**Configura tu entorno de desarrollo de manera profesional con Python y Django**

[#python](https://dev.to/t/python)[#django](https://dev.to/t/django)[#vscode](https://dev.to/t/vscode)

Configurar tu entorno de desarrollo de manera profesional te ayudará a ahorrarte mucho tiempo, y en esta ocasión veremos como configurar tu entorno de desarrollo para que trabajes con Python y Django desde Visual Studio Code, el editor de texto favorito para muchos.

**¿Qué es Django?**

Django es un framework de Python para desarrollo web, que nos permite crear y mantener aplicaciones web de una manera rápida, segura y escalable.

**Requerimientos**

* [Git](https://git-scm.com/downloads)
* [Visual Studio Code](https://code.visualstudio.com/)
* Instalar la [extensión de Python](https://marketplace.visualstudio.com/items?itemName=ms-python.python) para VS Code
* [Python 3.6 o superior](https://www.python.org/downloads/)

**Configura un repositorio de Git en tu proyecto**

Git es una herramienta esencial para cualquier proyecto de software, que permite compartir código, participar en un proyecto open source o manejar las versiones de tu aplicación.

Es por ello que te invito a usar Git en tus proyectos, y en este no será la excepción, vamos a inicializar un repositorio de Git, creando primero una carpeta para el proyecto:

* En tu sistema de archivos, crea una carpeta para tu proyecto, puedes llamarla como tu quieras, en este ejemplo la llamaremos proyecto\_django.
* Abre tu consola desde la ubicación de tu carpeta proyecto\_django y ejecuta el siguiente comando para inicializar un repositorio de Git:

git init

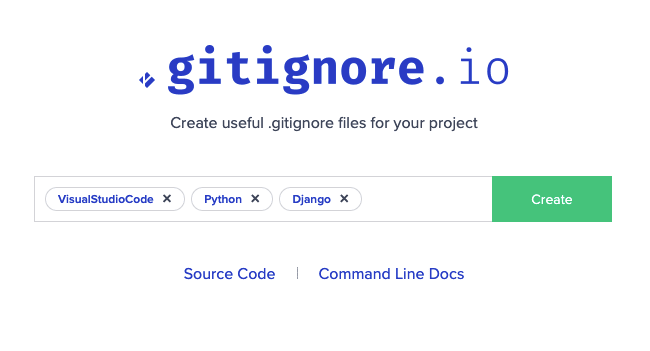
El siguiente paso es crear un archivo .gitignore, que permitirá ignorar archivos que no queremos darle un seguimiento a través del sistema control de versiones, estos pueden ser archivos de configuración, que contienen datos sensibles como contraseñas, archivos binarios, como imágenes, ya que suelen ser muy pesados, archivos de compilación de Python (.pyc), etc.

* Abre la carpeta de tu proyecto desde Visual Studio Code, ya sea ejecutando el comando code . desde tu consola o abriendo Visual Studio Code y en la opción **File > Open Folder**.
* Crea el archivo .gitignore en la raíz de la carpeta del proyecto, y no olvides agregar el punto (.) al principio del nombre.

**¿Qué contendrá este archivo .gitignore?**

El archivo .gitignore contendrá una lista de todos los archivos que vamos a ignorar, y para crear esa lista nos apoyaremos de la página [https://gitignore.io](https://gitignore.io/) que nos facilitará la creación de este archivo pasándole un conjunto de etiquetas de todas las herramientas que utilizaremos.

Para este caso, pasamos las etiquetas VisualStudioCode, Python y Django:

[](https://res.cloudinary.com/practicaldev/image/fetch/s--khauNLdy--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/uploads/articles/7jckauc5ouzygp2dta1u.png)

Damos clic al botón **Create** y enseguida nos aparecerá todo un conjunto de archivos que son utilizados por estas herramientas, pero que no es necesario darles un seguimiento en nuestro repositorio. Copia el contenido y pégalo en tu archivo .gitignore.

**Crear un entorno virtual para nuestro proyecto de Django**

Un entorno virtual nos permitirá instalar todos los paquetes y librerías necesarias para tu proyecto de manera independiente, es decir, podremos probar distintas versiones de Django sin entrar en conflicto con otros proyectos. Piensa en un entorno virtual como una máquina virtual, capaz de aislar todos los paquetes instalados dentro de ese entorno virtual.

Ahora vayamos a la creación del entorno virtual:

* Dentro de tu carpeta principal del proyecto, desde la línea de comandos, ejecuta el siguiente comando para crear un entorno virtual llamado venv (asegurate de tener instalado Python):

# Linux

sudo apt-get install python3-venv # En caso necesario de no contar con python3-venv

python3 -m venv venv

# macOS

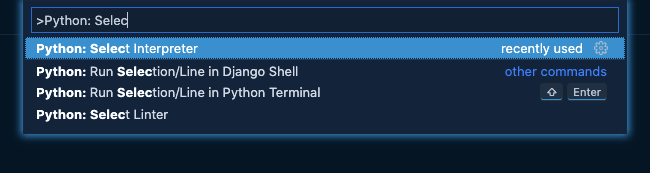
python3 -m venv venv

# Windows

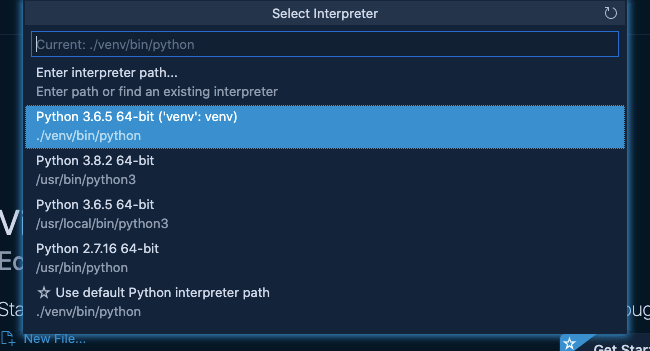
python -m venv venv

En Visual Studio Code tenemos que seleccionar el intérprete de Python que se utilizará para ejecutar nuestro proyecto. Ahí es donde entra el entorno virtual que acabamos de crear. Prácticamente le decimos a VS Code, quiero que utilices la versión de Python que tengo dentro de mi entorno virtual junto con todos sus paquetes instalados.

* Dentro de VS Code, vamos a la opción **View > Command Palette** y seleccionamos el comando **Python: Select Interpreter**

[](https://res.cloudinary.com/practicaldev/image/fetch/s--qy0Z-lnZ--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/uploads/articles/pgjhixzu334zc3qgblrn.png)

* El comando mostrará una lista de los intérpretes disponibles que VS Code puede localizar automáticamente. Desde la lista, selecciona el entorno virtual que se encuentra dentro de la carpeta del proyecto:

[](https://res.cloudinary.com/practicaldev/image/fetch/s--64DLMxpE--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/uploads/articles/feack7joxn6k1l8crui3.png)

Hasta este momento solamente hemos creado nuestro entorno virtual, pero aún no lo usamos, para ello tenemos que activarlo.

* Y aquí viene la magia de VS Code, vamos a abrir una línea de comandos desde la opción **Terminal > New Terminal**, esto abre una consola integrada en nuestro editor y automáticamente activa el entorno virtual ejecutando el script de activación.
* Del lado izquierdo visualizarás la barra de estado de VS Code indicando que estamos utilizando un entorno virtual.

[Entorno virtual activado](https://res.cloudinary.com/practicaldev/image/fetch/s--zOlRtRLw--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/uploads/articles/ptu8qw3j7pkw6p04clar.png)

**Actualizando y utilizando pip**

Por defecto Python ya cuenta con un conjunto de funcionalidades integradas listas para usar, que se encuentran agrupadas en módulos. De seguro has usado el módulo random para generar números aleatorios, pero que pasa con Django.

Ya que Django es un paquete que no se encuentra directamente dentro de Python tenemos que instalarlo de manera externa.

**¿Cómo instalamos Django?**

Para instalar Django necesitamos de un gestor de dependencias o en otras palabras de un instalador de paquetes, y el más popular en Python es **PIP** (Package Installer for Python) ya que viene integrado junto con Python en el momento de la instalación.

Si vienes de JavaScript de seguro conoces **npm** y se te hará muy familiar los comandos a la hora de instalar paquetes con **pip**.

* Un paso adicional antes de empezar a instalar paquetes, es tener actualizado **pip** en nuestro entorno virtual, que si recuerdas ya se encuentra activado en la consola de VS Code, ejecuta el siguiente comando:

pip install --upgrade pip

* Ahora procederemos a instalar Django en el entorno virtual:

pip install django

* Puedes verificar que Django fue instalado con el siguiente comando:

pip freeze

**Crear un proyecto de Django**

Ahora que ya tenemos instalado Django tenemos disponible el comando django-admin que nos permitirá crear nuestro primer proyecto.

Vamos a la consola de VS Code con tu entorno virtual activado y ejecuta el siguiente comando:

django-admin startproject proyecto\_django .

El punto (.) al final del comando indica que el proyecto será creado en la carpeta actual.

**Creando archivos de configuración para cada entorno**

En un entorno profesional de trabajo es ideal manejar un servidor diferente por cada etapa del proyecto, es decir, cuando realizamos cambios al código del proyecto, usamos un entorno de desarrollo, cuando el cliente prueba funcionalidades nuevas para aprobarlas, se usa un servidor de staging, y finalmente cuando la aplicación es accesible para todos, se usa un servidor de producción.

Es por ello importante manejar archivos de configuración diferentes por cada entorno, ya que esas configuraciones serán diferentes para cada servidor, por lo que sería tedioso manejar un solo archivo y estar cambiando a cada rato cuando nos cambiamos de entorno.

Django por defecto maneja toda la configuración del proyecto en un solo archivo (proyecto\_django/settings.py), pero ahora queremos manejar múltiples archivos de configuración por cada entorno manejado.

Para ello vamos a agrupar todos esos archivos de configuración en un sola carpeta llamada settings, creamos la carpeta dentro de la subcarpeta proyecto\_django y creamos la siguiente estructura de archivos:

proyecto\_django/settings

├── \_\_init\_\_.py

├── base.py

├── local.py

└── production.py

Vamos a describir cada archivo de la carpeta settings:

* \_\_init\_\_.py: Un archivo vacío que le dice a Python que esta carpeta se trata de un paquete.
* base.py: Todas las configuraciones en común entre entornos.
* local.py: Contiene todas las configuraciones de tu entorno local (que incluyen las de base.py).
* production.py: Configuraciones para el entorno de producción.

**¿Qué contenido tendrá base.py?**

Todas las configuraciones que no cambien entre los distintos entornos manejados, estará dentro del archivo base.py. Como recomendación copia el contenido del archivo settings.py y pegalo en base.py luego ve identificando que configuraciones son distintas entre tus diferentes entornos.

**Identificando configuraciones que cambien entre entornos**

La primera configuración que podemos identificar es la siguiente:

# SECURITY WARNING: don't run with debug turned on in production!

DEBUG = True

ALLOWED\_HOSTS = []

En un entorno local tener el modo DEBUG activado te facilitará visualizar una página detallada de errores a la hora de codear. En un entorno de producción no queremos mostrarle a nuestros usuarios esos errores detallados, ya que contiene datos sensibles del servidor, por lo que estaríamos expuestos a detalles de seguridad.

La variable de configuración ALLOWED\_HOSTS es una lista de hosts/nombres de dominio que nuestro sitio de Django puede servir. Si la variable DEBUG es True y ALLOWED\_HOSTS es una lista vacía se proporcionará acceso solamente a ['.localhost', '127.0.0.1', '[::1]'].

Por el momento copiamos esas dos variables de configuración y las pegamos en local.py y production.py, por ahora solamente cambiamos en el archivo production.py la variable DEBUG en False. Ya podemos eliminar esas dos variables de configuración en el archivo base.py.

local.py

# SECURITY WARNING: don't run with debug turned on in production!

DEBUG = True

ALLOWED\_HOSTS = []

production.py

# SECURITY WARNING: don't run with debug turned on in production!

DEBUG = False

ALLOWED\_HOSTS = []

Ahora tenemos la variable de configuración DATABASES que contiene las configuraciones para todas las bases de datos usadas con Django:

# Database

# https://docs.djangoproject.com/en/3.2/ref/settings/#databases

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.sqlite3',

'NAME': BASE\_DIR / 'db.sqlite3',

}

}

En un entorno local nos interesa manejar datos de prueba para verificar el funcionamiento de la aplicación, Django trae por defecto la configuración con una base de datos SQLite, que no es más que un simple archivo donde son guardados los datos.

Para el entorno de producción, Django soporta múltiples motores de base de datos para usar, como MariaDB, MySQL, Oracle o PostgreSQL. El siguiente es un ejemplo para PostgreSQL (Recuerda tener instalado el driver correspondiente con el motor de base de datos):

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.postgresql',

'NAME': 'mydatabase',

'USER': 'mydatabaseuser',

'PASSWORD': 'mypassword',

'HOST': '127.0.0.1',

'PORT': '5432',

}

}

Copia la variable de configuración DATABASES y pegalo en tu archivo local.py, para el caso de production.py puedes utilizar por ahora el ejemplo para PostgreSQL. Ya puedes eliminar la configuración para la variable DATABASES en el archivo base.py.

Por último tenemos la variable de configuración STATIC\_URL que es la URL a usar para hacer referencia a los archivos estáticos, puede tener diferentes valores dependiendo del entorno, pero por ahora usaremos el mismo valor para los archivos local.py y production.py.

Ya para finalizar tenemos que importar todas las variables de configuración del archivo base.py a local.py y production.py:

from .base import \*

Nuestros archivos de configuración se verán de la siguiente manera:

base.py

"""

Django settings for proyecto\_django project.

Generated by 'django-admin startproject' using Django 3.2.6.

For more information on this file, see

https://docs.djangoproject.com/en/3.2/topics/settings/

For the full list of settings and their values, see

https://docs.djangoproject.com/en/3.2/ref/settings/

"""

from pathlib import Path

# Build paths inside the project like this: BASE\_DIR / 'subdir'.

BASE\_DIR = Path(\_\_file\_\_).resolve().parent.parent

# Quick-start development settings - unsuitable for production

# See https://docs.djangoproject.com/en/3.2/howto/deployment/checklist/

# SECURITY WARNING: keep the secret key used in production secret!

SECRET\_KEY = 'my-secret-key'

# Application definition

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

]

MIDDLEWARE = [

'django.middleware.security.SecurityMiddleware',

'django.contrib.sessions.middleware.SessionMiddleware',

'django.middleware.common.CommonMiddleware',

'django.middleware.csrf.CsrfViewMiddleware',

'django.contrib.auth.middleware.AuthenticationMiddleware',

'django.contrib.messages.middleware.MessageMiddleware',

'django.middleware.clickjacking.XFrameOptionsMiddleware',

]

ROOT\_URLCONF = 'proyecto\_django.urls'

TEMPLATES = [

{

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

'DIRS': [],

'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',

],

},

},

]

WSGI\_APPLICATION = 'proyecto\_django.wsgi.application'

# Password validation

# https://docs.djangoproject.com/en/3.2/ref/settings/#auth-password-validators

AUTH\_PASSWORD\_VALIDATORS = [

{

'NAME': 'django.contrib.auth.password\_validation.UserAttributeSimilarityValidator',

},

{

'NAME': 'django.contrib.auth.password\_validation.MinimumLengthValidator',

},

{

'NAME': 'django.contrib.auth.password\_validation.CommonPasswordValidator',

},

{

'NAME': 'django.contrib.auth.password\_validation.NumericPasswordValidator',

},

]

# Internationalization

# https://docs.djangoproject.com/en/3.2/topics/i18n/

LANGUAGE\_CODE = 'en-us'

TIME\_ZONE = 'UTC'

USE\_I18N = True

USE\_L10N = True

USE\_TZ = True

# Default primary key field type

# https://docs.djangoproject.com/en/3.2/ref/settings/#default-auto-field

DEFAULT\_AUTO\_FIELD = 'django.db.models.BigAutoField'

local.py

from .base import \*

# SECURITY WARNING: don't run with debug turned on in production!

DEBUG = True

ALLOWED\_HOSTS = []

# Database

# https://docs.djangoproject.com/en/3.2/ref/settings/#databases

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.sqlite3',

'NAME': BASE\_DIR / 'db.sqlite3',

}

}

# Static files (CSS, JavaScript, Images)

# https://docs.djangoproject.com/en/3.2/howto/static-files/

STATIC\_URL = '/static/'

production.py

from .base import \*

# SECURITY WARNING: don't run with debug turned on in production!

DEBUG = False

ALLOWED\_HOSTS = []

# Database

# https://docs.djangoproject.com/en/3.2/ref/settings/#databases

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.postgresql',

'NAME': 'mydatabase',

'USER': 'mydatabaseuser',

'PASSWORD': 'mypassword',

'HOST': '127.0.0.1',

'PORT': '5432',

}

}

# Static files (CSS, JavaScript, Images)

# https://docs.djangoproject.com/en/3.2/howto/static-files/

STATIC\_URL = '/static/'

**Modularizando la variable INSTALLED\_APPS**

Dentro del archivo base.py te encontrarás con la variable de configuración INSTALLED\_APPS que es una lista de todas las aplicaciones que están habilitadas para el proyecto. Django maneja el concepto de **aplicación** como un conjunto de funcionalidades únicas y que se pueden reutilizar en otros proyectos.

Si miras la lista verás que un proyecto de Django trae por defecto todo un conjunto de funcionalidades listas para usar, como el sitio de administración de Django, el sistema de autenticación, el manejo de sesiones, etc.

Todos estos paquetes pertenecen a Django, pero sorpresa, podemos instalar paquetes de terceros, o mejor aún crear nuestros propios paquetes. Una buena práctica es separar estos diferentes tipos de paquetes en distintas variables.

Para los paquetes de Django declaramos la variable DJANGO\_APPS, para los paquetes de terceros la variable THIRD\_PARTY\_APPS y para paquetes propios la variable LOCAL\_APPS, las variables se verán de la siguiente manera:

# Application definition

DJANGO\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

]

THIRD\_PARTY\_APPS = []

LOCAL\_APPS = []

INSTALLED\_APPS = DJANGO\_APPS + THIRD\_PARTY\_APPS + LOCAL\_APPS

**Actualiza el valor por defecto de DJANGO\_SETTINGS\_MODULE**

Ahora que cambiamos el archivo de configuración settings.py que trae Django por defecto a múltiples archivos de configuración por cada entorno, tenemos que actualizar los archivos asgi.py, wsgi.py y manage.py.

Cambiamos

os.environ.setdefault('DJANGO\_SETTINGS\_MODULE', 'proyecto\_django.settings')

por

os.environ.setdefault('DJANGO\_SETTINGS\_MODULE', 'proyecto\_django.settings.local')

Esto quiere decir que ahora Django utilizará el archivo de configuración local.py.

Para un entorno de producción tienes que configurar la variable de entorno DJANGO\_SETTINGS\_MODULE para apuntar hacia el archivo de configuración proyecto\_django.settings.production

**Creando la base de datos de desarrollo**

Ya que se trata de un proyecto inicial de Django, por ahora no tenemos ningún modelo creado, pero tenemos los modelos que trae Django por defecto, ejecuta el siguiente comando para crear una base de datos de desarrollo:

python manage.py migrate

De inmediato visualizarás dentro de la subcarpeta proyecto\_django un archivo db.sqlite3 en donde se guardarán los datos para el entorno local.

**Mantener algunas configuraciones de manera secreta**

En algunas variables de configuración tendrás valores que no vas a querer compartir o subir a tu repositorio por razones de seguridad o porque simplemente solo funcionarán para tu entorno de desarrollo, estas variables pueden ser SECRET\_KEY que no debemos usar la misma para producción y para desarrollo, DATABASES si estas manejando de manera local una base de datos PostgreSQL, en donde tendrás que especificar nombre de usuario, contraseña del usuario, nombre de la base de datos, puerto, etc.

Estas son solo algunas variables que pueden cambiar sus valores y necesitas mantenerlas secretas, es decir, no darle seguimiento a través del sistema control de versiones, o en caso de que otras personas están colaborando en el proyecto y van a configurar su propio entorno de desarrollo.

Para ello vamos a instalar una librería con **pip** llamada **Python Decouple** que nos ayudará a separar esos valores en un archivo .env, ejecuta el siguiente comando en tu consola:

pip install python-decouple

Ahora crea un archivo .env en la raíz de tu proyecto con el siguiente contenido:

SECRET\_KEY=my-secret-key

DB\_NAME=django

DB\_USER=root

DB\_PASSWORD=

DB\_HOST=127.0.0.1

DB\_PORT=5432

El archivo .env no se agregará al historial de cambios en el sistema control de versiones, ya que en un paso anterior lo hemos agregado en el .gitignore

Necesitamos importar la librería en el archivo base.py:

from decouple import config

Cambia el valor de SECRET\_KEY por lo siguiente:

SECRET\_KEY = config('SECRET\_KEY')

En caso de que quieras manejar un motor de base de datos diferente en tu entorno local, en lugar del archivo de SQLite que maneja Django por defecto, tendrás que configurar las variables correspondientes en tu archivo local.py:

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.postgresql',

'NAME': config('DB\_NAME'),

'USER': config('DB\_USER'),

'PASSWORD': config('DB\_PASSWORD'),

'HOST': config('DB\_HOST'),

'PORT': config('DB\_PORT'),

}

}

Recuerda cambiar a los valores correctos según tu entorno en el archivo .env.

**Convirtiendo a otros tipos de datos**

Todos los valores que se leen del archivo .env son manejados como una cadena de texto, pero en caso de que quieras configurar una variable como EMAIL\_PORT, manejada por Django para el envió de correos, esta debe ser configurada como un entero, por lo que podemos pasarle el argumento cast a la función config() para transformar el valor de cadena como un número entero:

config('EMAIL\_PORT', cast=int)

**Manejando valores por defecto**

Un argumento extra que le podemos pasar a la función config() es default, que definirá un valor por defecto en caso de que la variable de entorno no este definida en el archivo .env:

config('EMAIL\_PORT', default=25, cast=int)

Ahora no es necesario que la variable de entorno EMAIL\_PORT este definida en el archivo .env

**Compartir las variables de entorno definidas para el proyecto**

Seguramente te preguntarás porque vamos a compartir algo que al principio mencionamos que mantendríamos en secreto, pues no precisamente vamos a compartir los valores, sino, más bien las variables que necesita el proyecto para funcionar correctamente.

Si una persona se une al proyecto no sabrá como configurar y ejecutar la aplicación de manera correcta, por lo que debemos dejar un ejemplo de las variables que tiene que configurar. Para ello vamos a crear un archivo .env.example en la raíz del proyecto que contendrá datos de ejemplo, esto servirá como referencia para crear el archivo .env con los valores del entorno apropiado.

El archivo .env.example si se agregará al repositorio, ya que solo contendrá datos de ejemplo con las variables de entorno para crear el archivo .env, y recuerda cambiar los valores para que tu entorno funcione correctamente.

**Creando nuestra primera app**

Recuerdas que te había mencionado que podíamos crear nuestros propios paquetes, pues hemos llegado a ese momento.

Si ejecutamos el siguiente comando, Django va a crear una app en la raíz del proyecto:

python manage.py startapp home

Aunque esto es correcto, si el proyecto va creciendo y se van creando más apps, se tendrá en la raíz del proyecto una lista interminable de carpetas.

Entonces vamos a aplicar desde el inicio del proyecto buenas prácticas. Crea en la raíz del proyecto una carpeta llamada apps, aquí englobaremos todas las apps creadas.

Si ejecutaste el comando anterior recuerda eliminar la carpeta home de la raíz del proyecto.

Desde tu línea de comandos en VS Code y con tu entorno virtual activado, ingresa a la carpeta apps y ejecuta el siguiente comando:

cd apps

django-admin startapp home

Visualizarás que dentro de la carpeta apps se ha creado una subcarpeta home con la siguiente estructura:

apps

└── home

├── \_\_init\_\_.py

├── admin.py

├── apps.py

├── migrations

│   └── \_\_init\_\_.py

├── models.py

├── tests.py

└── views.py

Vamos a explicar rápidamente cada archivo:

* views.py: Contiene la lógica o funcionalidad de la aplicación.
* models.py: Contiene un conjunto de clases que definen la estructura de datos a utilizar por la aplicación.
* migrations: Una carpeta que es usada por Django para manejar las versiones de la base de datos.
* apps.py: Configuraciones propias de la aplicación
* admin.py: Creación de la interfaz administrativa para el manejo de modelos.
* tests.py: Creación de pruebas.

**Creando nuestra primera vista**

Modifica el archivo home/views.py para crear una vista que nos retorne un saludo como respuesta:

from django.http import HttpResponse

def index(request):

return HttpResponse("Hello, Django!")

Ahora crea un archivo home/urls.py para definir las diferentes rutas asociadas a las vistas:

from django.urls import path

from . import views

urlpatterns = [

path("", views.index, name="home"),

]

Dentro de la subcarpeta proyecto\_django hay un archivo urls.py que es el encargado de manejar el enrutamiento dentro de la aplicación. Una buena práctica es crear un archivo urls.py dentro de cada app, con la idea de modularizar las rutas de cada aplicación.

En el archivo proyecto\_django/urls.py tienen que incluirse las rutas de cada app:

from django.contrib import admin

from django.urls import include, path

urlpatterns = [

path("", include("apps.home.urls")),

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

]

**Debugging con Django**

Para probar nuestra aplicación simplemente ejecutamos el siguiente comando:

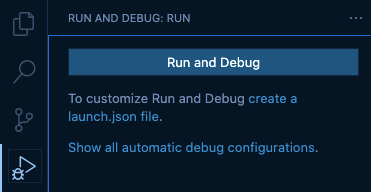
python manage.py runserver

Abre un navegador y visita http://127.0.0.1:8000 para ver una página que renderiza "Hello, Django!".

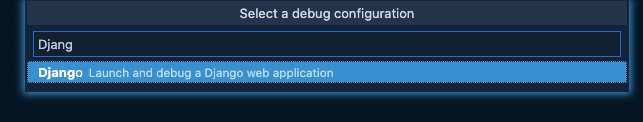
Seguramente te estarás preguntando si existe una manera más fácil de ejecutar el servidor de desarrollo, y has llegado a la parte indicada.

Vamos a crear algo genial en VS Code que ayudará tanto a correr la aplicación como para realizar debugging.

Cambia el modo **Run and Debug** en el lado izquierdo de tu barra de actividad y verás el mensaje "To customize Run and Debug create a launch.json file", si das clic en el enlace **create a launch.json file** esto va a crear un archivo launch.json que contendrá las configuraciones de depuración.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--QBAlhp6Z--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/uploads/articles/3fn8lko30kk4sxyc2f3k.png)

Selecciona el enlace y VS Code te solicitará una configuración para depuración. Selecciona **Django** y automáticamente VS Code crea el archivo launch.json con las configuraciones para ejecutar la aplicación de Django.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--DDZejts0--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/uploads/articles/vl1no5asdb7srl3zo6ao.png)

Visualiza la configuración que se genero:

{

// Use IntelliSense to learn about possible attributes.

// Hover to view descriptions of existing attributes.

// For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387

"version": "0.2.0",

"configurations": [

{

"name": "Python: Django",

"type": "python",

"request": "launch",

"program": "${workspaceFolder}/manage.py",

"args": [

"runserver"

],

"django": true

}

]

}

Prácticamente esta configuración dice que ejecute "${workspaceFolder}/manage.py" con los argumentos en la lista args.

Vamos ahora a probar la configuración, selecciona la opción **Run > Start Debugging**, o selecciona la flecha verde para ejecutar el servidor de desarrollo:

[Start Debugging Django](https://res.cloudinary.com/practicaldev/image/fetch/s--w6UV0xmE--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/uploads/articles/od05aubjlpw6y8vems9n.png)

Abre el navegador, visita http://127.0.0.1:8000 y verifica que la aplicación se ejecuta correctamente. De ahora en adelante puedes usar **Run > Start Debugging** en cualquier momento para probar la aplicación.