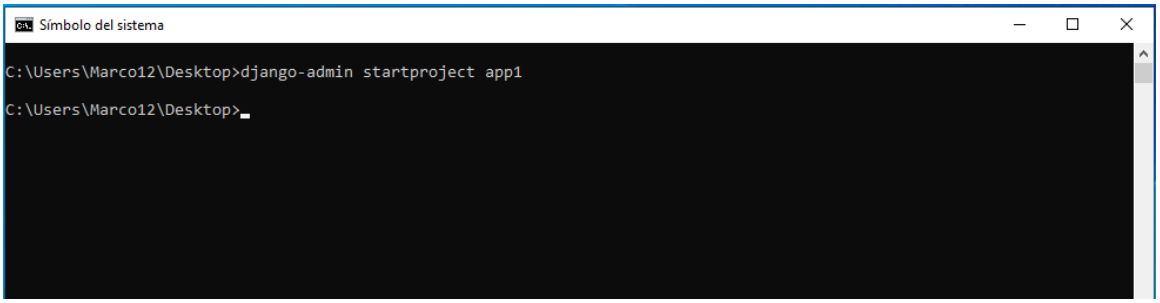
En este caso se instaló la versión **4.1** de Django, utilizaremos esa versión para trabajar en todos los ejercicios que realizaremos, con el comando *pip list* podemos listar todos los paquetes que tenemos instalados en nuestro equipo.



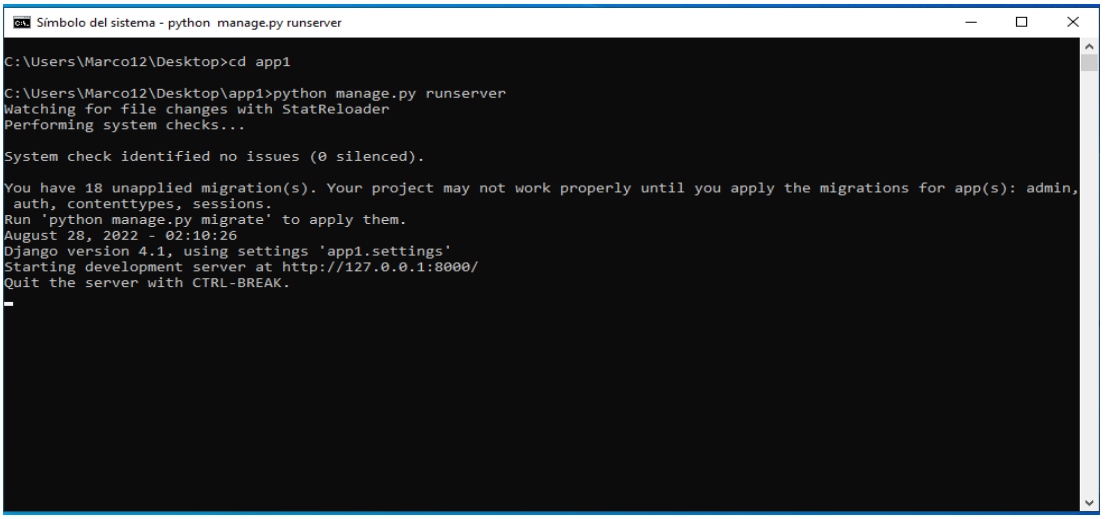
Con esto tendríamos todo listo para crear nuestro primer proyecto en Django, así que procederemos a realizar un pequeño proyecto como prueba. Esto también servirá para comprobar que no tenemos ningún error de ejecución.

Introducción

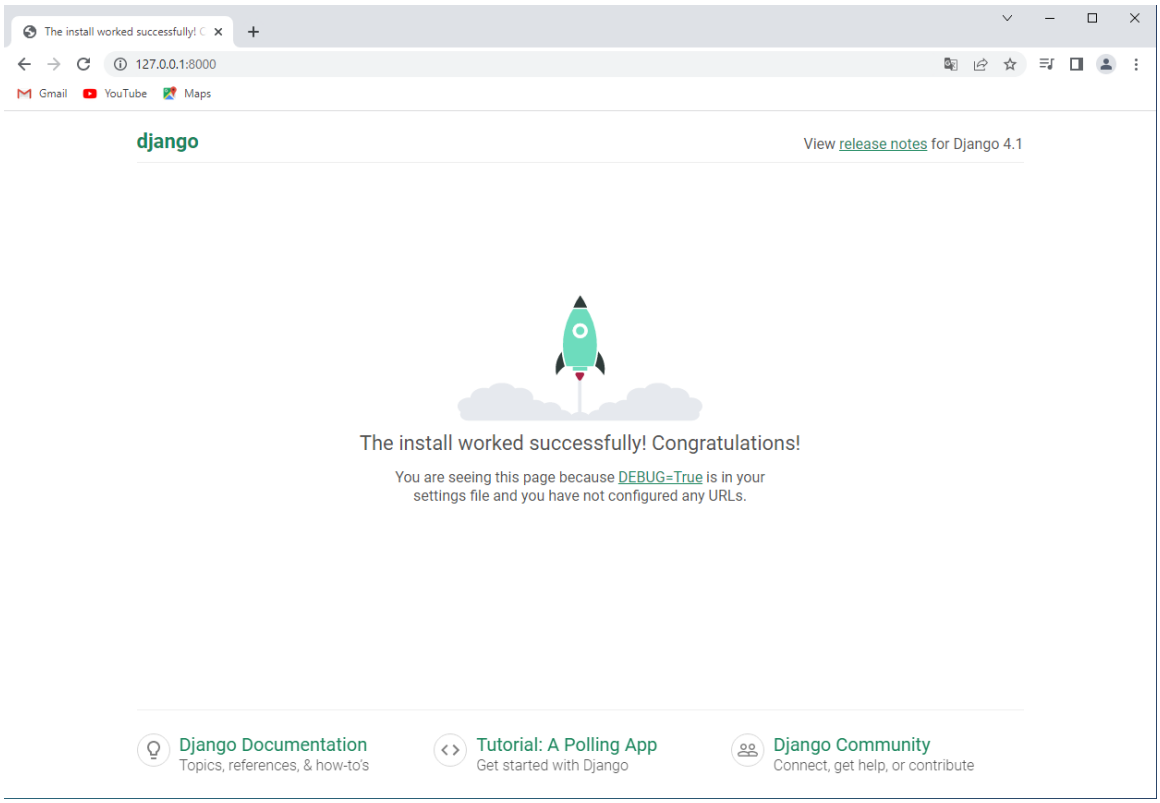
Para crear nuestro primer proyecto en Django tenemos que ejecutar el siguiente comando, *django-admin startproject app1*. En este ejemplo mi proyecto se llama **app1,** ustedes pueden usar el nombre que quieran, también recuerden revisar en que directorio están posicionados en la terminal de windows, en mi caso usaré el escritorio para guardar el proyecto.



Ahora que hemos creado nuestro primer proyecto, vamos a correr el servidor para probarlo, para ello tenemos que entrar a la carpeta del proyecto usando el comando *cd app1,* luego usaremos esta instrucción *python manage.py runserver* para ejecutar el servidor y probar nuestro proyecto.



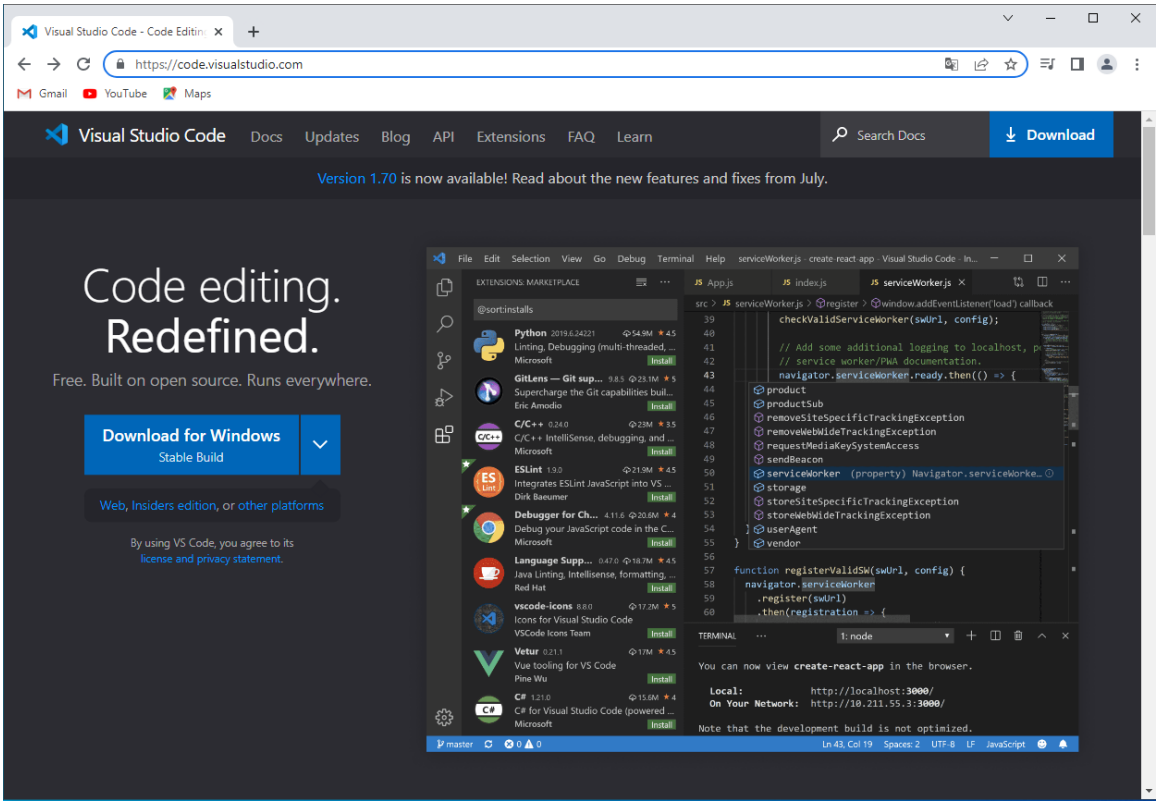
En la ventana de la terminal podrás observar una **URL** como esta **127.0.0.1:8000**, si pones esta **URL** en el navegador podrás observar tu proyecto en ejecución.



Así terminaríamos con éxito la instalación de Django, ya estamos listos para trabajar. Por último, me gustaría decirte como detener el servidor, para parar el servidor solo tienes que posicionarte en la terminal de windows y usar **CTRL + C,** eso detendrá el servidor.

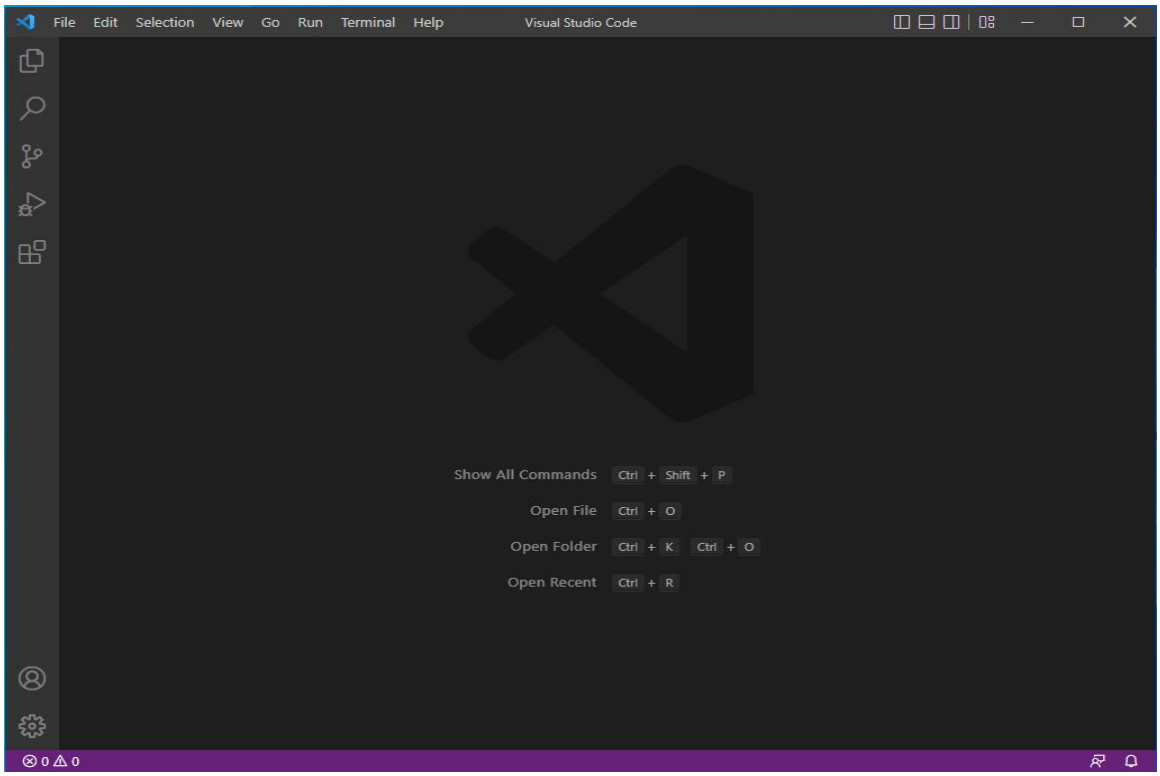
Instalación de Visual Studio Code

Ahora instalaremos el editor de código que usaremos para trabajar en este libro, en este caso es **Visual Studio Code,** para descargarlo solo tenemos que dirigirnos a **code.visualstudio.com**, y bajar el instalador de la versión más reciente de **Visual Studio Code.**



En este caso la versión más reciente disponible es la **1.70,** y esa es la que instalaremos, solo tenemos que hacer clic en el botón de **Donwload for Windows,** y comenzará la descarga del ejecutable **.exe**. Una vez descargado solo tenemos que hacer doble clic en el ejecutable e instalarlo como cualquier otro programa.

Cuando termine la instalación procederemos a abrir el editor y comprobar que todo funcione bien. Algo para tener en cuenta es que si abrimos cualquier archivo con extensión .**py**, Visual Studio Code nos sugerirá instalar la extensión para Python, es conveniente instalar la extensión para poder ejecutar código Python sin salir del editor de código, así que es conveniente tenerla instalada, o también puedes ir a la sección de extensiones de Visual Studio Code e instalarla de forma manual, solo tienes que buscarla como *Python* y proceder a instalarla.



Conclusión

En este primer capítulo dimos una pequeña introducción a lo que es Python y Django, también configuramos el entorno de trabajo que usaremos a lo largo del libro. Fue un capítulo sencillo, pero no menos importante, siempre es bueno cubrir todo por más sencillo que sea. En los siguientes capítulos ya entraremos más en materia con respecto a Django.

Estructura de Proyectos en Django

Django no tiene ninguna forma definitiva que debas seguir para estructurar tus aplicaciones. Hay muchas discusiones sobre las mejores prácticas para estructurar proyectos de Django, pero Django te proporciona una estructura por defecto cuando creas un proyecto. Se ocupa de muchas cosas por usted para que no tenga que hacerlas desde cero.

Es bueno que Django haga muchas cosas por ti. Sin embargo, comprender la estructura de un proyecto de Django puede ser un poco difícil para los principiantes. Cuando creas un proyecto de Django por primera vez, ya hay muchos archivos creados para ti. Y puede ser un poco abrumador saber por dónde empezar.

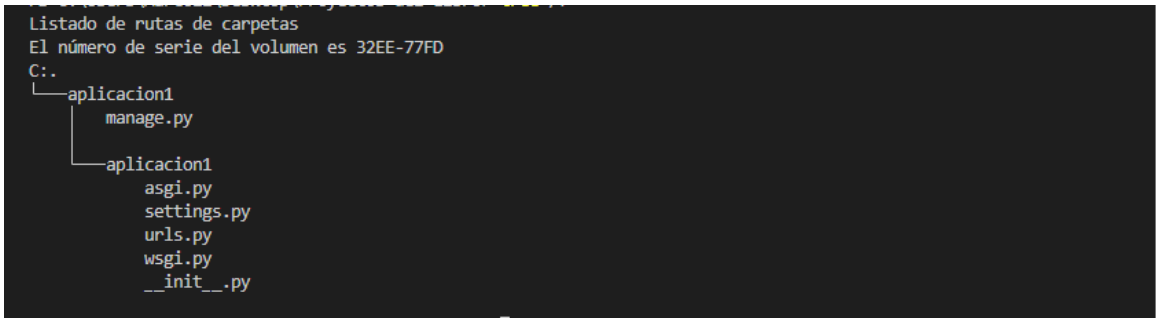
Pero una vez que se toma el tiempo de comprender cómo Django estructura los directorios y archivos de trabajo, lentamente comienza a apreciar el poder de Django.

Creando un Proyecto

Un proyecto es la unidad fundamental de tu aplicación web en Django. Para crear un proyecto en Django utiliza el siguiente comando:

django-admin startproject [nombredelproyecto]

El nombre del proyecto puede ser el que quieras. Pero debe asegurarse de no nombrar un proyecto de Django de la misma manera que un módulo o biblioteca de Python. En mi caso creé un proyecto llamado *aplicacion1* como ejemplo. Ejecuté el comando *django-admin startproject aplicacion1* y se creó un directorio llamado *aplicacion1*. Dentro del directorio *aplicacion1*, hay otro directorio con el mismo nombre *aplicacion1* y un archivo llamado *mange.py*. Aquí está la estructura inicial del proyecto:



El primer directorio *aplicacion1* es el directorio raíz del proyecto que usamos como ejemplo. Y el segundo directorio *aplicacion1* es el directorio del proyecto real. Hablemos de los archivos que tenemos hasta ahora.

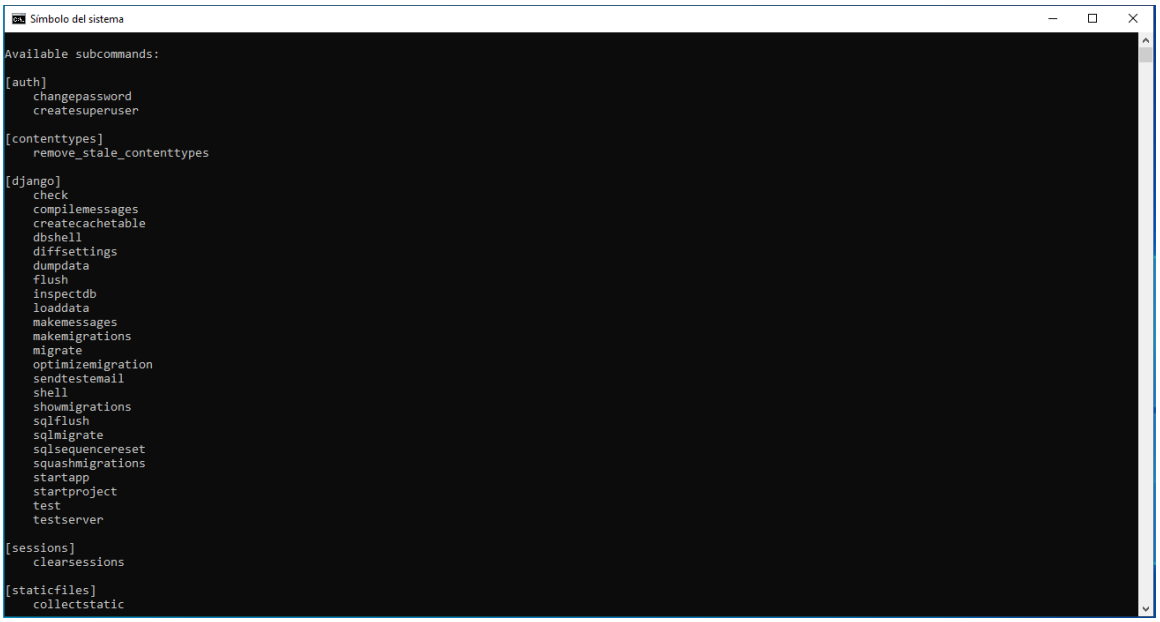
Manage.py

El archivo *manage.py* proporciona una utilidad de línea de comandos para un proyecto de Django. Utilizará esta utilidad de línea de comandos para realizar varias operaciones relacionadas con la depuración, ejecución e implementación de una aplicación web de Django. Por ejemplo, para ejecutar una aplicación Django en el servidor de desarrollo, usará el siguiente comando:

python manage.py runserver

Puedes ejecutar *manage.py* sin argumentos para ver los subcomandos disponibles:

Python manage.py



A continuación, veamos los archivos en la segunda carpeta del proyecto, que lleva el mismo nombre de la carpeta principal del proyecto, *aplicacion1*. Esta es la carpeta donde se encuentran los archivos de configuración del proyecto. El nombre de la carpeta es el mismo que el del proyecto. Esto hace que la carpeta sea única y, por lo tanto, crea una forma estándar de almacenar archivos de forma estructurada.

La carpeta es un paquete de Python que se indica mediante el archivo *__init__.py*. El propósito del archivo *__init__.py* es decirle al entorno de Python que la carpeta actual es un paquete de Python.

__init__.py

Es un archivo Python vacío. El archivo **__init__.py** le dice al intérprete de Python que el directorio *aplicacion1* es un paquete de Python.

Settings.py

Este es un archivo realmente importante desde el punto de vista del proyecto. Este contiene ciertas configuraciones que se pueden aplicar al resto del proyecto (o a todas las aplicaciones).

En el archivo *settings.py*, podemos hacer algunas de las siguientes operaciones:

- Listar aplicaciones que pueden estar preinstaladas o definidas por el usuario.
- Configurar el Middleware.
- Configurar y conectar la base de datos.
- Configurar plantillas, archivos estáticos y multimedia.
- Hacer una configuración personalizada para zonas horarias, correos electrónicos, autenticación, CORS, etc.

Además de las opciones mencionadas anteriormente, también hay muchas configuraciones específicas del proyecto o configuraciones específicas de la aplicación que puede definir en *settings.py*.

Urls.py

Este es un archivo para administrar las rutas URL del proyecto. Discutiremos las URL y las vistas en otro momento. Este archivo básicamente tiene una lista de direcciones URL que deben combinarse con una vista o cualquier otra función. En la carpeta del proyecto, los patrones de URL en su mayoría vinculan una URL base al archivo de URL de una aplicación en particular. No se preocupe si no puede entender algunos de los términos, los entenderá más adelante.

Wsgi.py

WSGI o **Web Server Gateway Interface** es un archivo que se utiliza para configurar el proyecto para producción o implementación. Esto se encarga del servidor cuando lo implementamos en producción. Es un servidor web sincrónico, es decir, escucha solo una solicitud y responde a eso a la vez.

Asgi.py

ASGI o **Asynchronous Server Gateway Interface** también es similar al archivo **WSGI** pero sirve como un servidor web asíncrono. Este archivo maneja las solicitudes que pueden ser asincrónicas, es decir, el servidor web puede responder a múltiples solicitudes y responderlas a la vez. Incluso podemos enviar tareas a un segundo plano utilizando este tipo de configuración del servidor.

Creando Nuestra Primera Aplicación

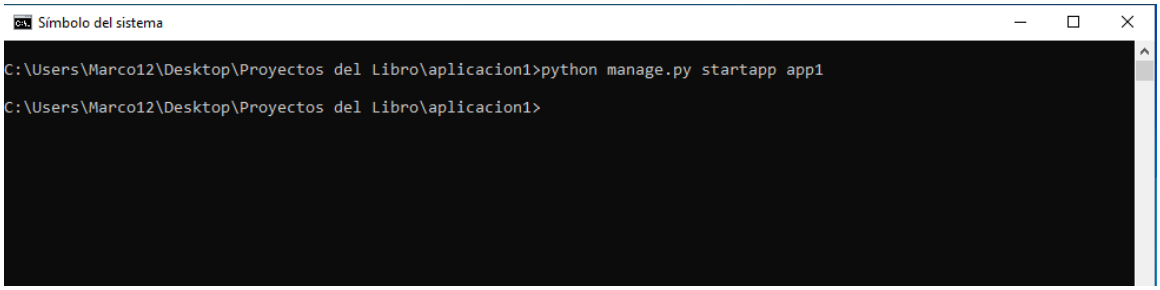
Entonces, creemos una aplicación para ver la estructura de una aplicación básica en Django. Para crear una aplicación, podemos usar la opción *startapp* con el comando *python manage.py* seguido del nombre de la aplicación como:

python manage.py startapp [Nombre_app]

En mi caso usare el nombre *app1* como ejemplo:

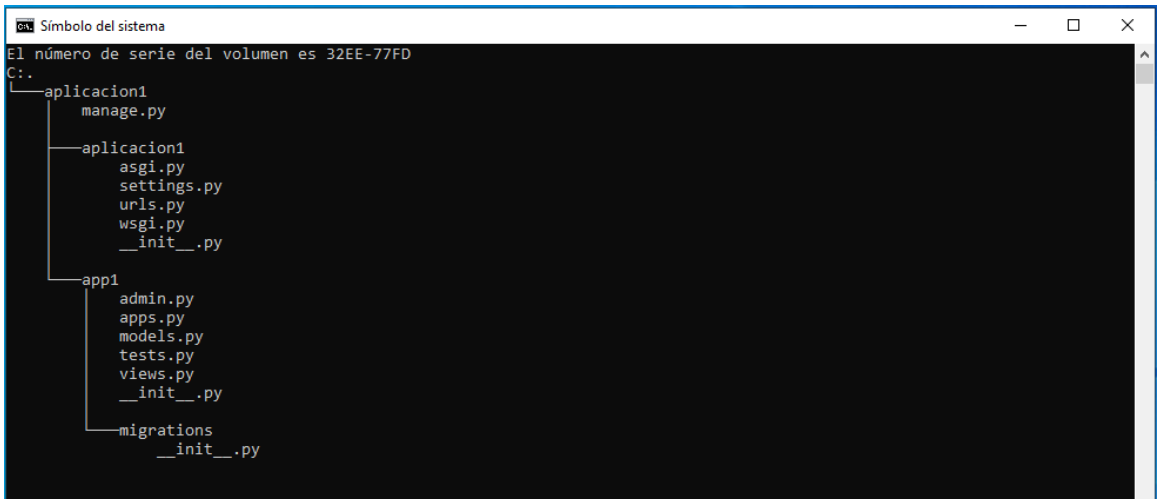
python manage.py startapp app1

El resultado sería en siguiente:



Estructura de la Aplicación

Después de crear una aplicación, debería tener una estructura similar a:



Como podemos ver, hay un par de cosas que discutir aquí. Los componentes principales en los que vamos a trabajar dentro del desarrollo de la aplicación en el proyecto son: *models.py*, *views.py*, *tests.py, etc*. Hay otros archivos que crearemos manualmente como *urls.py*, *serializers.py*, *etc*.

También debe agregar el nombre de la aplicación entre comillas en la lista *INSTALLED_APPS* en el archivo *settings.py*. Algo como esto:

```
# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'app1',
]
```

Models.py

Como lo sugiere, necesitamos definir el modelo de una base de datos aquí. La estructura real y la relación se crean con la ayuda de Python y Django en este archivo. Este es el lugar donde se podría definir la corteza de la aplicación web.

Hay varios aspectos en la creación de un modelo como campos, relaciones, metadatos, métodos, etc. Estos se definen con la ayuda de Python junto con los modelos de Django. En la mayoría de los casos, un modelo es como una sola tabla en una base de datos real.

El archivo es bastante importante e interesante ya que abstrae el trabajo manual de escribir consultas **SQL** para crear la base de datos.

Migrations

Esta carpeta de migraciones es una forma de que Django realice un seguimiento de los cambios en la base de datos. En cada migración o consulta real que se ejecuta para crear la tabla o la estructura de la base de datos, puede haber múltiples pasos o iteraciones de la base de datos, esta carpeta almacena esa información.

Para hacer una analogía, es como una carpeta **.git** pero para realizar un seguimiento de las migraciones o cambios en la base de datos.

Admin.py

Este es el archivo para realizar las operaciones a nivel de administrador. Generalmente usamos este archivo para registrar los modelos en la sección de administración sin tocar ninguna parte de la interfaz. Proporciona una función CRUD (Crear, Leer, Actualizar y Eliminar) incorporada al modelo. Esto es realmente bueno para probar el modelo manualmente antes de esforzarse en la parte del frontend.

Aparte de esto, podemos personalizar la sección de administración con este archivo.

Views.py

Este es un archivo que actúa como una lógica de controlador/servidor para el framework Django. Podemos definir funciones y clases como respuesta a las solicitudes entrantes del servidor a través del archivo *urls.py*. Hay un par de enfoques cuando se trata de escribir el formato de las funciones, como vistas basadas en clases, vistas basadas en funciones y otras, según el tipo de operación que se haya realizado.

Como se dijo anteriormente, es la **V** (**Vista**) en la arquitectura **MVT** de Django. Este es el lugar donde escribimos la lógica desde el lado del servidor para, digamos, representar páginas HTML (**Plantillas**), consultar la base de datos con operaciones CRUD, devolver una respuesta HTTP, etc.

Urls.py

Este es el archivo en el que se asigna una lista de patrones de URL a una función de una vista en particular. Este *urls.py* es específico de la aplicación y puede tener el prefijo de la ruta URL mencionada en el archivo *urls.py* de la carpeta del proyecto.

Entonces, sin profundizar mucho más, simplemente es un mapa de una ruta URL en particular con una función asociada que se activa (**llama**) cuando el usuario visita la URL. Entonces, creará un *urls.py* para cada aplicación en su proyecto de Django.

Tests.py

Este es un archivo donde podemos realizar pruebas automatizadas en la aplicación. Esto podría estar en integración con modelos, otras aplicaciones, configuraciones de proyectos, etc. Este es un componente que Django hace para tener pruebas unitarias fáciles y rápidas junto con los módulos de Python para pruebas avanzadas. Es bastante fácil integrar módulos y bibliotecas de Python en casi cualquier cosa en un proyecto de Django.

Apps.py

Este es el archivo para la configuración a nivel de aplicación. Podemos cambiar los campos predeterminados, el nombre de la aplicación, la configuración de correo electrónico, otras configuraciones específicas del módulo que se pueden usar en los modelos, las vistas, etc.

Sin embargo, la configuración predeterminada es suficiente para la mayoría de los casos. Por lo tanto, agregar la configuración de la aplicación es un caso raro.

__init__.py

El archivo **__init__.py** de una aplicación no es diferente del archivo **__init__.py** de un proyecto. Este archivo de Python vacío le dice al intérprete que app1 es un paquete de Python.

Otras Carpetas/Archivos

Además de la carpeta de la aplicación y la carpeta del proyecto, es posible que tengamos otras carpetas como *templates*, *static*, *media*, *etc*.

Templates

Hay un par de formas estándar en las que puede configurar su carpeta **Templates**. Ya sea en el proyecto raíz o dentro de aplicaciones individuales. La elección es tuya, usa la forma en la que te sientas más cómodo. Yo personalmente uso solo una carpeta de plantillas en el directorio raíz, pero puede guardarla donde quiera, pero estas dos son los estándares para facilitar la lectura y el mantenimiento de los proyectos.

Static

La carpeta **Static** es la carpeta en la que almacena su *css*, *javascript* e *imágenes* (imágenes o archivos multimedia que se utilizan en las plantillas). Esta es una buena manera de mejorar el rendimiento ya que en la producción, el servidor web recopila todos los archivos estáticos y los almacena en un solo lugar para responder a las solicitudes. La carpeta **Templates**, si está presente en la carpeta raíz, tiene subcarpetas con los nombres de las aplicaciones, y dentro de esas subcarpetas con los nombres de las aplicaciones, colocamos todos los archivos *.html* u otros archivos de plantilla.

Al igual que la carpeta de **Templates**, la ubicación se puede modificar o establecer como una configuración desde el archivo *settings.py*. Por lo general, los archivos estáticos (.css, .js, etc.) se almacenan en la carpeta raíz con los nombres de las aplicaciones como subcarpetas.

Media

La carpeta **Media** es donde puede almacenar los medios específicos para el usuario o los datos procesados por la aplicación. Por ejemplo, podemos almacenar las imágenes de perfil de los usuarios, archivos adjuntos de correo electrónico si se trata de una aplicación de correo electrónico, miniaturas de las publicaciones para una plataforma de blogs, etc.

La configuración de la carpeta **Media** es bastante similar a la carpeta **Static** pero tiene ciertas configuraciones adicionales.

Forms.py

Si su sitio web espera recibir entradas de los usuarios, debe usar formularios. Para trabajar con formularios en una aplicación, debe crear el archivo *Forms.py* en esa aplicación. Aquí escribirás los códigos para manejar formularios.

Conclusión

A partir de este capítulo, pudimos comprender la estructura de carpetas del framework Django. Exploramos los diversos archivos y carpetas con sus casos de uso y su propósito. Entonces, al leer la descripción anterior de los archivos y carpetas, es posible que tenga una idea aproximada sobre el flujo del ciclo de desarrollo en Django.

Introducción a Vistas

Las vistas en Django juegan un papel importante en su aplicación web porque siempre que escriba una URL en su navegador para acceder a su aplicación, Django buscará en *urlpatterns* la ruta coincidente y ejecutará la vista adjunta a esta URL en particular, luego esta vista se encargará de hacer consultas y renderizar una plantilla, así que si eso te emociona para aprender más sobre las vistas en Django, sigamos adelante

Después de familiarizarnos con la estructura de carpetas del framework Django, crearemos nuestra primera vista en una aplicación. Los conceptos básicos para crear y mapear una vista con una URL se aclararán al final de este capítulo.

Tipos de Vistas

En primer lugar, debe saber que las vistas de Django se dividen en dos tipos de vistas, basadas en funciones y vistas basadas en clases y ambas hacen el mismo trabajo, por lo que depende de usted elegir, pero si es principiante, le recomiendo que comience con la función.

Vistas basadas en Funciones

Las vistas basadas en funciones son las vistas en Django que están definidas por funciones. Estas funciones de Python toman una solicitud web y devuelven una respuesta web. Las vistas basadas en funciones se basan la mayor parte del tiempo en 'Lo que ves es lo que obtienes'. Esto se debe a que se debe incluir toda la lógica de la vista.

La mayor desventaja de este enfoque es la longitud y la repetición de códigos. Escribir un mismo bloque de código una y otra vez nunca es una buena práctica.

Vistas Basadas en Clases

Las vistas basadas en clases son vistas de Django basadas en clases de Python. Como se menciona en la documentación oficial de Django, "las vistas basadas en clases no reemplazan las vistas basadas en funciones, pero tienen ciertas diferencias y ventajas en comparación con las vistas basadas en funciones". Las vistas basadas en clases son muy fáciles de usar en casos simples, pero extenderlos a medida que aumenta la complejidad del proyecto se vuelve cada vez más difícil.

Las vistas basadas en clases pueden ser fáciles para empezar, pero necesita comprender las clases de Python junto con las vistas de Django para dominarlas.

Creando una Vista

Para crear una vista o una función típica, necesitamos escribir una función en el archivo *views.py* dentro de la carpeta de la aplicación. El nombre de la función puede ser cualquier cosa, pero debe ser un nombre sensato en lo que respecta a su usabilidad. Tomemos un ejemplo básico de envío de una respuesta HTTP de "*Hola Mundo"*.

```
1 from django.shortcuts import render
2 from django.http import HttpResponse
3
4 def holamundo(request):
5     return HttpResponse('Hola Mundo')
6
7
8 # Create your views here.
```

Sí, simplemente estamos devolviendo una respuesta HTTP en este momento, pero la representación en una plantilla HTML es bastante similar y fácil de entender en Django. Entonces, esta es una vista o una parte de la lógica, pero falta una parte. ¿Dónde se debe utilizar esta función? Por supuesto, en una URL, es decir, una ruta a un servidor web.

Veremos cómo asignar las vistas a una URL en Django en la siguiente sección.

Asignar una URL

Primero debemos crear un archivo *urls.py* en la carpeta de la aplicación para crear un mapa de la URL que se mapeará con la vista. Después de crear el archivo en la misma carpeta de la aplicación, donde se encuentra *views.py*, importe la función de la vista al archivo.

```
1 from .views import holamundo
2 from django.urls import path
3
4 urlpatterns = [
5     path('', holamundo, name='holamundo'),
6 ]
```

La ruta puede ser cualquier cosa que desee, pero por simplicidad, la dejaremos en blanco (') por el momento. Ahora, tiene la ruta para que su vista funcione, pero no está vinculada al proyecto principal. Necesitamos vincular las URLs de la aplicación a las URLs del proyecto.

Para vincular las URLs de su aplicación a la carpeta principal del proyecto, solo necesita agregar una sola línea de código en el archivo *urls.py* de la carpeta del proyecto.

```
1 from django.contrib import admin
2 from django.urls import path, include
3
4 urlpatterns = [
5     path('admin/', admin.site.urls),
6     path('', include('app1.urls')),
7 ]
```

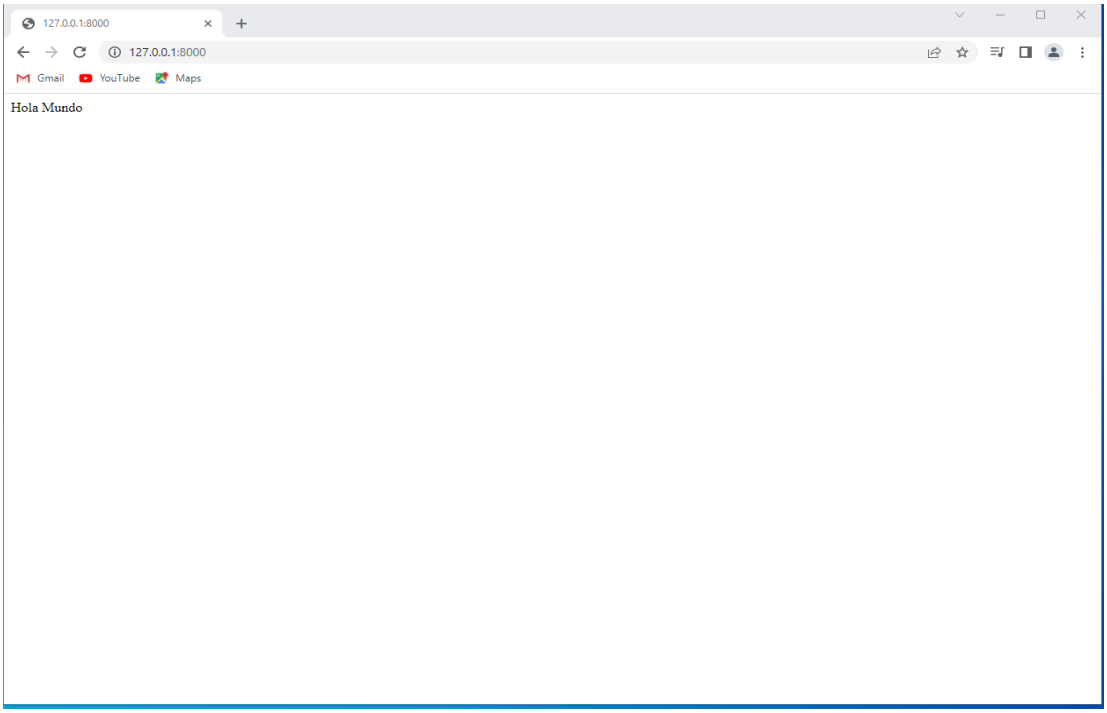
Debe agregar la línea *path('', include(' app1.urls'))*, y también importar la función *include* desde *django.urls*. Esta declaración adicional incluye las direcciones URLs o todas las *urlpatterns* de la aplicación *app1* desde el archivo *urls.py* hasta las rutas de direcciones URLs del proyecto.

Aquí, la ruta de la URL puede ser cualquier cosa como *'app/'*, *'holamundo/'*, *'vista/'*, etc. Pero como solo estamos entendiendo los conceptos básicos, la mantendremos como simples comillas ('') para hacer referencia a la URL raíz del proyecto.

También puede ver que hay otra ruta en nuestro proyecto *'admin/'* que es una ruta a la sección de administración. Exploraremos este camino y toda la sección de administración más adelante.

Ahora, si inicia el servidor y visita la URL predeterminada, es decir *http://127.0.0.1:8000*, verá un mensaje simple *Hola Mundo*.

Iniciamos el servidor con**: python manage.py runserver**



Función path()

La función *path()* en *urlpatterns* toma al menos 2 parámetros, es decir, el patrón de URL y la vista de cualquier otra función que pueda estar relacionada con el servidor web.

Ruta URL

La ruta URL es el patrón o, literalmente, la ruta que utiliza en la barra de búsqueda del navegador. Esto puede ser estático, es decir, algún texto codificado como *home/*, *user/*, *post/home/*, etc. Y también podemos tener URLs dinámicas como *post/id/*, *user/name/*, etc. Aquí los caracteres *id* y *name* serán reemplazados por la identificación real (integer/primary key) o el nombre (String) en sí.

Esto se usa en una aplicación web real, donde puede haber un perfil de usuario que necesita la identificación de usuario única para representar específicamente a ese usuario. El perfil de usuario es solo un ejemplo, puede ser cualquier cosa, como publicaciones, correos electrónicos, productos, cualquier forma de aplicación basada en contenido.

Vistas

La vista o la función es el nombre de la función que se adjuntará a esa ruta URL. Eso significa que una vez que el usuario visita esa URL, la función será llamada. Vista es solo una palabra elegante para una función (o cualquier lógica básicamente). Hay mucho que cubrir cuando se trata de vistas, ya que hay muchas formas de crearlas, cómo usarlas para varios casos de uso se puede aprender en el camino, ya que es un tema con bastante profundidad cuando hablamos de Django.

Aprenderemos a crear diferentes implementaciones y estructurar nuestras vistas, por el momento considéralas como la unidad donde se pueden realizar todas las operaciones en la web. Podemos crear otras funciones independientes en Python para trabajar con las vistas para que sea un poco estructurado y legible.

Nombre de la URL

Este es un parámetro opcional para la función *path()*, ya que no es obligatorio dar un nombre al mapa de URL. Esto puede ser realmente útil en sitios web con varias páginas donde necesita vincular una página a otra y eso se vuelve mucho más fácil con el nombre de la URL.

Ejemplos

Vamos a crear algunos ejemplos para comprender el funcionamiento de las vistas y las URLs. Crearemos una URL dinámica e integraremos módulos de Python en las vistas para familiarizarnos con el concepto.

URL Dinámica

Podemos usar las URLs dinámicas o las variables de marcador de posición para representar el contenido de forma dinámica. Vamos a crear otro conjunto de vista y URL, agregamos el siguiente código a nuestro archivo *views.py*.

```
def hola(request, name):
    return HttpResponse('Hola, ' + name)
```

Agregando esas dos últimas líneas de código, tendríamos el siguiente resultado en *views.py*:

```
from django.shortcuts import render
from django.http import HttpResponse

def holamundo(request):
    return HttpResponse('Hola Mundo')

def hola(request, name):
    return HttpResponse('Hola, ' + name)


# Create your views here.
```

Esta vista o función toma un argumento adicional llamado *name* y, en respuesta, simplemente dice *Hola, name* donde *name* puede ser cualquier cadena. Ahora, después de crear la vista, debemos asignar la vista a un patrón de URL. Agregaremos una ruta para esta función de saludo en el archivo *urls.py* de la aplicación.

```
path('hola/<str:name>/', hola, name='hola'),
```

Al añadir esta nueva línea de código a *urls.py*, el archivo quedaría así:

```
from .views import holamundo, hola
from django.urls import path

urlpatterns = [
    path('', holamundo, name='holamundo'),
    path('hola/<str:name>/', hola, name='hola'),
]
```

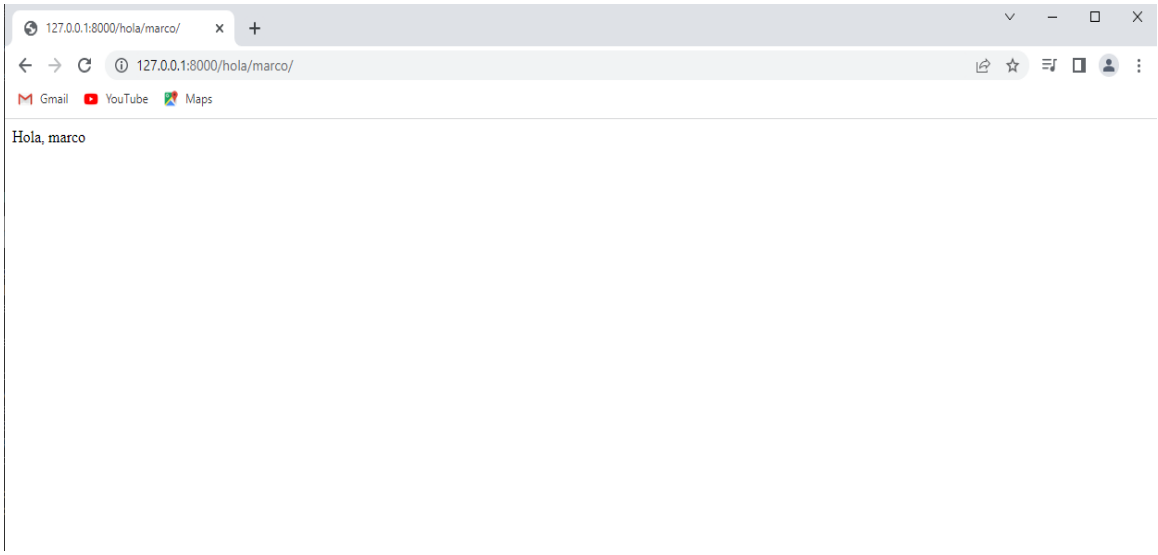
Puede ver cómo hemos creado el patrón de URL aquí. La parte *hola/* es estática, pero *<str:name>* es una variable o simplemente un parámetro de URL que se pasa a la vista como el valor de la variable *name*. También le hemos dado a la URL un nombre llamado *hola*, solo para demostrar su creación.

Un recordatorio para que no recibas un error al probar este código, Importa la función *hola* desde las vistas así:

```
1 from .views import holamundo, hola
```

Si regresas un poco atrás, podrás corroborar que ya había incluido esta función en el código, en la parte de asignar una URL a la nueva vista. Solo es un pequeño recordatorio para no tener errores.

Entonces, después de visitar la URL *https://127.0.0.1:8000/hola/marco*, debería ver una respuesta como esta *Hola, marco*, tan simple como eso.



Ahora, ¿cómo funciona esto? Primero vemos la vista. La función toma dos parámetros, uno es el más común, la solicitud que almacena los metadatos sobre la solicitud, el otro parámetro es el nombre que usaremos para responder dinámicamente al servidor. La variable *name* se usa en la cadena con la función *HttpResponse* para devolver una cadena simple.

Luego, en las URLs, necesitamos encontrar una manera de pasar el nombre de la variable a la vista, para eso usamos <*string:name*> que es como un parámetro de URL para la vista. La función *path* analiza automáticamente cual es la vista adecuada y, por lo tanto, llamamos a la función *hola* con la variable *name* desde la URL.

Usando Python

Usaremos algunas bibliotecas o funciones de Python en la aplicación de Django. De esta manera, veremos que es casi obvio usar funciones o bibliotecas de Python en el framework Django, ya que, de hecho, todos los archivos con los que estamos trabajando son archivos de Python.

Ahora agregue las siguientes líneas de código a *views.py*.

```
1 from random import randint
2
3 def numero(request):
4     numero = randint(1,15)
5     return HttpResponse(f"Su número es {numero}")
```

Al final, con todas las demás vistas que hemos hecho, tendríamos un archivo con todo este código:

```
1 from django.shortcuts import render
2 from django.http import HttpResponse
3 from random import randint
4
5 def holamundo(request):
```

```
    return HttpResponse('Hola Mundo')

def hola(request, name):
    return HttpResponse('Hola, ' + name)

def numero(request):
    numero = randint(1,15)
    return HttpResponse(f"Su número es {numero}")

# Create your views here.
```

Esta vista está usando el módulo aleatorio, puede usar prácticamente otros módulos o bibliotecas compatibles con la web. Hemos utilizado la función *random.randint* para generar un número aleatorio entre 1 y 15. Hemos utilizado una cadena de respuesta con estilo *(f"{variable}")* ya que *int* no es compatible con la concatenación de respuesta. Así que esta es la lógica de nuestro mapa, ahora necesitaremos vincularlo a una URL.

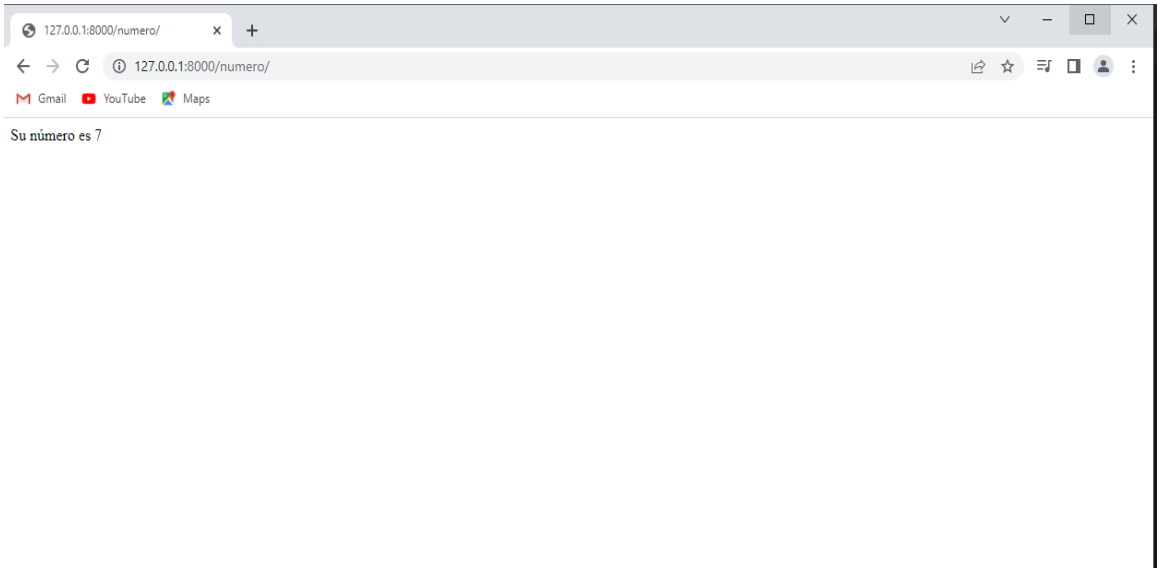
Esta seria la ruta que agregaríamos a *urls.py*, en la carpeta de la aplicación:

```
path('numero/', numero, name='numero'),
```

Al final el archivo *urls.py*, quedaría así:

```
from .views import holamundo, hola, numero
from django.urls import path

urlpatterns = [
    path('', holamundo, name='holamundo'),
    path('hola/<str:name>/', hola, name='hola'),
    path('numero/', numero, name='numero'),
]
```

Además, importe el nombre de la vista desde *from .views import* y agregue otras vistas si están presentes. Ahora si vamos a la URL *https://127.0.0.1:8000/numero/*, veremos un número aleatorio en la respuesta. Así es como usamos Python para hacer la lógica de nuestra vista.



Entonces, eso fue lo básico para crear y mapear vistas y URLs. Es el flujo de trabajo más fundamental en el desarrollo de proyectos de Django. Debe familiarizarse con el proceso de asignación de vistas y URLs antes de sumergirse en plantillas, modelos y otras cosas complejas.

Conclusión

A partir de esta parte del libro, abordamos los conceptos básicos de las vistas y las URLs. El concepto de asignación de URLs y vistas podría haberse aclarado mucho y será aún más apasionante después de que exploremos el manejo de plantillas y los archivos estáticos.

Plantillas

Después de aprender los conceptos básicos de las vistas y las URLs, ahora podemos pasar al siguiente concepto, es decir, Plantillas. En Django, las plantillas son un componente bastante importante para la aplicación, ya que actúan como el *frontend* de la aplicación web. Con la ayuda de plantillas y algunas funciones proporcionadas por Django, se vuelve muy intuitivo y simple crear contenido web dinámico. En esta parte, entendemos qué son las plantillas y cuál es la forma de representarlas.

¿Qué son las Plantillas?

Las plantillas son simplemente un documento *html* o una especie de estructura alámbrica para que el contenido se muestre en la aplicación web. Las plantillas nos permiten representar algunos datos más relevantes en lugar de simples respuestas *HTTP* de texto como hicimos anteriormente. Incluso podemos reutilizar ciertos componentes de una plantilla en otra utilizando el **lenguaje de plantillas de Django** (más sobre esto más adelante). Entonces, usando plantillas HTML, podemos escribir una página web completa.

Creación de Plantillas

Para crear una plantilla, podemos escribir un documento HTML simple como el siguiente:

Cree una carpeta de *template* en la carpeta base, dentro de la carpeta de *template*, cree un archivo *index.html*

En el archivo *index.html* agrega el siguiente código:

```
<!DOCTYPE html>
<html lang="es-MX">
<head>
    <meta charset="UTF-8">
    <title>Ejemplo de HTML</title>
</head>
<body>
    <h1>Hola, Mundo!</h1>
</body>
</html>
```

Esta es una plantilla *HTML* simple, con las etiquetas *<h1>*. Como Django es un framework, existe un estándar para almacenar todas las plantillas para el proyecto y la aplicación. Hay un par de opciones estándar:

Uno de los cuales es crear una carpeta de plantillas en la carpeta raíz como se discutió anteriormente, también necesitamos modificar el archivo *nombre_proyecto/settings.py*.

```
import os

TEMPLATES = [
    {
        'BACKEND':'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, 'template'),],
        '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',
            ],
        },
    },
]
```

En este fragmento, hemos cambiado la opción *DIRS* para buscar las plantillas en la carpeta *template* del directorio raíz, y también agregamos *import os*.

El otro estándar es crear una carpeta de plantillas en cada aplicación

Renderizar Plantillas

Después de crear una plantilla y realizar las configuraciones requeridas para asegurarnos de que Django pueda recoger esas plantillas, necesitamos trabajar con vistas y URL para representar realmente esas plantillas.

Hay un par de formas de renderizar plantillas en Django y algunas de ellas se analizan a continuación.

Usando TemplateView

TemplateView es una clase que viene con la biblioteca *django.views.generic*. Esta clase nos permite representar una plantilla al proporcionar el nombre de la plantilla, argumentos o variables para analizar, etc.

La forma más sencilla de renderizar una plantilla es de la siguiente manera, para ello modificaremos el archivo *urls.py* del proyecto:

```
1 from django.contrib import admin
2 from django.urls import path, include
3 from django.views.generic import TemplateView
4
5 urlpatterns = [
6     path('admin/', admin.site.urls),
7     path('', TemplateView.as_view(template_name="index.html"),
  name="index"),
8 ]
```

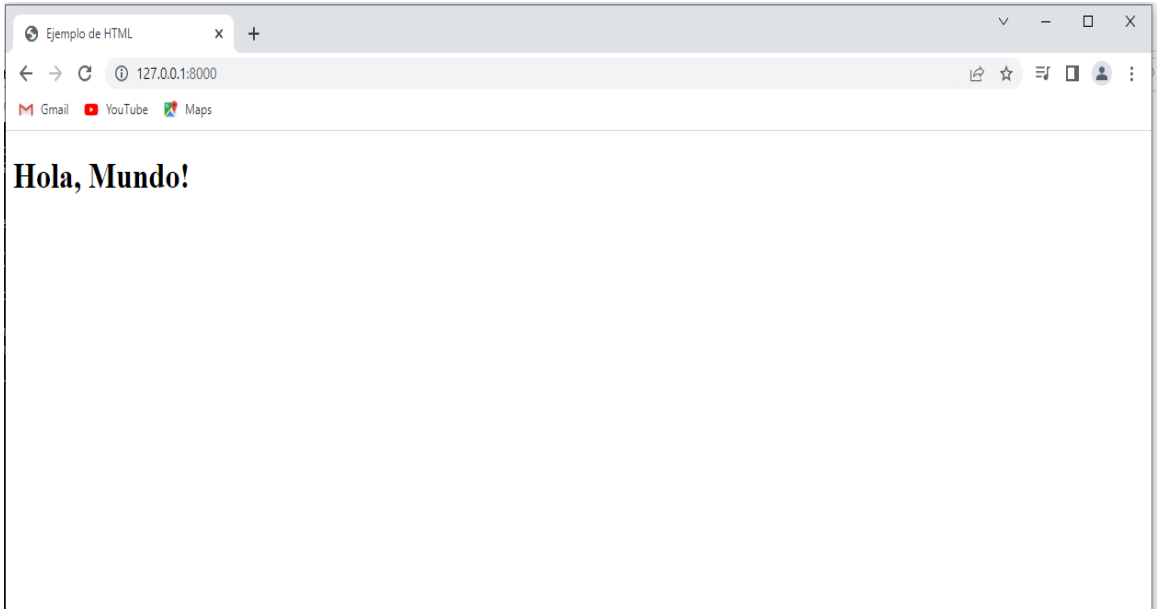
Necesitamos importar *TemplateView* desde *django.core.generic* para usar la clase para representar la plantilla.

La clase *TemplateView* toma un par de argumentos, usaremos *template_name* como un argumento que toma el nombre de la plantilla. Aquí, usamos *index.html* como la plantilla que creamos anteriormente. No necesitamos especificar la ruta completa a la plantilla ya que hacemos modificaciones en el archivo *settings.py* para seleccionar la plantilla del directorio mencionado. Usamos la función *as_view* para cargar la clase como una función/vista.

Si ejecutamos el servidor:

python manage.py runserver

Ahora podemos ver el siguiente resultado y, por lo tanto, ahora estamos representando una plantilla HTML simple en Django.



Usando Render

También podemos usar la función *render* de *django.shortcuts* para simplemente renderizar una plantilla. Pero crearemos una función de Python o una vista para representar la plantilla. Entonces, crearemos un mapa *View-URL* como lo creamos en el capítulo anterior.

En primer lugar, vamos a crear una función de vista en el archivo *app1/views.py*, más generalmente (*app_nombre/views.py*). En primer lugar, necesitamos importar la función de renderizado desde *django.shortcuts* y luego devolver la llamada de función de renderizado.

Abrimos el archivo *views.py* de la aplicación y agregamos el siguiente código:

```
1 from django.shortcuts import render
2
3 def inicio(request):
4     return render(request,'index.html')
5
6 # Create your views here.
```

Y en *urls.py*, crearemos un patrón diferente como, *'inicio/'*

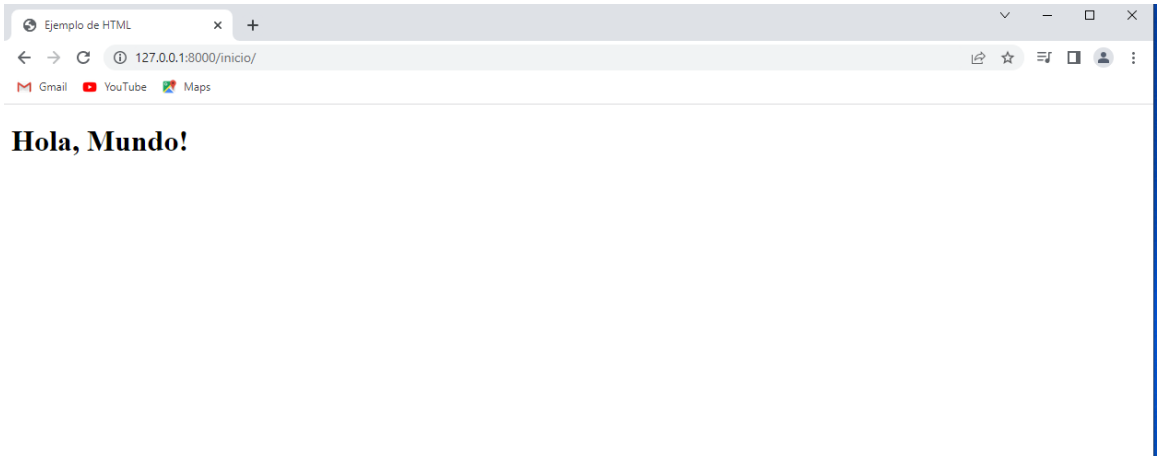
```
1 from .views import inicio
2 from django.urls import path
3
4 urlpatterns = [
5     path('inicio/', inicio, name="inicio")
6 ]
```

No olvides agregar las URLs de la aplicación al archivo *urls.py* del proyecto, si no lo haces tendrás un error cuando ejecutes el servidor.

Agrega las siguientes líneas de código al archivo *urls.py* del proyecto:

```
from django.contrib import admin
from django.urls import path, include
from django.views.generic import TemplateView


urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('app1.urls')),
    path('', TemplateView.as_view(template_name="index.html"),
  name="index"),
]
```

Entonces, después de crear el mapa View-URL y asegurarnos de que la URL de la aplicación esté cargada en las URLs del proyecto, podemos ver el resultado como una plantilla HTML simple.



Lenguaje de Plantillas de Django

El lenguaje de plantillas de Django es la forma en que Django hace que las plantillas sean más dinámicas y sea fácil escribir aplicaciones web dinámicas. Echaremos un breve vistazo a lo que podemos hacer con este tipo de lenguaje de plantillas en Django.

Variables

Este es el caso de uso más común para el lenguaje de plantillas de Django, ya que podemos usar las variables del *backend* e inyectarlas en la plantilla. Podemos analizar la variable en la plantilla por la sintaxis: *{{ variable_name }}*.

Para mostrar sus casos de uso, podemos declarar una variable en una vista y luego usarla en una plantilla. Aunque no es dinámico en este momento, más adelante podemos obtener valores de la base de datos y almacenarlos en forma de variables en nuestras vistas.

Añade una nueva plantilla a la carpeta *template*, en mi caso la llamare *variable.html*:

```
<!DOCTYPE html>
<html lang="es-MX">
<head>
    <meta charset="UTF-8">
    <title>Variable</title>
</head>
<body>
    <h1>Hola, {{ nombre }}</h1>
</body>
</html>
```

Añade la siguiente función a *views.py*:

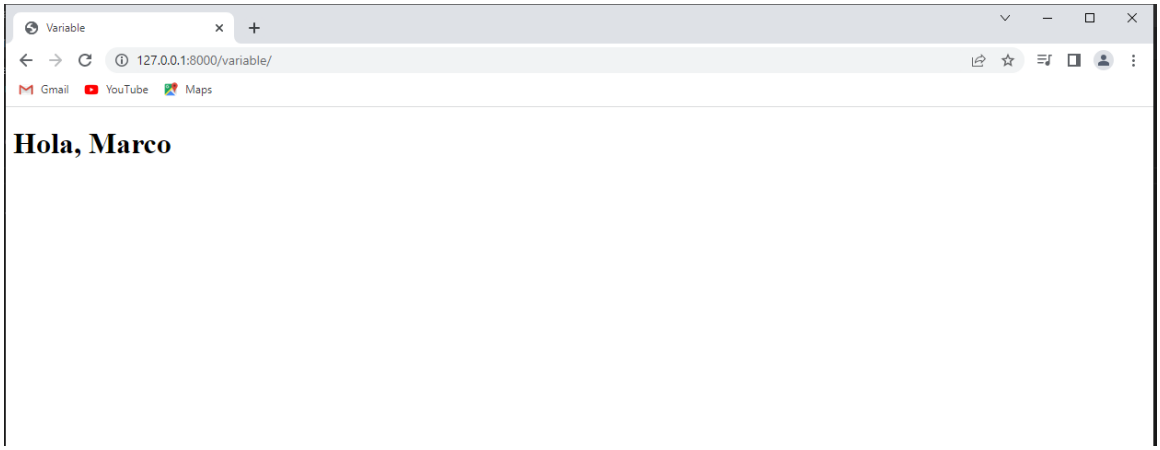
```
from django.shortcuts import render

def inicio(request):
    return render(request,'index.html')

def variable(request):
    nombre = "Marco"
    return render(request, 'variable.html', {'nombre':nombre})

# Create your views here.
```

Como podemos ver, la variable en las vistas se pasa como un diccionario en Python. La clave de referencia junto con un valor de la variable como el nombre de la variable. Usaremos la clave en las plantillas para usar el valor de la variable.

Solo falta agregar la URL a *urls.py*:

```
1 from .views import inicio, variable
2 from django.urls import path
3
4 urlpatterns = [
5     path('inicio/', inicio, name="inicio"),
6     path('variable/', variable, name="variable"),
7 ]
```

Ejecutamos **python manage.py runserver:**



Sentencia Condicional

Incluso podemos usar la declaración condicional en la Plantilla usando una sintaxis muy simple. Podemos usar *{% if condition %}* para usar ciertos tipos especiales de bloques en la plantilla.

Necesitamos terminar esos bloques también usando la sintaxis *{% endif %}*, aquí sí puede haber otros bloques que exploraremos más adelante.

Para crear una condición *if* básica en la plantilla, usaremos el siguiente ejemplo.

Agregamos la siguiente vista a *views.py*:

```
1 from django.shortcuts import render
2 from random import randint
3 def inicio(request):
4     return render(request,'index.html')
5 
6 def variable(request):
7     nombre = "Marco"
8     return render(request, 'variable.html', {'nombre':nombre})
9 
10 def condicion(request):
11    numero = randint(1,20)
12    return render(request, 'condicional.html', {'num':numero})
13 
14 # Create your views here.
```

Aquí, hemos usado el nombre de la clave como *num*, lo que indica que podemos dar diferentes nombres a la clave que debe usarse en la plantilla para representar los valores.

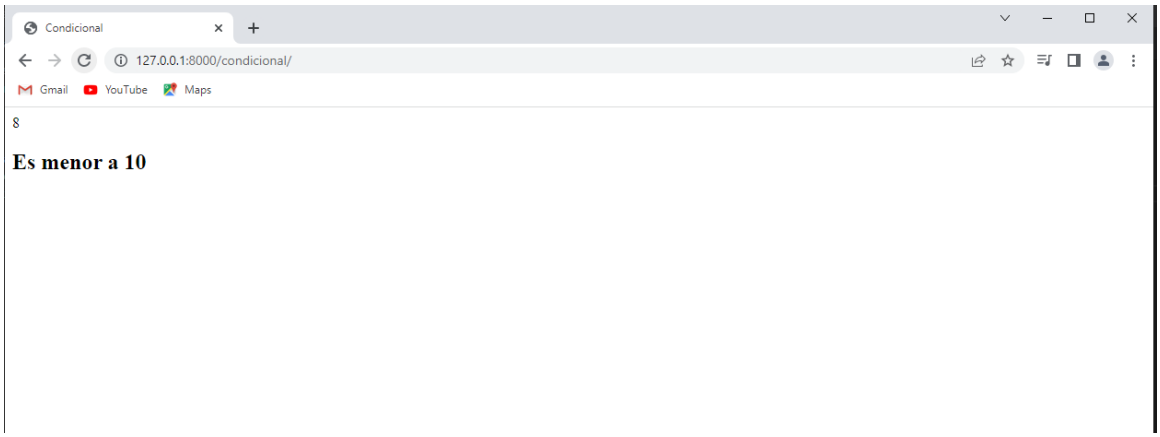
Ahora Agregamos la URL a *urls.py*:

```
1 from .views import inicio, variable, condicion
2 from django.urls import path
3 
4 urlpatterns = [
5     path('inicio/', inicio, name="inicio"),
6     path('variable/', variable, name="variable"),
7     path('condicional/', condicion, name="condicion"),
8 ]
```

Por último creamos la plantilla *condicional.html*:

```
<!DOCTYPE html>
<html lang="es-MX">
<head>
    <meta charset="UTF-8">
    <title>Condicional</title>
</head>
<body>
    {{  num }}
    {% if num > 10 %}
        <h2>Es mayor a 10</h2>
    {% elif num == 10 %}
        <h2>¡Es diez!</h2>
    {% else  %}
        <h2>Es menor a 10</h2>
    {% endif %}
</body>
</html>
```

Si ejecutamos esto con **python manage.py runserver:**



Entonces, como podemos ver, podemos usar las condiciones *if-else* en la plantilla y eso ya es poderoso. Esto puede ser un poco complicado en cuanto a manejar operaciones o condiciones matemáticas en una sola condición. Esto realmente se puede usar para conjuntos de datos realmente grandes que se pueden reducir a menos codificación y también mejorar la legibilidad.

Bucle For

Ahora, el componente más crucial del lenguaje de plantillas de Django son los bucles. De hecho, podemos iterar sobre objetos/listas en la plantilla. Esto se convierte en un gran concepto para hacer una aplicación web dinámica. Es posible que queramos iterar sobre todas las entradas en una base de datos, o cualquier otra forma de datos que pueda hacer que la aplicación sea mucho más dinámica y se sienta en tiempo real.

La sintaxis del ciclo *for* es casi similar a la condición *if-else*. Simplemente reemplazamos la condición con el iterador y la lista/objeto del contexto de la vista. *{% for i in list %}*, también termina el ciclo *for* como *{% endfor %}*.

Agregamos la siguiente función a *views.py*:

```
from django.shortcuts import render
from random import randint
def inicio(request):
    return render(request,'index.html')

def variable(request):
    nombre = "Marco"
    return render(request, 'variable.html', {'nombre':nombre})

def condicion(request):
   numero = randint(1,20)
   return render(request, 'condicional.html', {'num':numero})

def bucle(request):
    comida = ('hamburguesa', 'pizza', 'palomitas', 'fruta', 'jamon')
    return render(request, 'bucle.html', {'comi': comida})
```

Hemos creado una lista simple de Python llamada *comida*, y la mostramos en la plantilla usando un objeto de diccionario, *comi* como clave para almacenar el valor de la lista de comida.

Agregamos la URL a *urls.py*:

```
from .views import inicio, variable, condicion, bucle
from django.urls import path

urlpatterns = [
    path('inicio/', inicio, name="inicio"),
    path('variable/', variable, name="variable"),
    path('condicional/', condicion, name="condicion"),
    path('bucle/', bucle, name="bucle"),
]
```

Por último creamos la plantilla *bucle.html*:

```
<!DOCTYPE html>
<html lang="es-MX">
<head>
    <meta charset="UTF-8">
    <title>Bucle</title>
</head>
<body>
    <ul>
        {% for comida in comi %}
        <li>{{ comida }}</li>
        {% endfor %}
    </ul>
</body>
</html>
```

Si ejecutamos esto con **python manage.py runserver:**

Hemos usado un bucle *for* simple en la sintaxis, usamos un iterador en este caso, *comida* actúa como un iterador. Usamos esto para almacenar valores uno por uno de la lista *comi* que se pasó anteriormente en las vistas como una clave en el diccionario.

Por lo tanto, esto es bastante escalable y se usa para obtener las entradas en la base de datos y, por lo tanto, hace que sea mucho más fácil hacer que una aplicación web dinámica sea más rápida.

Herencia de Plantilla

Hasta ahora hemos visto que necesitamos crear la plantilla base una y otra vez como todos los elementos básicos de HTML, el título y toda la estructura básica. Pero, ¿y si podemos reutilizar una plantilla específica en otra y extender la funcionalidad de esa plantilla a una nueva? Esto evita la redundancia de escribir la plantilla básica completa o el diseño de una aplicación web una y otra vez.

Para hacer eso, Django tiene la herencia de plantilla. Podemos usar una plantilla como su diseño básico o un componente específico en la aplicación web. Nuevamente, similar a los bloques *for*, *if-else*, la sintaxis para heredar una plantilla es bastante similar.

Tomemos, por ejemplo, *index.html* que constaba solo de una etiqueta <h1>. Podemos usar este tipo de plantilla en otras plantillas para convertirla realmente en la página de inicio. Para eso, primero necesitamos encerrar la plantilla en un *block*, que es lo que nos permite usarla en otras

plantillas. Para crear un *block*, simplemente necesitamos escribir la siguiente sintaxis antes del componente que no queremos en otras plantillas:

Plantilla *index.html*:

```
<!DOCTYPE html>
<html lang="es-MX">
<head>
    <meta charset="UTF-8">
    <title>Ejemplo de HTML</title>
</head>
<body>
    <h1>Hola, Mundo!</h1>
    <h2>Esto será heredado</h2>
    {% block ere %}
    <h2>Esto no será heredado</h2>
    {% endblock %}
</body>
</html>
```

En esto hemos usado el *block* con un nombre *ere* como *{% block ere %}* este puede ser lo que quieras. Terminamos el bloque con una sintaxis similar a la de los bloques *for/if* como *{% endblock %}*. Cualquier cosa entre los bloques, es decir, *block ere* y *endblock* no se hereda, es decir, es exclusivo de esta plantilla.

Veremos cómo podemos usar esta plantilla en otras plantillas. De hecho, extenderemos esta plantilla y usaremos los bloques para representar el contenido en la plantilla.

Plantilla *condicional.html*:

```
<!DOCTYPE html>
<html lang="es-MX">
<head>
    <meta charset="UTF-8">
    <title>Condicional</title>
```

```
</head>
<body>
    {% extends 'index.html' %}
    {% block ere %}
    {{  num }}
    {% if num > 10 %}
        <h2>Es mayor a 10</h2>
    {% elif num == 10 %}
        <h2>¡Es diez!</h2>
    {% else  %}
        <h2>Es menor a 10</h2>
    {% endif %}
    {% endblock %}
</body>
</html>
```

Entonces, primero le decimos a Django que extienda la plantilla *index.html*, es decir, Django solo cargará los bloques de esta plantilla, recuerde que solo cargará y no usará los bloques hasta que se lo indiquemos explícitamente.

Para usar los bloques o el tipo de complemento en el contenido de la plantilla *condicional.html* o en cualquier otra plantilla, debemos volver a llamar al *block*. Aquí, debemos escribir el contenido dentro del *block* para analizar correctamente los bloques, ya que se trata de una plantilla HTML. El orden de los elementos de apertura y cierre sí importa.
Entonces, cuando decimos *endblock* se carga la última parte de la plantilla base, es decir, el cierre de las etiquetas *body* y *html* . Esto es como tapar la plantilla antes y después del cuerpo del bloque.

Funciones en *views.py*:

```
from django.shortcuts import render
from random import randint
def inicio(request):
    return render(request,'index.html')

def variable(request):
    nombre = "Marco"
```

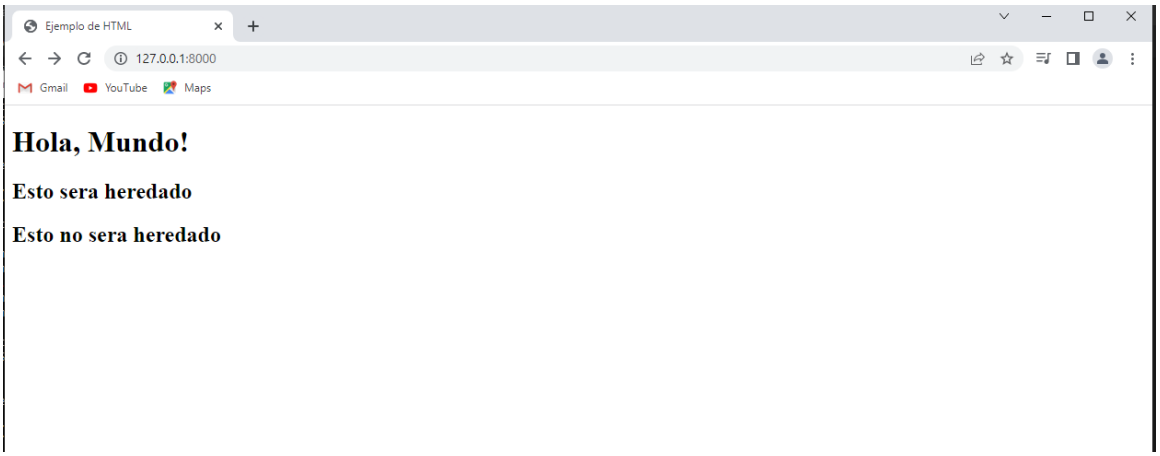
```
 8 return render(request, 'variable.html', {'nombre':nombre})
 9
10 def condicion(request):
11     numero = randint(1,20)
12     return render(request, 'condicional.html', {'num':numero})
13
14 def bucle(request):
15      comida = ('hamburguesa', 'pizza', 'palomitas', 'fruta', 'jamon')
16      return render(request, 'bucle.html', {'comi': comida})
17
18 # Create your views here.
```

URLs en *urls.py*:

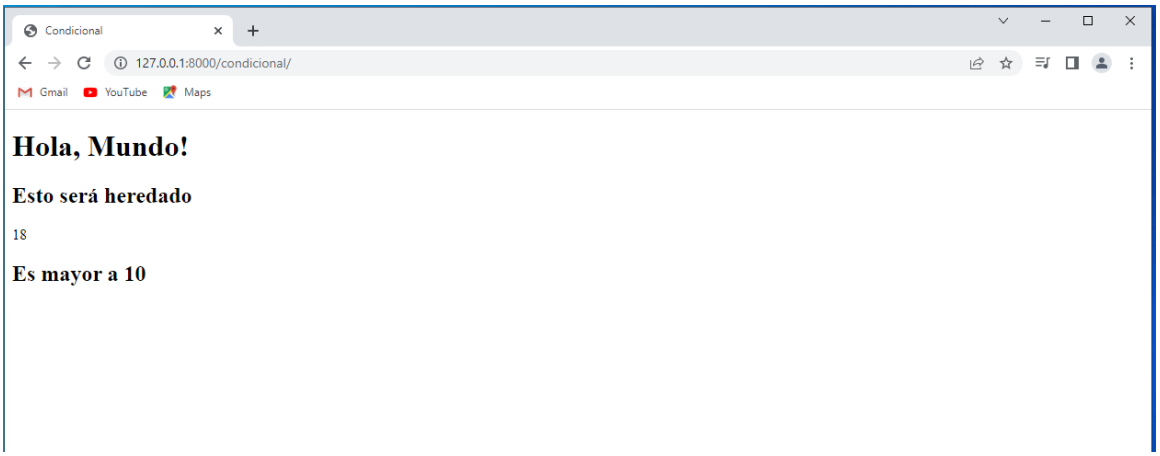
```
1 from .views import inicio, variable, condicion, bucle
2 from django.urls import path
3
4 urlpatterns = [
5     path('inicio/', inicio, name="inicio"),
6     path('variable/', variable, name="variable"),
7     path('condicional/', condicion, name="condicion"),
8     path('bucle/', bucle, name="bucle"),
9 ]
```

Si ejecutamos esto con **python manage.py runserver:**

Tendremos primero la plantilla *index.html* con la estructura de la herencia:



Si vamos a *http://127.0.0.1:8000/condicional*, tendremos la plantilla *condicional.html* en ejecución, esta plantilla demuestra la herencia de plantillas.



Conclusión

Entonces, en este capítulo, pudimos comprender el concepto de plantillas en Django, pudimos usar variables, bucles, declaraciones condicionales y la herencia de plantillas en una aplicación de Django. En el siguiente capítulo, intentaremos retocar los archivos estáticos y veremos cómo estructurarlos y configurarlos correctamente.

Archivos Estáticos

Después de crear plantillas, debería ser bastante tentador agregarles algunos estilos y lógica. Pues sí, veremos cómo agregar archivos estáticos en una aplicación web usando Django. Los archivos estáticos no son solo CSS, sino también archivos multimedia/imágenes y JavaScript. En esta parte del libro, cubriremos los conceptos básicos del trabajo con archivos estáticos en Django, incluida la configuración, la representación y el almacenamiento de los archivos estáticos.

¿Qué son los archivos estáticos?

Los archivos estáticos, como sugiere el nombre, son los archivos que no cambian, sus hojas de estilo (*css/scss*) no cambiarán para cada solicitud del lado del cliente, aunque la plantilla puede ser dinámica. Además, su logotipo, las imágenes en el diseño no cambiarán a menos que lo rediseñe. Así que estos son los archivos estáticos que deben renderizarse junto con las plantillas.

Tenemos básicamente 3 tipos de archivos estáticos, *archivos CSS*, *archivos JavaScript* y *archivos multimedia*. Todos se representan de la misma manera pero según sus convenciones y uso.

Cómo Configurar los Archivos Estáticos

En primer lugar, puede crear una carpeta para todos los archivos estáticos en la carpeta raíz. Por lo general, la convención es estática como el nombre de la carpeta. Entonces, si ha creado la carpeta de plantillas en el directorio raíz, se puede crear una carpeta estática similar en esa ruta.

A continuación, después de crear la carpeta estática en la carpeta raíz del proyecto, debemos configurar el archivo *settings.py* para decirle al servidor web de Django que busque todos nuestros archivos estáticos en esa carpeta. Para hacer eso, vaya al archivo *settings.py*, ahora ya sabrá dónde está el archivo *settings.py* (dentro de la carpeta con el nombre del proyecto). Agregue lo siguiente al final del archivo *settings.py*.

```
1 # import os
2 STATIC_URL = '/static/'
3
4 STATICFILES_DIRS = (
5     os.path.join(BASE_DIR, "static/"),
6 )
```

Ignore el *import os* si ya ha importado y el *STATIC_URL* si ya está en el archivo. *STATICFILES_DIRS* es la configuración que le dice al entorno de Django donde buscar todos nuestros archivos estáticos en el directorio raíz del proyecto donde está la carpeta *static/.* Él *os.path.join()* en realidad obtiene la ruta del directorio en nuestro sistema operativo a la carpeta especificada en el caso de nuestro proyecto, *BASE_DIR* es la ruta del proyecto. La pieza final y crucial es la ruta "*static/*", esta puede ser otra ubicación donde haya creado su carpeta estática dentro del proyecto.

¡Eso es todo! Sí, es así de simple. Ahora podemos crear archivos estáticos y representarlos en nuestras plantillas.

Creación y Almacenamiento de Archivos Estáticos

Ahora, esta parte es personalizable y depende de su preferencia, cómo desea organizar la carpeta estática. La convención que sigo es crear carpetas separadas, principalmente para *css*, *js* y *img*. Y dentro de estas carpetas puede almacenar los respectivos archivos estáticos. Esto también hace que el proyecto sea más escalable en términos de su mantenimiento.

Aquí tiene una imagen con la estructura de directorios:

Vamos a crear un archivo estático y una imagen para demostrar el concepto de archivos estáticos en Django.

Creamos el siguiente archivo en el directorio *css* de la carpeta *static*, en mi caso lo llamare *style.css*:

```
body
{
    background-color:#1d1dff;
    color:white;
}

h1
{
    text-align:center;
    font-family:monospace;
}

h2
{
    color:#ff6600;
    font-weight:500;
}

ul
{
    list-style-type:square;
}
```

Imagen de demostración (mi logo):

Renderizar Archivos Estáticos Desde Plantillas

Entonces, después de configurar y crear los archivos estáticos, ahora podemos inyectarlos en nuestras plantillas. Si intenta hacerlo de la manera tradicional, es decir, vinculando las hojas de estilos en las plantillas HTML, simplemente no funcionará como esperaba y no tiene sentido usar la forma tradicional al crear una aplicación web con un framework. Por lo tanto, hay una forma específica de hacer las cosas en un framework, que lo hace más fácil y eficiente para el proyecto.

Para renderizar cualquier archivo estático, necesitamos cargar la etiqueta estática que nos permite incrustar enlaces para los archivos estáticos en las plantillas. Esto significa que si los archivos estáticos no se cargan directamente en producción (desplegando nuestra aplicación), los archivos estáticos se almacenan en una carpeta *STATIC_ROOT* que luego carga el servidor.

Para cargar los archivos estáticos desde su configuración, simplemente podemos incluir la etiqueta en la parte superior de la plantilla.

```
1 {% load static %}
```

La etiqueta de plantilla anterior cargará la etiqueta estática que nos permite incrustar los enlaces a los archivos estáticos como se explicó anteriormente.

Ahora, podemos acceder a cualquier archivo en la carpeta *static* mediante nuestra plantilla con una sintaxis particular como se muestra a continuación:

```
<link rel="stylesheet" href="{% static 'css/style.css' %}">
```

Es solo un ejemplo de cómo cargar el archivo, llamamos a la etiqueta *static* que hemos cargado previamente y desde allí hacemos referencia al archivo *css*. La sintaxis compacta sería: *{% static 'archivo_raíz ' %}*.

NOTA: La ruta al archivo estático es relativa a la carpeta *static*, es decir, ingrese la ruta del archivo considerando la carpeta *static* como el directorio base.

Demostración del Archivo Estático

Transformemos el archivo estático que creamos anteriormente en una plantilla, es decir, el archivo *css* y la *imagen*.

Suponiendo que tiene una aplicación llamada *app1* en su proyecto de Django, puede renderizar archivos estáticos como se muestra a continuación.

Agregué el siguiente código a *index.html*:

```
<!DOCTYPE html>
<html lang="es-MX">
<head>
    <meta charset="UTF-8">
    <title>Ejemplo de HTML</title>
    {% load static %}
    <link rel="stylesheet" href="{% static 'css/style.css' %}">
</head>
<body>
```

```
10      <h1>Hola, Mundo!</h1>
11      <h2>Esto será heredado</h2>
12      {% block ere %}
13      <h2>Esto no será heredado</h2>
14      {% endblock %}
15  </body>
16  </html>
```

Estamos cargando la etiqueta *static* y luego cargando el archivo *css* usando la sintaxis de la etiqueta como se explicó anteriormente.

Este seria nuestro archivo *style.css*:

```
 1  body
 2  {
 3      background-color:#1d1dff;
 4      color:white;
 5  }
 6
 7  h1
 8  {
 9      text-align:center;
10      font-family:monospace;
11  }
12
13  h2
14  {
15      color:#ff6600;
16      font-weight:500;
17  }
18
19  ul
20  {
21      list-style-type:square;
22  }
```

Este es el archivo estático, *style.css* almacenado dentro de la carpeta *css* de la carpeta *static*. Esto contiene un estilo CSS básico.

Después tenemos la función de la vista en el archivo *views.py*:

```
1 from django.shortcuts import render
2 from random import randint
3
4 def inicio(request):
5     return render(request,'index.html')
```

El archivo *views.py* tiene la función de renderizar la plantilla *index.html* desde la carpeta de plantillas dentro de la carpeta específica de la aplicación.

No olvides agregar la URL al archivo *urls.py*:

```
1 from .views import inicio, variable, condicion, bucle
2 from django.urls import path
3
4 urlpatterns = [
5     path('inicio/', inicio, name="inicio"),
6 ]
```

Esta es la configuración a nivel de aplicación para las rutas URL a las vistas que vinculan las funciones del archivo *views.py*.

También tienes que agregar tolas las URLs de la aplicación al archivo *urls.py* del proyecto:

```
1 from django.contrib import admin
2 from django.urls import path, include
3 from django.views.generic import TemplateView
4
5
6 urlpatterns = [
```

```
    path('admin/', admin.site.urls),
    path('', include('app1.urls')),
    path('', TemplateView.as_view(template_name="index.html"),
  name="index"),
]
```

Agregue lo siguiente en *settings.py* si sus plantillas y archivos estáticos no están configurados correctamente:

```
import os

TEMPLATES = [
    {
        'BACKEND':
  'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, 'template'),],
        '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',
            ],
        },
    },
]

STATIC_URL = '/static/'

STATICFILES_DIRS = (
    os.path.join(BASE_DIR, "static/"),
)
```

Entonces, el resultado del código anterior es una plantilla simple como se muestra en la imagen a continuación:

Usamos **python manage.py runserver:**

Veamos cómo se procesan los archivos estáticos en las plantillas heredadas. Vamos a jugar con la plantilla *bucle.html* creada en el capítulo anterior.

Primero modificamos la plantilla *bucle.html*:

```
1 <!DOCTYPE html>
2 <html lang="es-MX">
3 <head>
4     <meta charset="UTF-8">
5     <title>Bucle</title>
6 </head>
7 <body>
8     {% extends 'index.html' %}
```

```
 9     {% load static %}
10
11     {% block ere %}
12     <img src="{% static 'img/logo.png' %}" height="200px"
   width="200px" />
13     <ul>
14         {% for comida in comi %}
15         <li>{{ comida }}</li>
16         {% endfor %}
17     </ul>
18     {% endblock %}
19 </body>
20 </html>
```

Tendremos que volver a cargar la etiqueta *static* para cada plantilla solo si necesitamos incluir un nuevo archivo estático en la plantilla. Así que usamos *{% load static %}* nuevamente mientras cargamos el archivo estático *logo.pgn* en esta plantilla.

Modificamos las funciones en *views.py*:

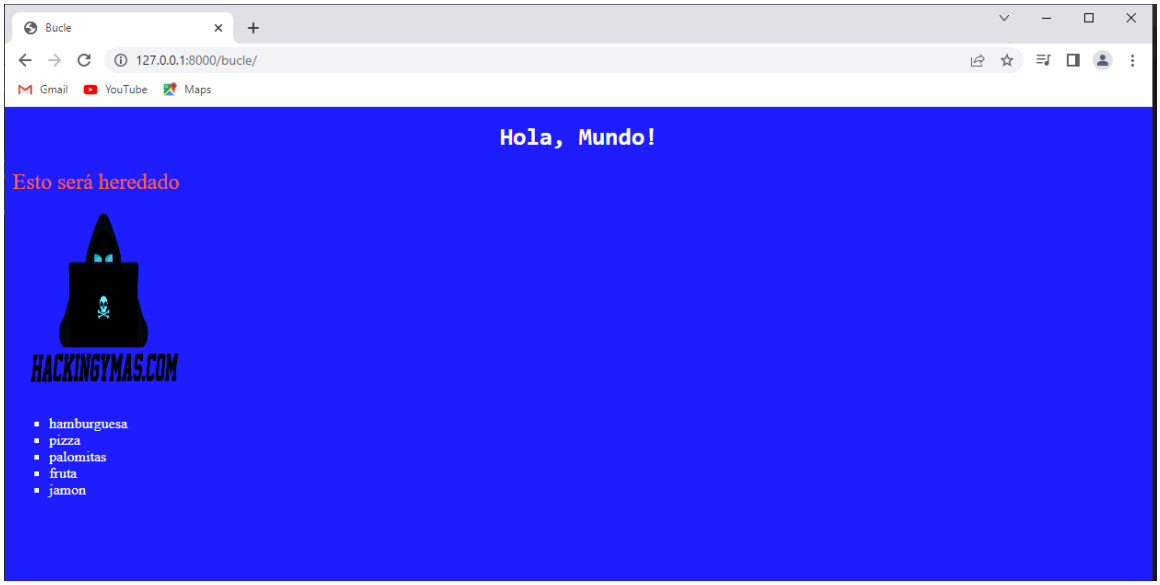
```
1 from django.shortcuts import render
2 from random import randint
3
4 def inicio(request):
5     return render(request,'index.html')
6
7 def bucle(request):
8     comida = ('hamburguesa', 'pizza', 'palomitas', 'fruta', 'jamon')
9     return render(request, 'bucle.html', {'comi': comida})
```

También agregamos una nueva URL a *urls.py*:

```
1 from .views import inicio, variable, condicion, bucle
2 from django.urls import path
```

```
3
4 urlpatterns = [
5     path('inicio/', inicio, name="inicio"),
6     path('bucle/', bucle, name="bucle"),
7 ]
```

Entonces, esa es la URL y la función creada, ahora podemos ver el resultado en la URL *http://127.0.0.1:8000/bucle/* para ver el siguiente resultado:



El estilo de la lista se ha cambiado y, por lo tanto, podemos ver que el CSS de la plantilla *index.html* también se hereda.

Aquí está la estructura del proyecto de Django que hemos creado hasta ahora:

Conclusión

Entonces, en este capítulo, pudimos configurar y renderizar archivos estáticos como CSS, imágenes y, opcionalmente JavaScript. Cubrimos desde cero cómo configurar, cargar y estructurar la carpeta para almacenar todos los archivos estáticos a nivel de proyecto.

Creación de Modelos

Hemos visto los conceptos básicos de las plantillas de Django en los capítulos anteriores. Ahora, podemos pasar a cosas de backend en Django como las bases de datos, las consultas, la sección de administración, etc. En esta parte en particular, cubriremos la parte fundamental de cualquier aplicación en Django, es decir, los modelos. Entenderemos qué es un modelo, cómo estructurar uno, cómo crear relaciones y agregar restricciones en los campos, etc.

¿Qué son los Modelos?

Un modelo es una forma de Django (Python) para estructurar una base de datos para una aplicación dada. Es técnicamente una clase que puede actuar como una tabla en una base de datos en general y dentro de la clase, las propiedades de la misma actúan como los atributos de esa base de datos. Es así de simple. Solo un plano para crear una tabla en una base de datos, no se preocupe por qué y dónde está nuestra base de datos. Exploraremos la base de datos y su configuración en la siguiente parte.

Al crear un modelo, no tiene que escribir todas las consultas SQL básicas, como en el siguiente ejemplo:

```
1    CREATE TABLE cliente(
2        id INT PRIMARY KEY,
3        nom VARCHAR(50),
4        ape VARCHAR(100),
5        dir VARCHAR(200)
6    );
```

Si su aplicación es bastante grande o es compleja en términos de las relaciones entre las entidades, escribir consultas SQL manualmente es una tarea abrumadora y también bastante repetitiva en ocasiones. Así que Django maneja toda la basura de SQL fuera del camino para el programador. Por lo tanto, los modelos son solo una forma de Python para crear una tabla para la base de datos de la aplicación.

¿Cómo Crear un Modelo?

Crear un modelo para una aplicación es tan fácil como crear una clase en Python. ¡Pero hey! Es más que eso, ya que hay otras preguntas que abordar al diseñar la clase. Debe diseñar la base de datos antes de definir los campos en el modelo.

De acuerdo, bueno, no es tan sencillo como parece, pero aun así puede crear proyectos simples y ficticios para empezar. Puede usar ciertas herramientas como *lucidchart*, *dbdiagrams.io* y otras herramientas con las que se sienta cómodo. Es importante visualizar el esquema de la base de datos o la estructura de la aplicación antes de jugar con la base de datos real dentro del proyecto. No nos volvamos locos y diseñemos un modelo simple para entender el proceso, usaremos el ejemplo más usado por todos, el modelo de un blog.

Aquí hay un modelo básico para un Blog:

```
 1 from django.db import models
 2
 3 # Create your models here.
 4 from django.contrib.auth.models import User
 5
 6 class Articulo(models.Model):
 7     title = models.CharField(max_length=255)
 8     post = models.TextField()
 9     author = models.ForeignKey(User, on_delete=models.CASCADE,
   related_name='Article')
10     created = models.DateTimeField(auto_now_add=True)
11     updated = models.DateTimeField(auto_now=True)
```

Este es un modelo básico con el que quizás quieras jugar, pero no lo tires a ningún lado.

Definimos o creamos nuestros modelos en la aplicación dentro del proyecto. Dentro de la aplicación ya hay un archivo llamado *models.py*, simplemente agregue el código anterior. La aplicación puede ser cualquier aplicación que tenga más sentido para usted o, mejor, cree una aplicación si aún no la ha creado y asígnele el nombre de artículo, publicación o lo que desee.

Si está familiarizado con Python POO (programación orientada a objetos), básicamente hemos heredado la clase *models.Model* del módulo *django.db* en nuestro modelo.

Si desea más ejemplos de este tipo, veamos más modelos de este tipo, **no se recomienda su uso en aplicaciones reales así que solo son ejemplos:**

Un modelo básico de aplicación de correo electrónico. Atributos como el remitente (*sender*), el asunto del correo (*subject*), el cuerpo del correo (*body*), el destinatario (*recipients_list*), es decir, la sección *Para*, en un sistema de correo y el archivo adjunto (*atteachment_file*) para un archivo adjunto a un correo, si corresponde.

```
1 # Create your models here.
2 class EMail(models.Model):
3     sender = models.EmailField(max_length = 255)
4     subject = models.CharField(max_length = 78)
5     body = models.CharField(max_length = 40000)
6     recipients_list = models.ManyToManyField(EmailUser, related_name
  = 'mail_list')
7     attachment_file = models.FileField(blank=True)
```

Un modelo de ejemplo para una aplicación de toma de notas, que consta de una nota y un libro. Un libro puede ser una colección de múltiples notas, es decir, un solo libro puede tener varias notas, por lo que estamos usando un *ManyToManyField*, ¿qué es eso? Lo veremos en breve.

```
1 # Create your models here.
2 class Notas(models.Model):
3     author = models.ForeignKey(User, on_delete=models.CASCADE)
4     title = models.CharField(max_length = 1024)
```

```
    content = models.Textfield()
    created = models.DateTimeField(auto_now_add = True)
    modified = models.DateTimeField(auto_now = True)
    book = models.ManyToManyField(Book, related_name = 'book')

class Libro():
    name = models.CharField(max_length = 1024)
```

Estos son solo ejemplos y no se recomienda su uso en ningún lugar, especialmente en un proyecto serio. Entonces, hemos visto un modelo, pero, ¿Que son estos campos y las restricciones como *on_delete* y *max_length*?, esto lo veremos en la siguiente sección.

Campos en Django

Los campos son técnicamente los atributos de la clase que aquí es el modelo, pero además se tratan como un atributo en una tabla de una base de datos. Entonces, el modelo se convierte en una lista de atributos que luego se analizarán en una base de datos real.

Al crear atributos dentro de una clase, estamos definiendo la estructura de una tabla. Django ya ha definido varios tipos de campos para facilitar la validación y la configuración restringida del esquema de la base de datos.

Veamos algunos de los tipos de campos en los modelos de Django.

Tipos de Campos

Django tiene muchos campos definidos en la clase de modelos. Si desea revisar todos los campos, lea las referencias en la documentación oficial de Django. Podemos acceder a los campos desde el módulo *models* como *name = models.CharField(max_length=10)*, este es un ejemplo de cómo definir un atributo *name* que es un *CharField*. Podemos establecer *max_length* que actúa como una restricción para el atributo, ya que no queremos que el campo de nombre sea mayor que 10 y, por lo tanto, analizar el parámetro *max_length* en 10.

Tenemos otros tipos de campos como:

1) **IntegerField:** Para un valor entero.
2) **TextField:** Para entrada larga de texto (como área de texto en HTML).
3) **EmailField:** Para un único campo de correo electrónico válido.
4) **DateField:** Para ingresar en un formato de fecha.
5) **URLField:** Para ingresar un campo de URL.
6) **BooleanField:** Para una entrada de valor booleano.

Y también hay otros campos que se pueden usar según los requisitos.

También tenemos algunos otros campos que no son directamente campos, por así decirlo, pero son campos que definen una relación como:

1) **ForeignKey:** Define una relación de muchos a uno con otro modelo/clase.
2) **ManyToManyField:** Define una relación de muchos a muchos con otro modelo/clase.
3) **OneToOneField:** Define una relación uno a uno entre diferentes modelos/clases.

Entonces, se trata de los tipos de campo para tener una idea de cómo estructurar o diseñar una tabla de base de datos utilizando un modelo con algunos tipos de atributos. También debemos hablar sobre las restricciones que deben agregarse a los campos dentro de los modelos.

Campos: Opciones y Argumentos

Podemos agregar restricciones y pasar argumentos a los campos en los modelos. Podemos agregar argumentos como *null*, *blank*, *defualt*, *choices*, etc.

1) **null=True/False:** Establezca una verificación para la entrada en la tabla como no nula en la base de datos.
2) **blank=True/False:** Establezca una verificación para que la validación de entrada esté vacía o no.
3) **unique=True/False:** Establezca una restricción para que la entrada sea única en toda la tabla.
4) **defualt=anyvalue:** Establecer un valor predeterminado para el campo.
5) **choices=list:** Establecer una lista de opciones definidas para seleccionar en el campo (una lista de dos tuplas valoradas).

También tenemos otra restricciones específicas para los campos como *max_length* para *CharField*, *on_delete* para *ForeignKey* que se puede usar como controlador para el modelo cuando se elimina el modelo relacionado, *verbose_name* para establecer un nombre diferente para hacer referencia a la entrada en el modelo de la sección de administración en comparación con el nombre predeterminado del modelo, *verbose_name_plural* similar a *verbose_name* pero para hacer referencia a todo el modelo. También *auto_now_add* y *auto_now* para *DateTimeField* para establecer la fecha y hora actual de forma predeterminada.

Más opciones y argumentos se pueden pasar a los campos, si quieres saber más, puedes visitar la documentación oficial de Django.

Estas son algunas de las opciones o argumentos que podemos o necesitamos pasar a los campos para configurar un esquema restringido para nuestra base de datos.

Clase Meta

La clase meta es una clase anidada dentro de la clase modelo que se usa la mayoría de las veces para ordenar las entradas (objetos) en la tabla, administrar permisos para acceder al modelo, agregar restricciones a los modelos relacionados con los atributos dentro de él, etc.

Métodos del Modelo

Como una clase puede tener funciones, también lo hace un modelo, ya que después de todo es una clase de Python. Podemos crear una especie de método auxiliar dentro del modelo. La clase modelo proporciona un útil método *__str__()* que se utiliza para cambiar el nombre de un objeto de la base de datos. También tenemos otras métodos auxiliares predefinidos como *get_absolute_url* que genera la URL y la devuelve para su posterior redireccionamiento o representación.

ORM de Django

Django tiene un mapeador relacional de objetos que es el concepto central en Django o el componente en Django que nos permite interactuar con la base de datos sin que el programador escriba consultas SQL/DB. Es una forma en la que Python escribe y ejecuta consultas sql, básicamente abstrae la capa para escribir manualmente consultas SQL.

Es realmente interesante y fascinante para un principiante crear aplicaciones web sin aprender SQL (no recomendado). Por ahora, es simplemente mágico ver a Django manejando las operaciones de la base de datos por ti.

Modelo de Ejemplo

Establezcamos un modelo a partir de lo que hemos aprendido hasta ahora.

```
1 from django.db import models
2
3 # Create your models here.
4
5 class Articulo(models.Model):
6     title = models.CharField(max_length=255)
7     post = models.TextField()
```

```
 8     created = models.DateTimeField(auto_now_add=True)
 9     updated = models.DateTimeField(auto_now=True)
10
11
12     def __str__(self):
13         return self.title
```

Entonces, podemos ver que Django nos permite estructurar el esquema de una base de datos. Aunque no se ve nada como resultado final, cuando configuremos y migremos el modelo a nuestra base de datos, veremos los resultados del arduo trabajo realizado para crear y diseñar el modelo.

Conclusión

Entendimos los conceptos básicos de la creación de un modelo. Todavía no tocamos la base de datos, pero el siguiente capítulo trata sobre la configuración y la migración, por lo que nos pondremos manos a la obra con las bases de datos. Cubrimos cómo estructurar nuestra base de datos, cómo escribir campos en el modelo, agregarles restricciones y lógica y exploramos las terminologías en Django.

Configuraciones de Base de Datos

En esta parte del libro, crearemos una base de datos externa y configuraremos los ajustes para esa base de datos. También analizaremos el proceso de migración, que consiste en convertir el modelo que creamos en la parte anterior en una estructura real de la base de datos. Usaré *PostgreSQL* para la mayoría de las demostraciones, pero debería ser similar para otras herramientas de administración de bases de datos. En este capítulo, haremos que su proyecto de Django se vincule a una base de datos local.

Selección de una Base de Datos

Veremos qué opciones tienes al seleccionar una base de datos.

Bases de Datos SQL:

1) **sqlite**
2) **PostgreSQL**
3) **MySQL**
4) **MariaDB**
5) **Oracle**

La selección de una base de datos depende en gran medida del tipo de aplicación que vaya a realizar, pero la mayoría de las veces es SQL, simplemente porque ha estado dominando el desarrollo de todo tipo de aplicaciones durante 4 décadas. Aun así, NoSQL está creciendo en popularidad y tiene algunas ventajas sobre SQL en muchas aplicaciones modernas. Debe analizar su proyecto un poco más a fondo y comprender mejor el flujo de datos para tomar una decisión sobre SQL y No-SQL, pero la mayoría de las veces será SQL.

Además, Django no admite oficialmente NoSQL, por lo que tendrá que activar algunas bibliotecas de terceros para integrar y administrar una base de datos.

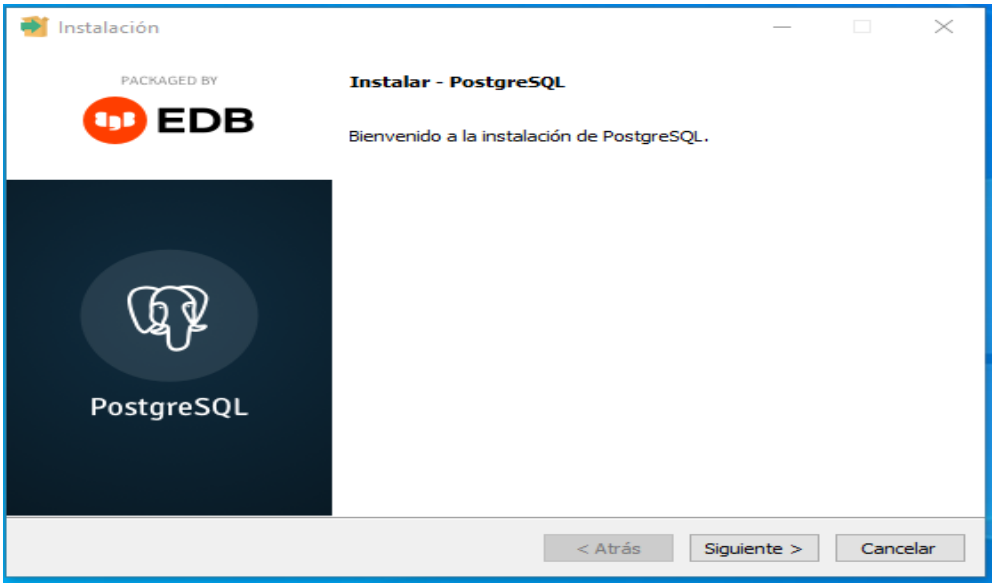
Hay muchas bases de datos como PostgreSQL, MySQL, MariaDB, Oracle, etc. Debe elegir la que le resulte más cómoda y la que mejor se adapte a la arquitectura y los requisitos de su proyecto.

Cómo Instalar PostgreSQL

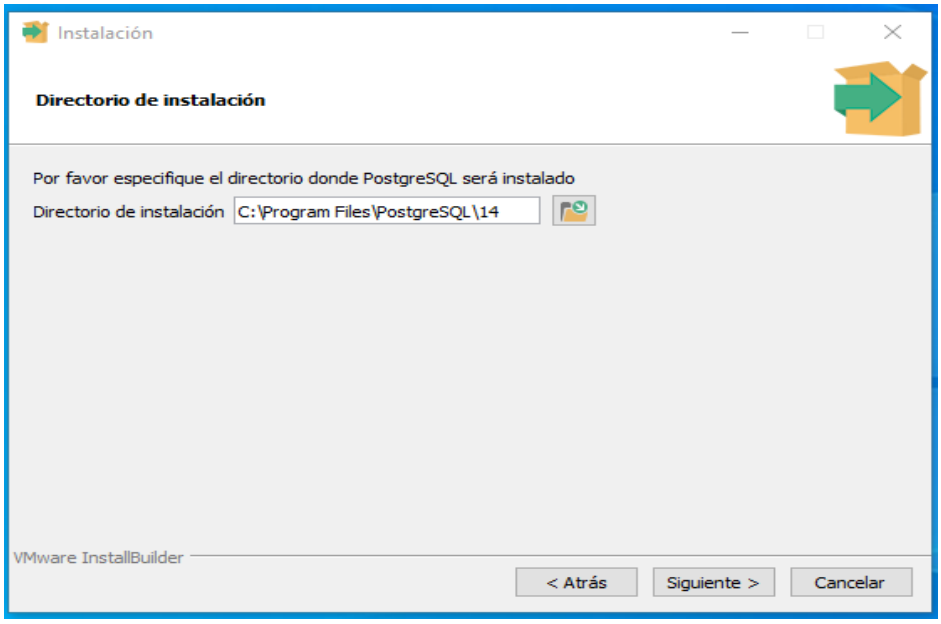
Esta es una guía paso a paso para instalar PostgreSQL en una máquina con Windows. Puede descargar el último instalador estable de PostgreSQL específico para su Windows haciendo clic aquí.

Después de descargar el instalador, haga doble clic en él y siga los pasos a continuación:

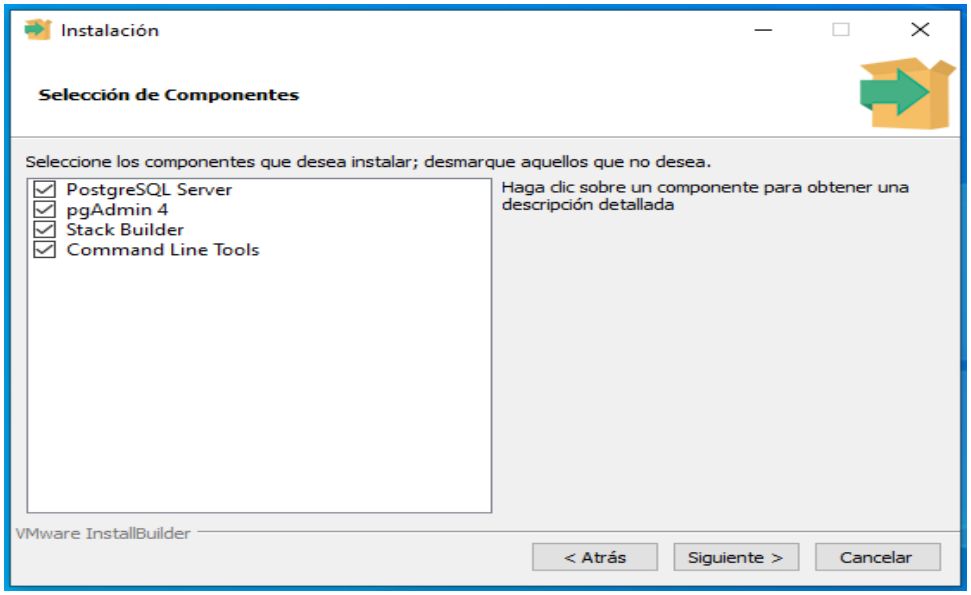
Paso 1: haga clic en el botón Siguiente.



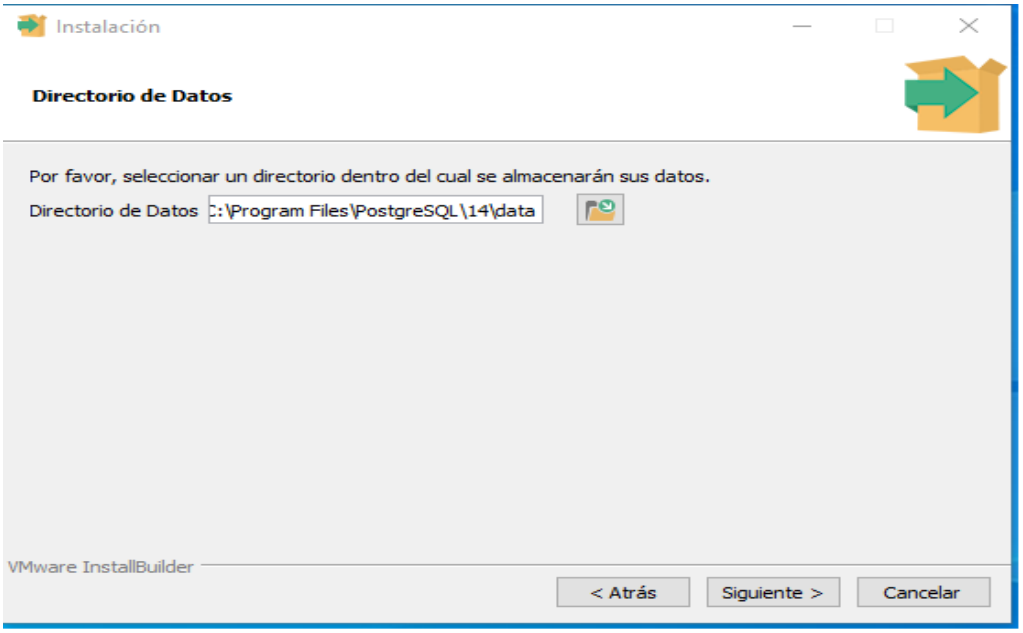
Paso 2: elija la carpeta de instalación, donde desea que se instale PostgreSQL, y haga clic en Siguiente.



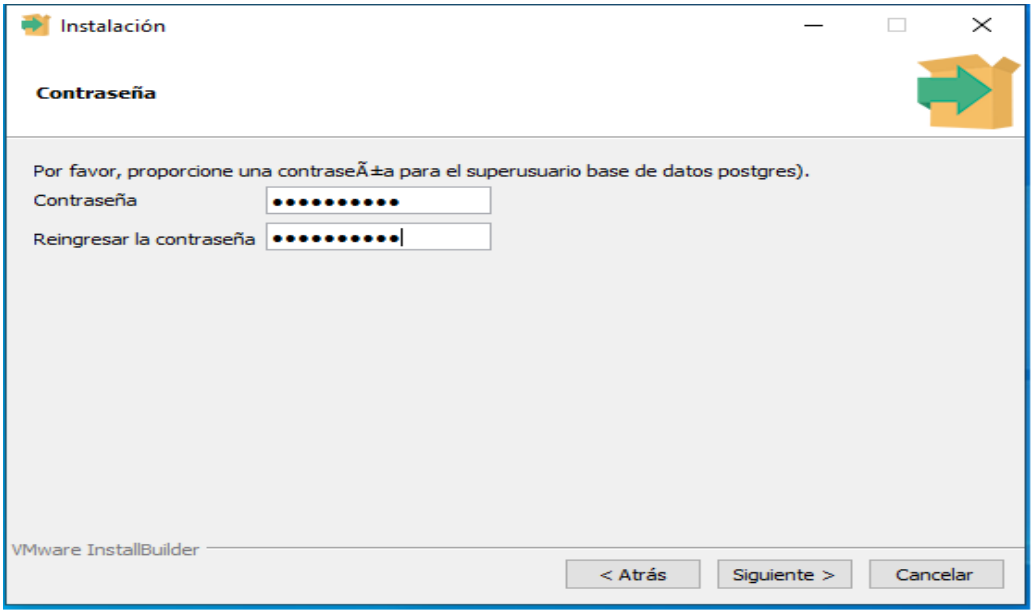
Paso 3: seleccione los componentes según sus requisitos para instalar y haga clic en el botón Siguiente.



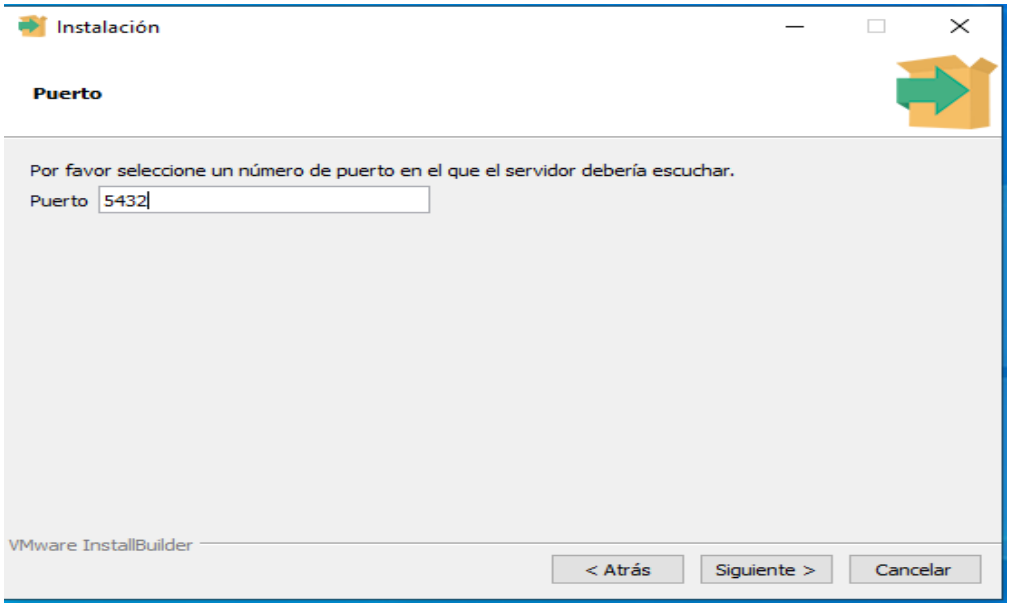
Paso 4: seleccione el directorio de la base de datos donde desea almacenar los datos y haga clic en Siguiente.



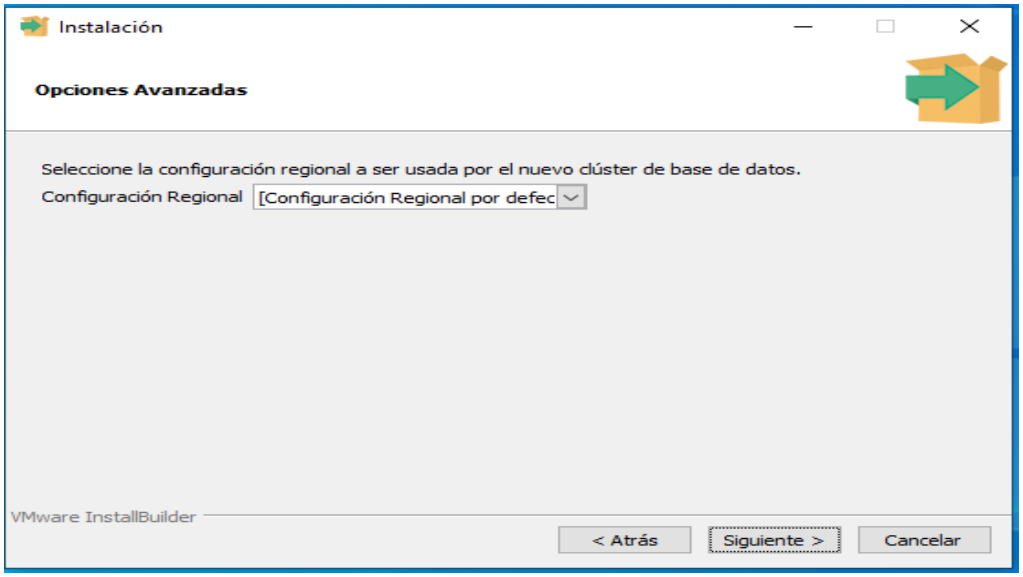
Paso 5: establezca la contraseña para el superusuario de la base de datos.



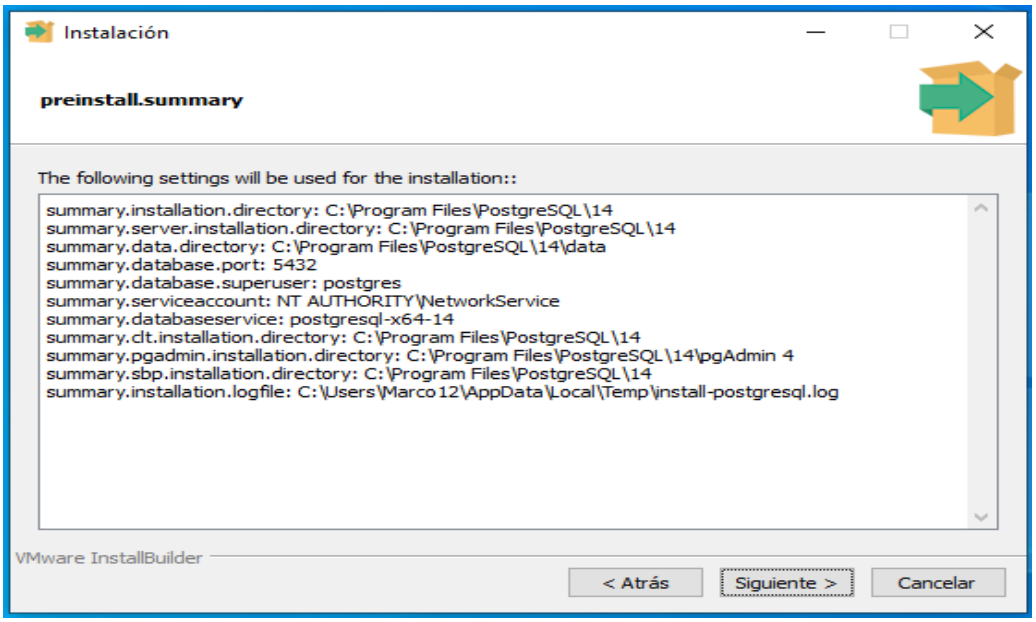
Paso 6: Configure el puerto para PostgreSQL. Asegúrese de que ninguna otra aplicación esté utilizando este puerto. Si no está seguro, déjelo en su valor predeterminado (5432) y haga clic en Siguiente.



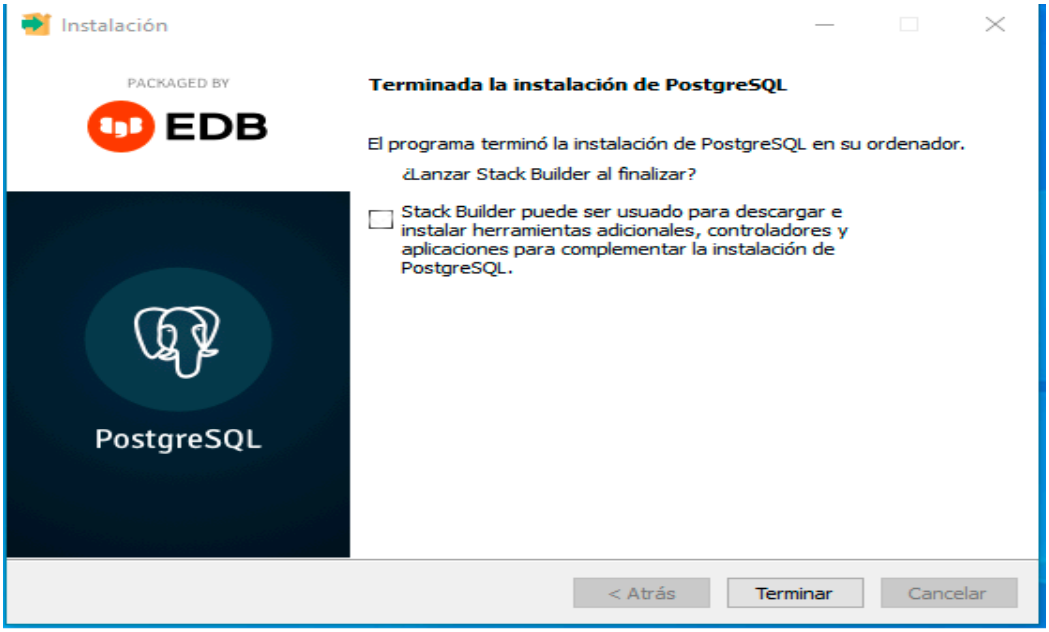
Paso 7: elija la configuración regional predeterminada utilizada por la base de datos y haga clic en el botón Siguiente.



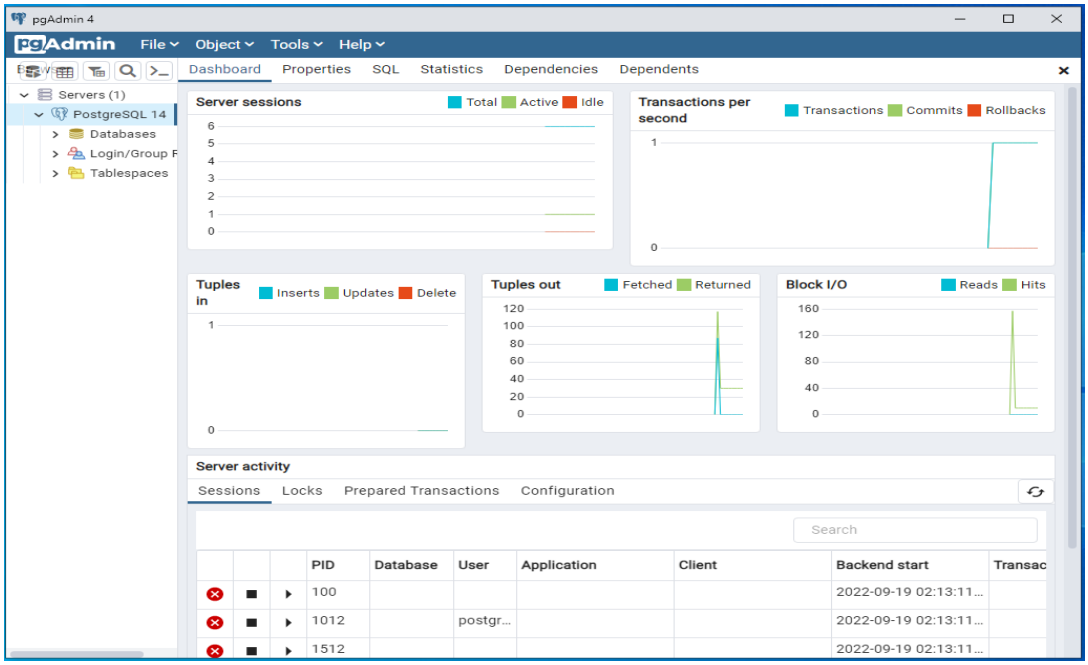
Paso 8: Haga clic en el botón Siguiente para iniciar la instalación.



Paso 9: haga clic en el botón Finalizar para completar la instalación de PostgreSQL.



Podemos utilizar la herramienta pgAdmin4 para gestionar y administrar el servidor PostgreSQL. Para conectarse a PostgreSQL. Inicie pgAdmin4. En la primera pantalla, especifique la contraseña del superusuario que se puede usar para conectarse al servidor PostgreSQL.



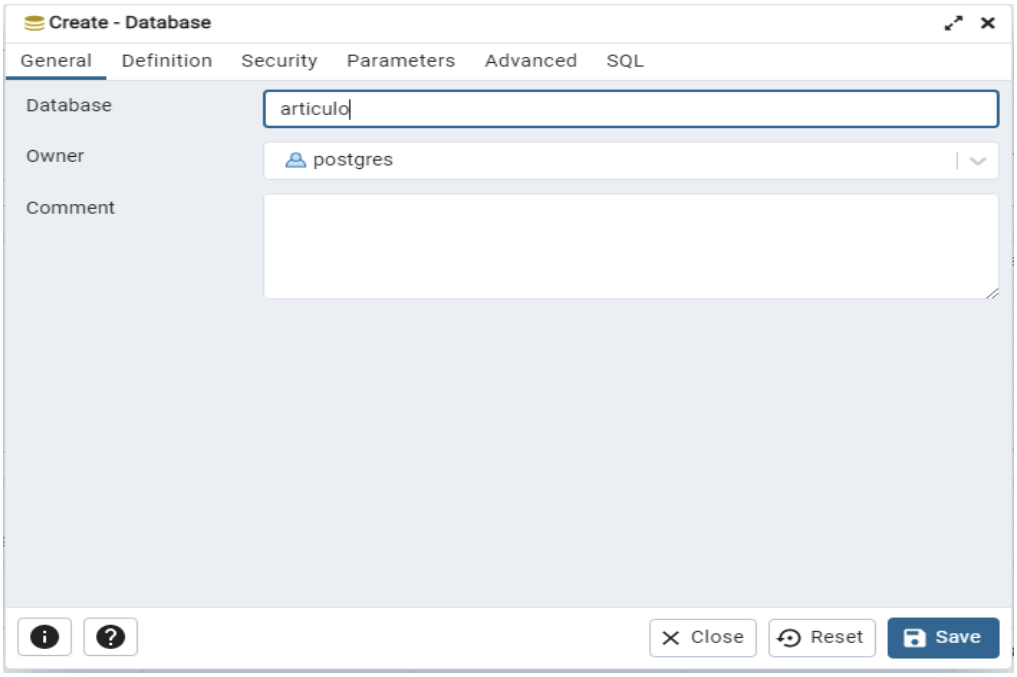
Crear una Base de Datos

Para crear una base de datos, debe ir a la aplicación de administración **DBMS** de la herramienta que está utilizando, para Postgre es pgAdmin4, para MySQL es MySQL Administrator o PHPMyAdmin. Debe hacer la investigación para configurar una base de datos localmente para su proyecto. Pero Django ya está emparejado con SQLite, que es una base de datos relacional pero con algunas peculiaridades. Es realmente genial comenzar con un proyecto sin crear o administrar un sistema de base de datos completo. La base de datos SQLite está contenida en un archivo llamado db.sqlite3.

Si desea trabajar en una base de datos en particular como PostgreSQL, MySQL, etc, debe crear la base de datos utilizando la herramienta de administración y conservar los datos de configuración como *nombre*, *host*, *contraseña*, etc. después de crear la base de datos.

Daré una demostración de cómo crear una base de datos simple en PostgreSQL, pero en su mayoría es un poco diferente en otras herramientas de **DBMS**, ya que cada una de ellas tiene sus propias aplicaciones GUI.

Para crear una base de datos en pgAdmin4, solo tienes que ir al menú **Object-Create-Database,** cuando des clic en **Database** aparecerá una ventana donde tienes que proporcionar un nombre a la base de datos, se verá algo así:



Por último, solo dale a **Save** y listo.

Así es como crea una base de datos en pgAdmin4, debería ser bastante sencillo y simple de seguir también en otras herramientas **DBMS**. También necesitará una contraseña para abrir la interfaz de administración para estas herramientas, así que tenga esa contraseña a mano, la necesitaremos más adelante.

Necesitamos que nuestro proyecto en Django vincule esa base de datos en particular y la use dentro de su aplicación. En la siguiente parte, configuraremos el archivo *settings.py* para acceder a la base de datos desde la máquina local.

Configurando Settings.py

Necesitamos configurar el archivo *settings.py* para integrar la base de datos en nuestro proyecto. Django tiene una sección dedicada para la base de datos en ese archivo. De forma predeterminada, la configuración de la base de datos se crea para la base de datos de *SQLite* que, como dije anteriormente, es la base de datos predeterminada que usa Django si no se especifica y configura.

Entonces, cuando vea por primera vez la configuración de la base de datos en el archivo *settings.py*, verá una configuración como la siguiente:

```
1 DATABASES = {
2     'default': {
3         'ENGINE': 'django.db.backends.sqlite3',
4         'NAME': BASE_DIR / 'db.sqlite3',
5     }
6 }
```

Para *PostgreSQL* o cualquier otra base de datos, necesitaremos más cosas que la configuración de *SQLite*. Es decir, necesitaremos lo siguiente:

- **Nombre** de la base de datos.
- **Usuario** de la base de datos.
- **Contraseña** para ese usuario.
- **Host** y **puerto** para la base de datos.

El puerto es opcional ya que la mayoría de las herramientas de base de datos eligen el puerto predeterminado para su propósito. Para PostgreSQL, el puerto predeterminado es *5432* y para MySQL es *3306*. Déjelo en blanco PORT: ", si no está seguro de cuál es el puerto para esa base de datos. También debemos especificar el ENGINE, ya que es el backend de la base de datos que se utilizará para el proyecto. Hay opciones para bases de datos específicas como se menciona en la documentación.

PostgreSQL

En PostgreSQL, el usuario predeterminado es *postgres*, aunque podría depender de su configuración. pgAdmin4 le pedirá una contraseña cuando instale PostgreSQL por primera vez en su máquina. La contraseña que se utilizará es para el usuario predeterminado de la base de datos. El nombre es el nombre que le dio al crear la base de datos en la sección de administración de PostgreSQL. Finalmente, el host es generalmente *localhost* ya que estamos usando la base de datos local que es nuestro sistema, y el puerto como se dijo anteriormente es *5432* por defecto para PostgreSQL.

Después de configurar *settings.py*, tendrá un archivo como este:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'articulo',
        'USER': 'postgres',
        'PASSWORD': 'ejemplocon',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}
```

También hay otras configuraciones que pueden ser específicas para el propósito y se pueden explorar en la documentación de Django.

Instalación del Adaptador de Base de Datos

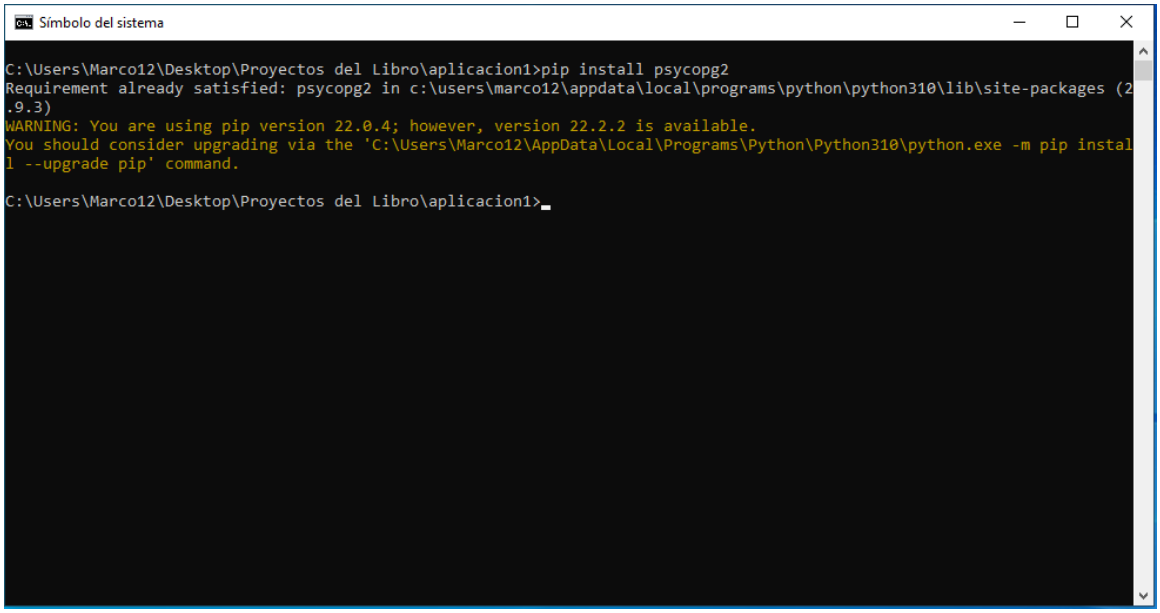
Antes de que podamos hacer algo con la base de datos, necesitamos una cosa que es un adaptador de base de datos. Ahora, esto depende de la base de datos que esté utilizando. La lista es la siguiente:

- **psycopg2** para PostgreSQL
- **mysqlclient** para MySQL y MariaDB
- **cx-Oracle** para Oracle

SQLite no requiere un adaptador ya que la base de datos es un archivo almacenado en el directorio base como db.sqlite3.

La lista anterior no es más que paquetes de Python que permiten que **Django ORM** (que está bajo el capó de Python) opere la base de datos. Para instalarlos, simplemente puede instalar con *pip* como *pip install psycopg2*, *pip install mysqlclient*, y así sucesivamente.

Cuando ejecute *pip install psycopg2*, obtendrá el siguiente resultado:



Verificar la Conexión de la Base de Datos

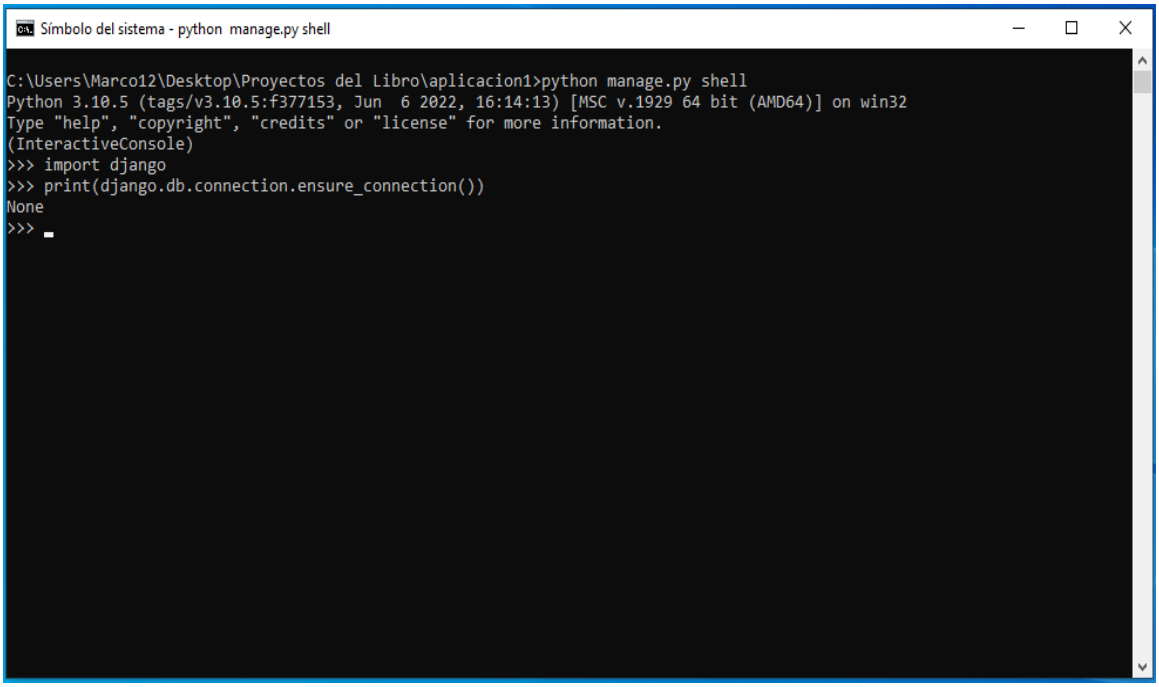
Para verificar si la base de datos estaba realmente vinculada en el proyecto de Django, le presentaré una gran herramienta en Django: *python manage.py Shell*.

Este comando abrirá un intérprete de Python en el shell. Es una consola interactiva para que podamos probar algunos aspectos de nuestro proyecto. Por ejemplo, para comprobar si la base de datos está conectada o no:

Ejecute el código después de ejecutar el comando *python manage.py Shell*:

```
1 import django
2 print(django.db.connection.ensure_connection())
```

Si esto devuelve *None*, está listo para continuar. Y si el resultado son toneladas de mensajes de error, algo falla en la configuración o en la propia base de datos.



Migrar los Modelos

Ahora, tenemos una instancia limpia y fresca de una base de datos creada. ¿Qué sigue? Ahora usaremos la lógica de la parte anterior, donde creamos y diseñamos el modelo. Podemos combinar nuestra base de datos física, es decir, la base de datos que acabamos de crear en una herramienta DBMS, y el modelo lógico para completarla y darle una estructura.

Modelo creado anteriormente para este ejemplo, si no lo tienes, agrégalo a *models.py*:

```
1 from django.db import models
2
3 # Create your models here.
4
5 class Articulo(models.Model):
6     title = models.CharField(max_length=255)
7     post = models.TextField()
8     created = models.DateTimeField(auto_now_add=True)
9     updated = models.DateTimeField(auto_now=True)
10
11
12     def __str__(self):
13         return self.title
```

Realizaremos la migración en nuestra base de datos.

Aquí es donde ocurre la magia y probablemente el paso que debe ejecutarse con cuidado si se trabaja con una base de datos en tiempo real (nivel de producción). Porque los comandos que ejecutará afectarán directamente a la base de datos.

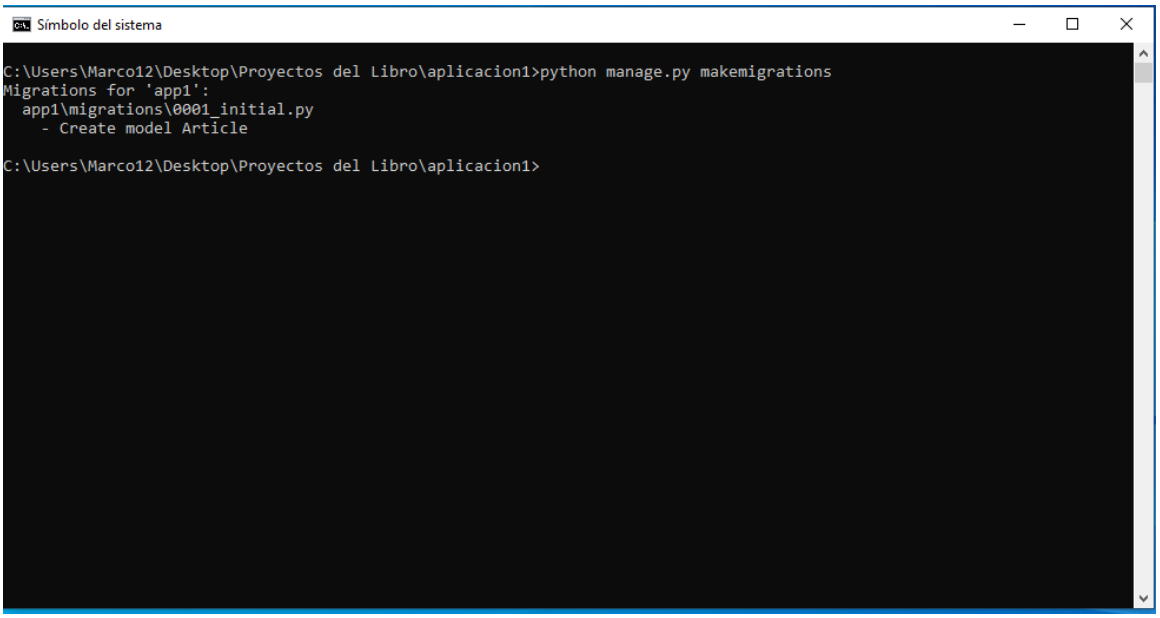
Hacer una migración es un proceso de dos pasos. Pero, ¿qué es la migración?

La documentación de Django lo establece como el control de versión para el esquema de la base de datos y toma su palabra al respecto. Básicamente es una carpeta (oculta) que almacena el estado de la estructura de su base de datos al igual que las confirmaciones en git.

Comando Makemigrations

Este comando crea un archivo dentro de la carpeta *migrations* dentro de la carpeta de la aplicación cuyo modelo se ha creado o actualizado recientemente. No afecta a la base de datos, pero crea un archivo que, a su vez, después de un comando *migrate*, se analizará en la base de datos real utilizando el ORM. Entonces, para cualquier atributo o cambio lógico dentro del modelo, usamos el comando *makemigrations* como se muestra a continuación:

Ejecutamos el comando **python manage.py makemigrations**:

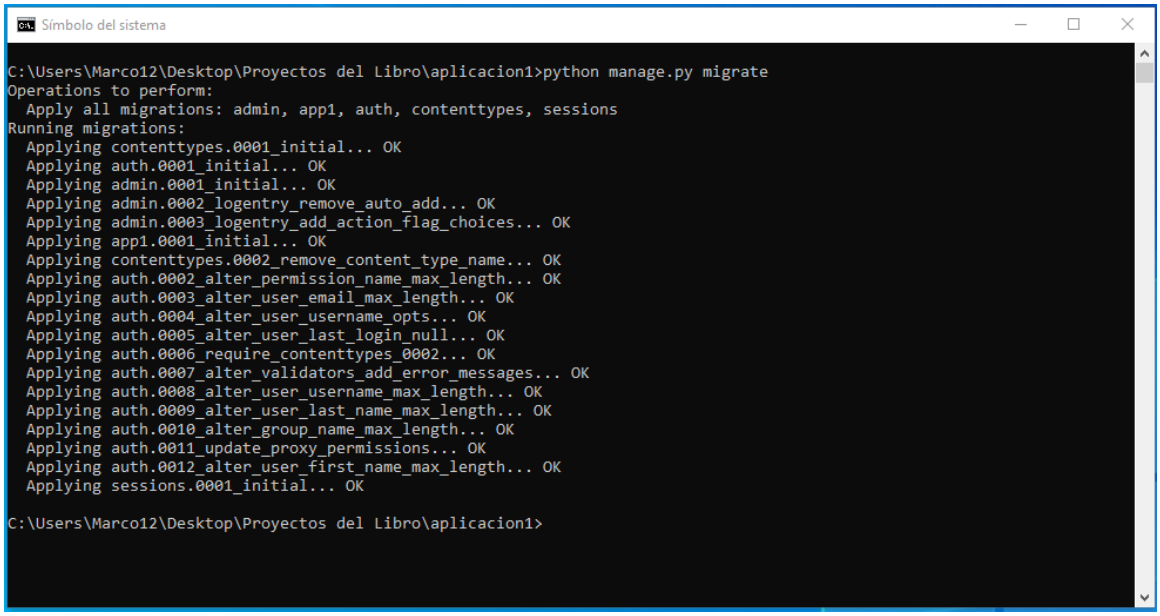


No usamos el comando anterior si hay algún cambio funcional, es decir, operaciones que implican consultar la base de datos y otras operaciones que no afectan cómo se estructura o almacena la base de datos. Aunque tenemos que usar el comando *makemigrations* cuando los campos en el modelo cambian aunque sea ligeramente.

Comando Migrate

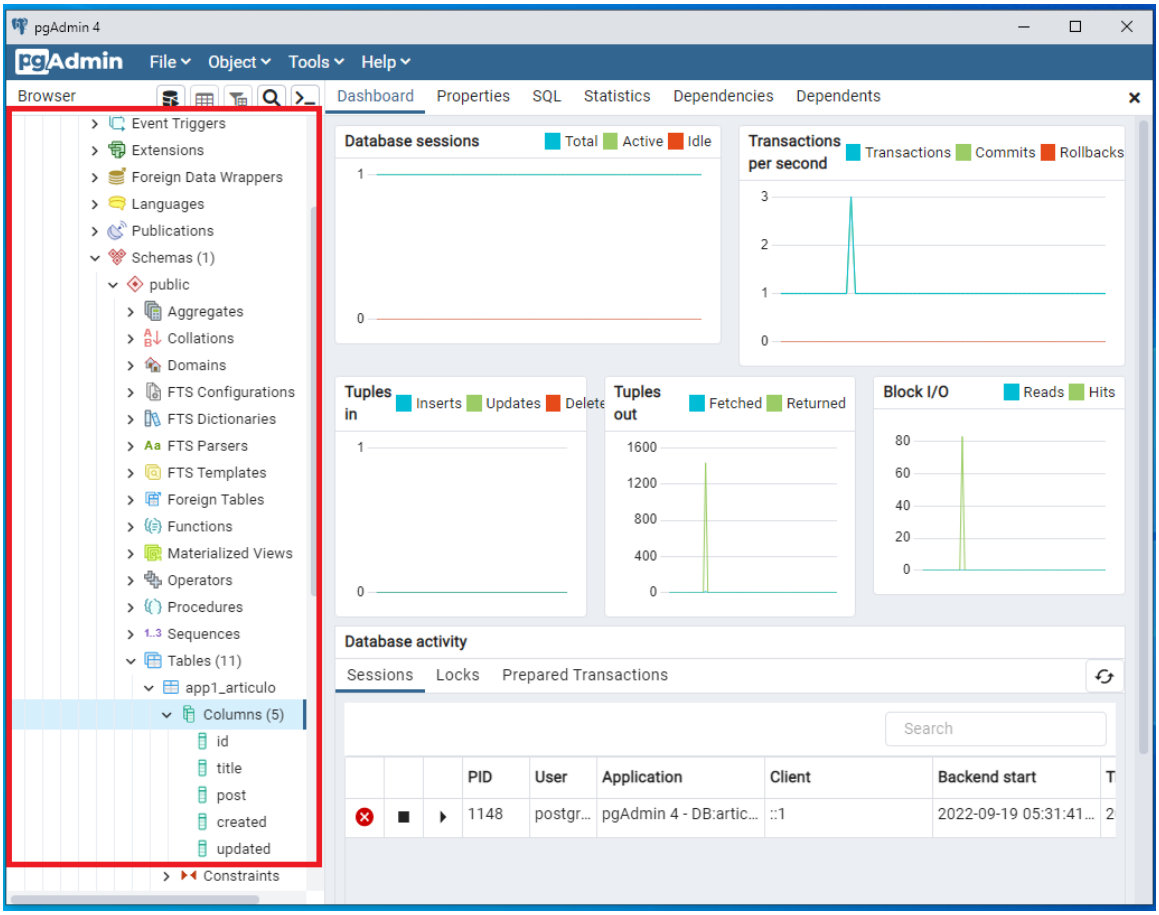
Para ver el resultado o crear las tablas, los atributos y las relaciones reales entre las tablas de la base de datos, debemos ejecutar el comando *migrate*, que verá el archivo más reciente en la carpeta *migration* y ejecutará las consultas para cambiar el esquema de la base de datos. Entonces, este es un comando muy poderoso que puede realizar consultas SQL con Python.

Ejecutamos el comando **python manage.py migrate**:



Para ver el resultado de las migraciones tenemos que ir al **pgAdmin 4**, del lado izquierdo tendremos una sección llamada **Browser**, en esa sección podremos ver la base de datos que habíamos creado anteriormente. En mi caso nombré a esa base de datos como **articulo**, si hacemos clic en la flecha se desplegará la estructura de la base de datos. Para ver la tabla que acabamos de migrar seguiremos la siguiente ruta, **Schemas-Table-app1_articulo-Columns**.

Si todo salió bien tendrás algo así en **pgAdmin 4**:



Conclusión

Entonces, en este capítulo del libro, pudimos configurar una base de datos, más específicamente una base de datos externa (no la base de datos SQLite predeterminada) en un proyecto de Django. El concepto de migraciones fue entendido y demostrado con ejemplos en vivo. Además, se utilizó el proceso de creación y diseño de modelos del capítulo anterior para crear la estructura en una base de datos real.

Sección de Administración

En el capítulo anterior, configuramos la base de datos en nuestro proyecto de Django. Avanzaremos e interactuaremos con la sección de administración en Django. Django tiene una sección de administración incorporada que podemos usar para administrar nuestras aplicaciones y modelos. Crearemos una cuenta de administrador (superusuario) e intentaremos representar nuestro modelo personalizado en la interfaz de administración..

¿Qué es la Sección de Administración?

La sección de administración es una potente utilidad integrada proporcionada por Django. Otorga los derechos administrativos sobre el proyecto web, la interfaz es ordenada y proporciona una funcionalidad lista para usar, para interactuar con los modelos en nuestro proyecto sin que nosotros creemos manualmente ningún mapeo de las vistas y direcciones URL. Está restringido solo a superusuario o usuarios de confianza para usarlo tal como está con fines administrativos.

La sección de administración está presente de forma predeterminada para cualquier aplicación de Django. La interfaz proporciona el modelo de usuario y grupo de forma predeterminada. Además, podemos tener nuestros propios modelos personalizados para interactuar. Para cada modelo registrado, tiene la funcionalidad **CRUD** (crear/leer/actualizar/eliminar) que hace que sea muy fácil y conveniente probar el funcionamiento del modelo antes de trabajar con las API o avanzar en el proyecto.

Configuración de una Cuenta de Administrador

Para acceder a la sección de administración, necesitamos crear un superusuario. Un superusuario, como su nombre indica, es un usuario que tiene la autoridad suprema para realizar operaciones en el proyecto, en este caso, una aplicación web. Para crear un superusuario necesitamos ejecutar un comando desde la línea de comandos que toma nuestro *nombre*, *correo electrónico* y *contraseña* como entrada para crear el superusuario.

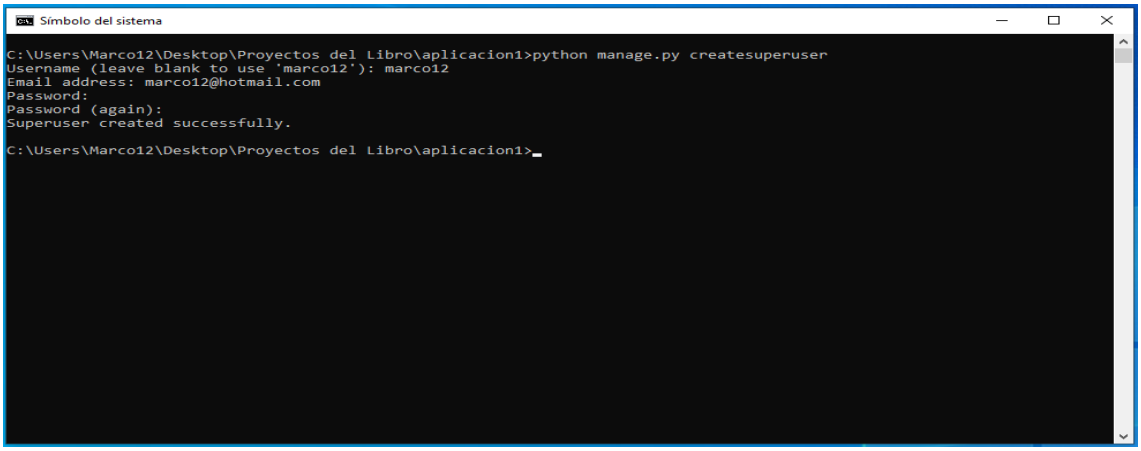
Comando para crear un superusuario: **python manage.py createsuperuser**

Esto le pedirá un par de cosas como:

- *Username*: el valor predeterminado es *admin*.
- *Email*: no es necesario poner uno.
- *Password*: debe tener al menos ocho caracteres.

La entrada de contraseña será silenciosa, lo que significa que no puede ver lo que escribe por razones de seguridad, y el campo de contraseña se confirmará una vez, por lo que deberá ingresar la contraseña una vez más. Pero eso es todo lo que tienes que hacer para crear un superusuario en Django.

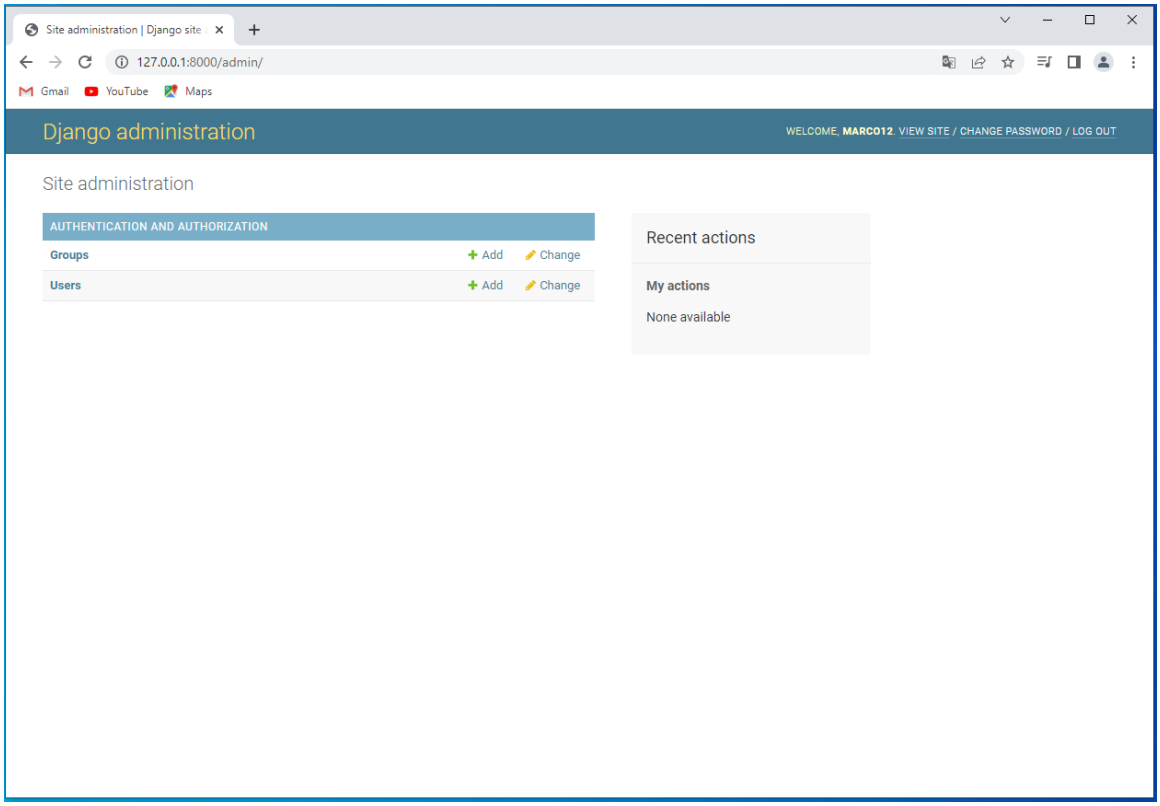
Creación de superusuario:



Navegando por la Sección de Administración

Después de crear un superusuario, ahora podemos navegar por la sección de administración desde el navegador. La sección de administración está ubicada de forma predeterminada en la URL */admin*, es decir, debe navegar a *http://127.0.0.1:8000/admin*.

Después de visitar la ruta de administración, se le indicará que inicie sesión. Simplemente necesita agregar el nombre de usuario y la contraseña que ingresó al crear el superusuario hace un momento, y debería estar en la **Sección de administración**. La sección de administración predeterminada a partir de **Django 4.1** tiene el siguiente aspecto:



Componentes de la Sección de Administración

La interfaz de administración parece bastante simple, pero es bastante potente y personalizable. Tenemos los modelos predeterminados en Django que están etiquetados en la sección *Authentication and Authorization*, es decir, *Users* y *Groups*. Puede ver que tenemos los enlaces + *Add* y *Change* para crear los datos asociados con esos modelos. En la sección de administración, básicamente puede jugar con sus modelos, no fue diseñado para actuar como interfaz para su aplicación, eso es lo que dice la documentación de Django y es absolutamente correcto.

Por lo tanto, no tenemos muchas cosas que explorar cuando se trata de la interfaz de usuario, ya que es simple y directa de entender.

Registrando Modelos en la Sección de Administración

Ahora, podemos pasar a registrar nuestros propios modelos en la sección de administración. Para hacer eso, dentro de la carpeta de la aplicación verá un archivo *admin.py*. Asegúrate de estar en la aplicación en la que has creado un modelo. Necesitamos registrar un modelo en la sección de administración.

El siguiente código en *admin.py*, agregará nuestro modelo a la sección de administración:

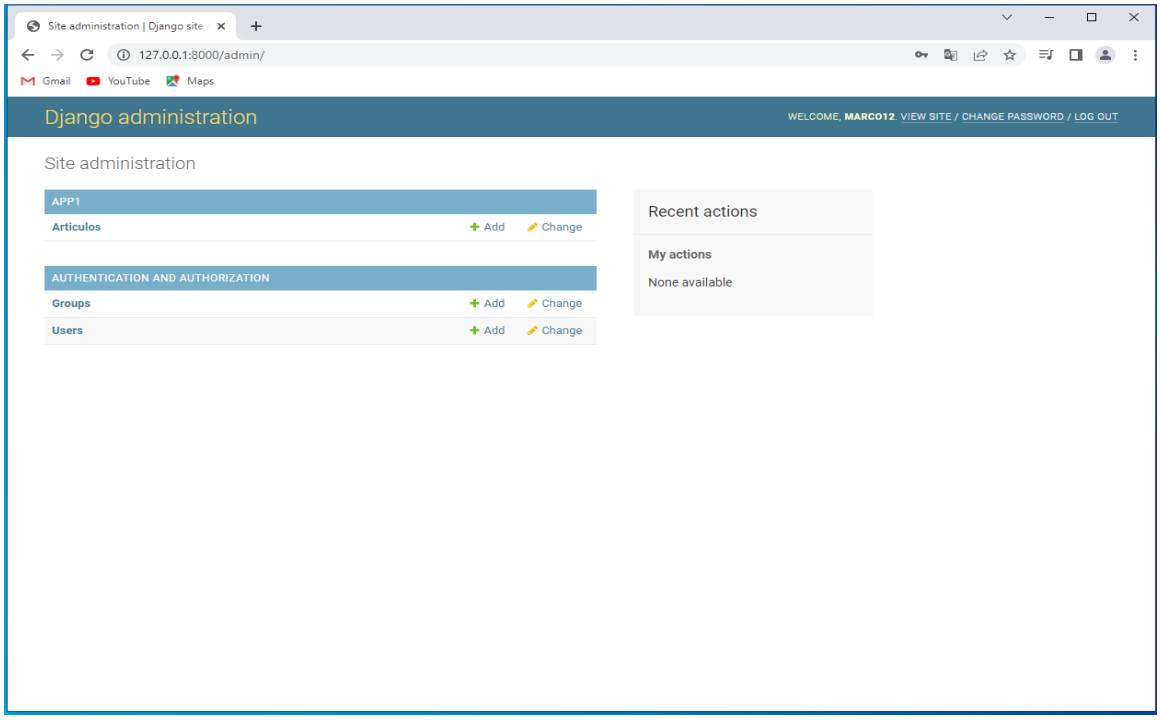
```
1 from django.contrib import admin
2 from .models import Articulo
3
4 admin.site.register(Articulo)
5
6 # Register your models here.
```

El *admin.site.register* básicamente agrega un modelo a la interfaz de administración.

El modelo *Artículo* se define de la siguiente manera:

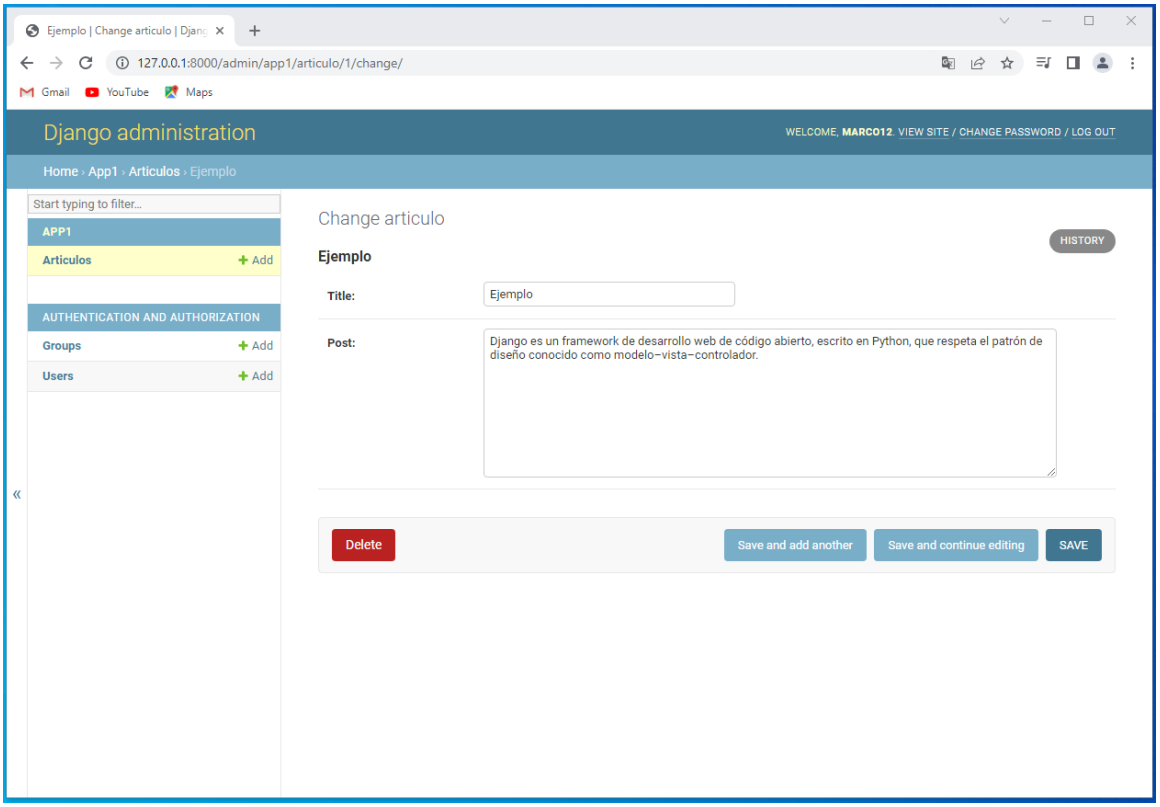
```python
from django.db import models

# Create your models here.

class Articulo(models.Model):
    title = models.CharField(max_length=255)
    post = models.TextField()
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)


    def __str__(self):
        return self.title
```

Esto agregará el modelo *Articulo* a la sección de administración. Ahora podemos realizar operaciones CRUD en la sección de administración.



Añadiendo un Articulo

Entonces, realice operaciones **CRUD**, podemos navegar al modelo *Articulo* y simplemente hacer clic en el botón *Add Article* para agregar un objeto al modelo. Se nos presentará un formulario para llenar. Ahora aquí podemos ver los campos que realmente debe ingresar el usuario. No vemos los campos *created* y *updated*, ya que se configuran automáticamente según la hora actual.



Ahora, podemos ver que también tenemos la capacidad de eliminar y actualizar la publicación. Esto es lo mejor de la interfaz de administración de Django. Es bastante intuitivo probar modelos complejos antes de seguir adelante y hacer las correcciones necesarias si es necesario.

Conclusión

Entonces, en este capítulo pudimos interactuar con la sección de administración en Django. Pudimos registrar nuestro modelo personalizado en la interfaz de administración y luego personalizar el formato de cómo se muestra en la interfaz.