Django 2023

Diego Saavedra

Dec 2, 2023

Table of contents

1 Curso de Django **6**
- 1.1 Bienvenida . . . 6
- 1.2 ¿Qué es este Curso? . . . 6
- 1.3 ¿A quién está dirigido? . . . 6
- 1.4 ¿Cómo contribuir? . . . 7

I Git y Github **8**

2 Módulo 0: Git/Github **9**
- 2.1 Introducción a Git y Control de Versiones. . . . 9
- 2.2 Configuración de un Repositorio en GitHub . . . 9
- 2.3 Uso Básico de Comandos de Git. . . . 9
- 2.4 Colaboración en un Proyecto Utilizando GitHub . . . 10

3 Ejercicio práctico: **11**
- 3.1 Configurar un Repositorio y Realizar Cambios . . . 11
- 3.2 Resolución del Ejercicio Práctico . . . 11

II Introducción a Django **13**

4 Módulo 1: Introducción a Django. **14**
- 4.1 ¿Qué es Django y por qué utilizarlo? . . . 14
- 4.2 MVC vs MTV. . . . 14
- 4.3 Instalación de Django 4.2.3 y Configuración del Entorno de Desarrollo . . . 14
- 4.4 Creación de un Proyecto en Django. . . . 15
- 4.5 Estructura de directorios generada: . . . 15
- 4.6 Estructura de Directorios de un Proyecto Django. . . . 15
- 4.7 Creación de una Aplicación en Django . . . 16

5 Ejemplo Práctico: **18**
- 5.1 Creación de un Proyecto de E-commerce en Django . . . 18

6 Actividad Práctica. **20**
- 6.1 Creación de Un Blog . . . 20
- 6.2 Resolución de la Actividad Práctica: . . . 20

III Modelos y Bases de Datos **23**

7 Módulo 2: Modelos y Bases de Datos. **24**
7.1 Diseño de Modelos en Django . . . 24
7.2 1. Definimos el Modelo. . . . 24
7.2.1 Creación de un Modelo. . . . 25
7.3 2. Definir Métodos del Modelo. . . . 25
7.4 3. Aplicar Migraciones. . . . 26
7.5 Ejemplo de las clases Publicación y Comentario en Django: . . . 26
7.6 Migraciones de la Base de Datos . . . 28
7.7 Consultas a la Base de Datos con el ORM de Django. . . . 29
7.8 Ejemplo de Consulta con el ORM de Django . . . 29
7.9 Diagrama de Consulta con ORM de Django. . . . 30
7.10 Relaciones entre Modelos. . . . 31
7.11 Clave Primaria, Clave Foránea y Relaciones entre Modelos en Django. . . . 31
7.12 Clave Primaria. . . . 32
7.13 Clave Foránea. . . . 32
7.14 Relaciones entre Modelos de Django: . . . 33
7.15 Ejemplo de Conexión del Proyecto Blog con Bases de Datos . . . 34
7.16 **SQLite**: . . . 34
7.17 **MySQL**: . . . 34
7.18 **PostgreSQL:** . . . 37
7.19 **MongoDB:** . . . 40

8 Ejemplo Práctico: **44**
8.1 Diseño de un Modelo de Usuarios y Publicaciones en un Blog. . . . 44
8.1.1 Importación de módulos necesarios: . . . 44
8.1.2 Definición del modelo “Usuario”: . . . 44
8.1.3 Definición del modelo “Publicacion”: . . . 45

9 Actividad Práctica: **47**
9.1 Agrega un Campo Adicional al Modelo de Publicaciones . . . 47
9.2 Resolución de la Actividad Práctica. . . . 47
9.3 Extra. . . . 48
9.4 Conclusión. . . . 48

IV Vistas y Plantillas **50**

10 Módulo 3: Vistas y Plantillas. **51**
10.1 Creación de Vistas en Django . . . 51
10.2 Otra forma de generar un Hola Mundo en Django. . . . 52
10.3 Mostrar Publicaciones y Comentarios . . . 53
10.4 Crea una vista para mostrar las publicaciones: . . . 53
10.5 Sistema de Plantillas de Django (Jinja2) . . . 55
10.6 Conceptos Principales del Sistema de Plantillas de Django: . . . 56
10.7 Pasos para Utilizar el Sistema de Plantillas de Django en el Proyecto “blog”: 56

11 CRUD de Publicaciones **59**
11.1 Crear Publicaciones . . . 59
11.2 Leer Publicaciones . . . 60
11.3 Actualizar Publicaciones . . . 61
11.4 Eliminar Publicaciones . . . 62
11.5 CRUD de Comentarios . . . 64
11.6 Leer Comentarios . . . 65
11.7 Actualizar Comentarios . . . 66
11.8 Eliminar Comentarios . . . 68
11.9 Migrar y Ejecutar el Servidor . . . 69

12 Correcciones **70**
12.1 Integración de Botones. . . . 70
12.2 Para las Publicaciones: . . . 70
12.3 Para los Comentarios: . . . 72
12.4 Corregir el Modelo . . . 76
12.5 Backend: Construcción de una API con Django Rest Framework . . . 77
12.6 Documentación de las API . . . 79

13 FRONTEND: Herencia de Plantillas con Django. **81**
13.1 Herencia de Plantillas . . . 81
13.2 Modificación de Views. . . . 83
13.3 Bootstrap en base.html . . . 84
13.4 Agregando bootstrap a los templates. . . . 85

14 Reactjs **88**
14.1 Frontend con Reactjs . . . 88
14.1.1 1. index.css . . . 94
14.1.2 2. App.css . . . 95
14.1.3 3. Importar los archivos CSS en App.js . . . 95
14.2 Inconvenientes . . . 97

V Docker **98**

15 Módulo 5:Docker. **99**
15.1 Fundamentos de Docker. . . . 99
15.1.1 ¿Qué es Docker? . . . 99
15.1.2 ¿Por qué utilizar Docker? . . . 99
15.2 Problemas que están presentes en el Desarrollo de Software. . . . 100
15.3 Maquinas Virtuales vs Docker. . . . 102
15.3.1 Virtualización . . . 102
15.3.2 Máquinas Virtuales. . . . 102
15.3.3 Problemas de las VMs . . . 103
15.3.4 Contenedores. . . . 103
15.3.5 Containerización. . . . 105
15.4 Arquitectura de Docker. . . . 105
15.5 Trabajar con Docker. . . . 106
15.6 Docker desde Visual Studio Code. . . . 108

15.7 Introducción a Docker Compose 110
15.8 Ventajas de usar Docker Compose 110
15.8.1 Cómo usar Docker Compose en nuestro proyecto Django “Blog” . . 111

1 Curso de Django

1.1 Bienvenida

¡Bienvenidos al Curso Completo de Django! Este libro te guiará a través de un viaje desde los fundamentos hasta la creación de un blog funcional utilizando el framework de desarrollo web Django en Python.

1.2 ¿Qué es este Curso?

Este curso detallado te llevará desde los conceptos básicos de Django hasta la implementación práctica de un blog. A través de una combinación de teoría y proyectos prácticos, explorarás cada módulo diseñado para fortalecer tus habilidades en el desarrollo web con Django. Desde la configuración inicial hasta el despliegue en la nube con Azure, este curso te proporcionará las herramientas y conocimientos necesarios para crear aplicaciones web robustas.

1.3 ¿A quién está dirigido?

Este curso está diseñado tanto para aquellos que están dando sus primeros pasos en el desarrollo web como para aquellos que desean mejorar sus habilidades con Django. Si eres un estudiante, un profesional en busca de nuevas habilidades o simplemente alguien apasionado por la programación web, este curso te brindará la base necesaria para construir aplicaciones con Django.

1.4 ¿Cómo contribuir?

Valoramos tu participación en este curso. Si encuentras errores, deseas sugerir mejoras o agregar contenido adicional, ¡nos encantaría recibir tus contribuciones! Puedes contribuir a través de nuestra plataforma en línea, donde puedes compartir tus comentarios y sugerencias. Juntos, podemos mejorar continuamente este recurso educativo para beneficiar a la comunidad de desarrolladores Django.

Este libro ha sido creado con el objetivo de proporcionar acceso gratuito y universal al conocimiento de Django. Estará disponible en línea para que cualquiera, sin importar su ubicación o circunstancias, pueda acceder y aprender a su propio ritmo.

¡Esperamos que disfrutes este emocionante viaje de aprendizaje y descubrimiento en el mundo del desarrollo web con Django!

Part I

Git y Github

2 Módulo 0: Git/Github

2.1 Introducción a Git y Control de Versiones.

Git es un sistema de control de versiones distribuido que permite rastrear cambios en el código de forma eficiente.

El control de versiones es esencial para mantener un historial de los cambios realizados en un proyecto y facilitar la colaboración en equipo.

2.2 Configuración de un Repositorio en GitHub

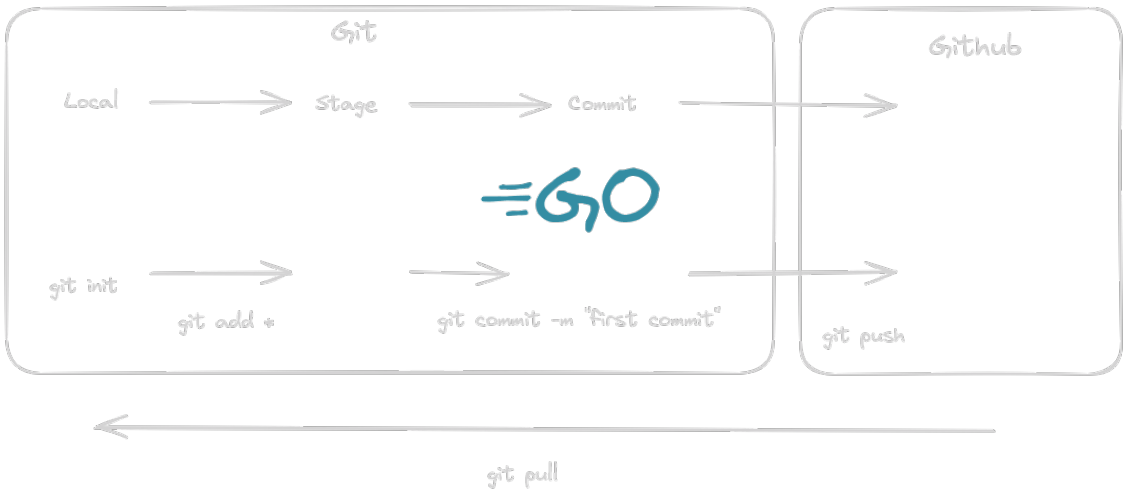
GitHub es una plataforma de alojamiento de repositorios Git en la nube.

Crea una cuenta en GitHub si no tienes una.

Para configurar un nuevo repositorio en GitHub, sigue las instrucciones en la página web.

2.3 Uso Básico de Comandos de Git.

Flujo básico de git.



En la imágen anterior se describe el proceso básico para pasar de **Local** a **Stage**, de **Stage** a **Commit** y de **Commit** a **Github** y/o **Cloud**.

Clonar un repositorio existente desde GitHub a tu máquina local:

```
git clone url_repositorio
```

Crear una nueva rama para trabajar en una funcionalidad específica:

```
git checkout -b nombre_rama
```

Hacer commits para guardar los cambios realizados:

```
git add archivo_modificado.py
git commit -m "Mensaje del commit"
```

Fusionar Ramas y Resolución de Conflictos

Cambiar a la rama principal:

```
git checkout main
```

Fusionar una rama con la rama principal:

```
git merge nombre_rama
```

Resolver conflictos que puedan surgir durante la fusión.

2.4 Colaboración en un Proyecto Utilizando GitHub

Para colaborar en un proyecto en GitHub, realiza lo siguiente:

Haz un Fork del repositorio original en tu cuenta de GitHub.

Clona tu Fork a tu máquina local.

Crea una nueva rama para realizar tus cambios.

Hace commits en tu rama.

Envía un Pull Request al repositorio original para que los colaboradores revisen tus cambios y los fusionen.

3 Ejercicio práctico:

3.1 Configurar un Repositorio y Realizar Cambios

1. Crea un nuevo repositorio en GitHub.
2. Clona el repositorio a tu máquina local con el comando git clone url_repositorio.
3. Crea una nueva rama con el comando git checkout -b nombre_rama.
4. Realiza cambios en tus archivos y haz commits con git add y git commit.
5. Cambia a la rama principal con git checkout main.
6. Fusiona tu rama con la rama principal con git merge nombre_rama.
7. Envía tus cambios al repositorio en GitHub con git push origin main.

¡Excelente! Ahora has aprendido los conceptos básicos de Git y GitHub, así como cómo configurar un repositorio y colaborar en un proyecto utilizando esta plataforma. En los próximos módulos, abordaremos el desarrollo web con Django.

3.2 Resolución del Ejercicio Práctico

Configurar un Repositorio y Realizar Cambios en el Proyecto Blog

En este ejercicio, configuraremos un repositorio Git para el proyecto del blog que vamos a dearrollar en los módulos 1 al 4. Luego, haremos algunos cambios en el proyecto y realizaremos commits para registrar esos cambios en el historial de versiones.

Paso 1: Configurar el repositorio en GitHub

Abre tu cuenta de GitHub y haz clic en el botón “New” para crear un nuevo repositorio.

Asigna un nombre al repositorio y configura la visibilidad como desees.

Opcionalmente, puedes agregar una descripción y una licencia.

Haz clic en “Create repository” para crear el repositorio en GitHub.

Paso 2: Clonar el repositorio en tu máquina local

Copia la URL del repositorio que acabas de crear en GitHub (se verá como https://github.com/tu_usuario/nombre_repositorio.git).

Abre una terminal o línea de comandos en la carpeta donde deseas clonar el repositorio.

Utiliza el siguiente comando para clonar el repositorio en tu máquina local:

```
git clone url_repositorio
```

Reemplaza “url_repositorio” con la URL que copiaste en el paso 1.

Paso 3: Realizar cambios en el proyecto del blog

Abre el proyecto del blog en tu editor de código o IDE favorito.

Realiza algunos cambios en los archivos de tu proyecto, como agregar nuevas funcionalidades, modificar plantillas o corregir errores.

Paso 4: Hacer commits para registrar los cambios

Después de hacer cambios en el proyecto, utiliza los siguientes comandos para hacer commit y registrar esos cambios en el historial de versiones:

```
git add .
git commit -m "Mensaje descriptivo del commit"
```

El comando git add . agrega todos los cambios realizados en los archivos del proyecto al área de preparación, y el comando git commit -m "Mensaje" crea un nuevo commit con un mensaje descriptivo para los cambios realizados.

Paso 5: Enviar los cambios al repositorio en GitHub

Después de hacer commit de los cambios en tu repositorio local, utiliza el siguiente comando para enviar los cambios al repositorio en GitHub:

```
git push origin main`
```

Reemplaza "main" con el nombre de la rama principal de tu proyecto si utilizas otro nombre diferente.

¡Felicitaciones! Ahora has configurado un repositorio Git para tu proyecto y has realizado cambios en el proyecto, registrando esos cambios mediante commits.

Los cambios ahora están disponibles en el repositorio en GitHub. Puedes repetir estos pasos cada vez que desees realizar cambios en el proyecto y mantener un historial de versiones de tu proyecto en GitHub.

Part II

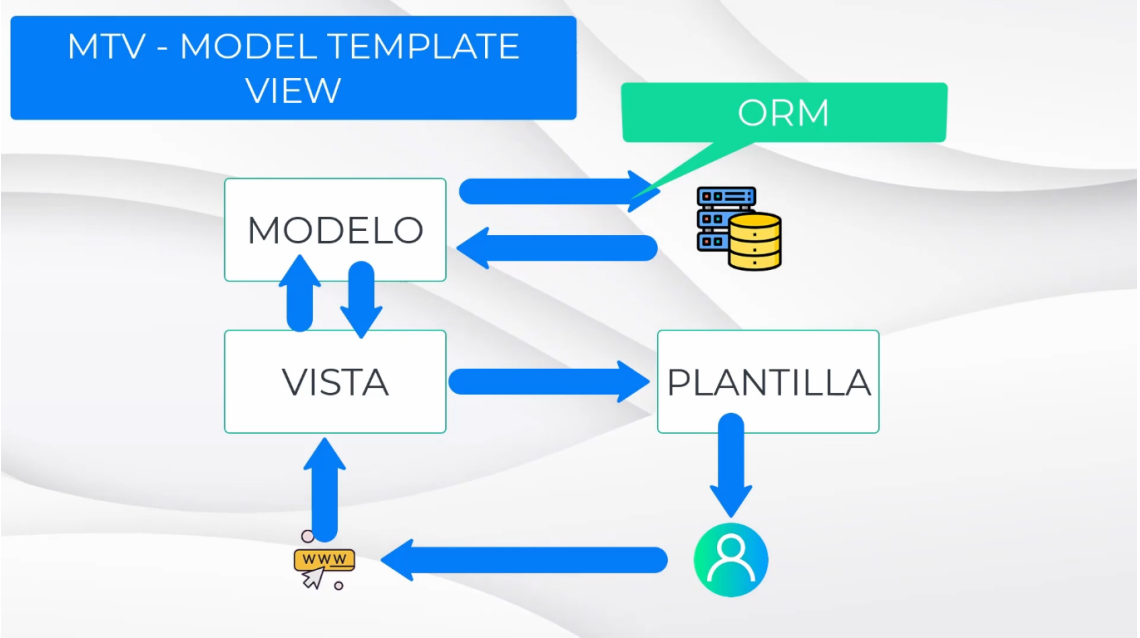
Introducción a Django

4 Módulo 1: Introducción a Django.

4.1 ¿Qué es Django y por qué utilizarlo?

- Django es un framework web de alto nivel basado en Python.
- Facilita el desarrollo rápido de aplicaciones web robustas y seguras.
- Ventajas de Django: MVC (Modelo-Vista-Controlador), administrador de base de datos, seguridad integrada y comunidad activa.

4.2 MVC vs MTV.



4.3 Instalación de Django 4.2.3 y Configuración del Entorno de Desarrollo

Para instalar Django 4.2.3, se recomienda utilizar un entorno virtual (por ejemplo, con virtualenv o conda).

Comandos para instalar Django y crear un entorno virtual:

Instalación de virtualenv

```
pip install virtualenv
```

Creación del entorno virtual

```
python -m venv env
```

Activación del Entorno Virtual

```
cd env/Scripts
activate
```

Salir del directorio Scripts

```
cd ../../
```

Instalación de Django usando pip

```
pip install django==4.2.3
```

4.4 Creación de un Proyecto en Django.

Para crear un nuevo proyecto Django, utilizamos el comando

```
django-admin startproject nombre_proyecto .
```

4.5 Estructura de directorios generada:

```
nombre_proyecto/
    manage.py
        __init__.py
        settings.py
        urls.py
        asgi.py
        wsgi.py
```

4.6 Estructura de Directorios de un Proyecto Django.

La estructura de directorios de un proyecto Django se organiza de la siguiente manera:

```
nombre_proyecto/
    manage.py
```

```
nombre_proyecto/
     __init__.py
     settings.py
     urls.py
     asgi.py
     wsgi.py

otras_aplicaciones/
      ...
```

- **nombre_proyecto/:** Es el directorio raíz del proyecto. Contiene el archivo "manage.py", una herramienta para administrar y ejecutar comandos de Django.
- **nombre_proyecto/nombre_proyecto/:** Este directorio es la configuración principal del proyecto. Contiene varios archivos esenciales, entre ellos:
  - **init.py:** Indica que el directorio es un paquete y permite la importación de módulos dentro de él.
  - **settings.py:** Aquí se encuentran todas las configuraciones del proyecto, como bases de datos, aplicaciones instaladas, rutas de plantillas, configuraciones de seguridad, etc.
  - **urls.py:** Contiene las configuraciones de las URLs del proyecto, es decir, cómo se manejan las solicitudes y se mapean a las vistas.
  - **asgi.py y wsgi.py:** Son archivos de configuración para el servidor ASGI (Asynchronous Server Gateway Interface) y WSGI (Web Server Gateway Interface), respectivamente. Estos archivos son utilizados por servidores web para servir la aplicación Django.
- **otras_aplicaciones/:** Este directorio es opcional y se utiliza para organizar aplicaciones adicionales desarrolladas para el proyecto. Cada aplicación puede tener su propia estructura de directorios.

4.7 Creación de una Aplicación en Django

Una aplicación es un componente reutilizable de un proyecto Django. Comando para crear una nueva aplicación:

```
python manage.py startapp nombre_app
```

Estructura de directorios de una aplicación:

```
nombre_app/
   migrations/
       ...
       __init__.py

       admin.py
```

```
        apps.py
        models.py
        tests.py
        views.py
```

- **nombre_app/:** Es el directorio raíz de la aplicación. Contiene archivos esenciales y directorios para el funcionamiento de la aplicación.
- **migrations/:** Este directorio es generado automáticamente por Django cuando se realizan cambios en los modelos de la aplicación. Contiene archivos de migración que representan los cambios en la base de datos.
- **init.py:** Este archivo indica a Python que el directorio es un paquete y permite la importación de módulos dentro de él.
- **admin.py:** En este archivo se pueden registrar los modelos de la aplicación para que aparezcan en el panel de administración de Django.
- **apps.py:** Es el archivo donde se define la configuración de la aplicación, como su nombre y configuraciones adicionales.
- **models.py:** Aquí se definen los modelos de la aplicación utilizando la clase "Model" de Django. Los modelos representan las tablas en la base de datos y definen los campos y relaciones de la aplicación.
- **tests.py:** Es el archivo donde se pueden escribir pruebas unitarias y de integración para la aplicación.
- **views.py:** Contiene las vistas de la aplicación, que son funciones o clases que manejan las solicitudes y generan las respuestas. Las vistas determinan qué se muestra en las páginas web de la aplicación.

5 Ejemplo Práctico:

5.1 Creación de un Proyecto de E-commerce en Django

En esta actividad práctica, crearás un nuevo proyecto de e-commerce en Django desde cero. Asegúrate de tener Django instalado en tu entorno de desarrollo antes de comenzar.

Paso 1: Crear un Proyecto de Django

Abre una terminal o línea de comandos en la ubicación donde desees crear tu proyecto de e-commerce.

Ejecuta el siguiente comando para crear un nuevo proyecto Django llamado "mi_ecommerce":

```
django-admin startproject mi_ecommerce
```

Verás que se ha creado un nuevo directorio llamado "mi_ecommerce" que contiene la estructura inicial del proyecto.

Paso 2: Crear una Aplicación para el E-commerce

Cambia al directorio recién creado "mi_ecommerce":

```
cd mi_ecommerce
```

Ahora, crea una nueva aplicación llamada "ecommerce" utilizando el siguiente comando:

```
python manage.py startapp ecommerce
```

Se creará un nuevo directorio "ecommerce" dentro de tu proyecto, que contendrá todos los archivos necesarios para la aplicación.

Paso 3: Configurar el Proyecto y la Aplicación

Abre el archivo "settings.py" ubicado en el directorio "mi_ecommerce/mi_ecommerce".

Asegúrate de agregar la aplicación "ecommerce" en la lista de "INSTALLED_APPS" para que Django la reconozca:

```
INSTALLED_APPS = [
    # Otras aplicaciones...
    'ecommerce',
]
```

Paso 4: Ejecutar el Servidor de Desarrollo

Ahora, ejecuta el servidor de desarrollo para ver el proyecto en acción:

```
python manage.py runserver
```

Abre tu navegador y visita la dirección "http://localhost:8000/". Deberías ver la página de bienvenida de Django.

Si has llegado a este punto, ¡Felicidades! Has creado con éxito un nuevo proyecto de e-commerce en Django y una aplicación llamada "ecommerce". Ahora puedes comenzar a desarrollar las funcionalidades del e-commerce y diseñar las diapositivas para cada uno de los módulos del curso.

¡Buena suerte!

6 Actividad Práctica.

6.1 Creación de Un Blog

En esta actividad práctica, crearás un nuevo proyecto de Blog en Django desde cero. Asegúrate de tener Django instalado en tu entorno de desarrollo antes de comenzar.

- ☐ Iniciar un Repositorio
- ☐ Crear el entorno virtual
- ☐ Activación del entorno virtual
- ☐ Instalación de Django
- ☐ Crear el archivo requirements.txt
- ☐ Agregar el archivo .gitignore
- ☐ Crear un Proyecto con Django
- ☐ Crear la App dentro del Proyecto
- ☐ Agregar la App al Archivo settings.py del Proyecto
- ☐ Correr el Servidor de Pruebas

6.2 Resolución de la Actividad Práctica:

Para crear el proyecto de Blog en Django y completar las actividades prácticas, sigue los siguientes pasos:

Paso 1: Iniciar un Repositorio

Inicia un nuevo repositorio en tu sistema de control de versiones (por ejemplo, en GitHub) para gestionar el código de tu proyecto.

Paso 2: Crear el Entorno Virtual

Crea un nuevo entorno virtual para aislar las dependencias del proyecto y evitar conflictos con otras aplicaciones Python instaladas en tu sistema.

```
# Ejecutar en la terminal o consola
python -m venv mi_entorno_virtual
```

Paso 3: Activación del Entorno Virtual

Activa el entorno virtual antes de instalar Django o trabajar en el proyecto.

```
# En Windows
mi_entorno_virtual\Scripts\activate
```

```
# En macOS o Linux
source mi_entorno_virtual/bin/activate
```

Paso 4: Instalación de Django

Dentro del entorno virtual, instala Django utilizando pip.

```
# Ejecutar en la terminal o consola
pip install django
```

Paso 5: Crear el Archivo requirements.txt

Crea un archivo "requirements.txt" en la raíz del proyecto para almacenar todas las dependencias de Python utilizadas en el proyecto.

```
# requirements.txt
Django==4.2.3
```

Paso 6: Agregar el Archivo .gitignore

Crea un archivo ".gitignore" en la raíz del proyecto para evitar que los archivos y directorios innecesarios se incluyan en el repositorio.

```
# .gitignore
mi_entorno_virtual/
__pycache__/
*.pyc
db.sqlite3
```

Paso 7: Crear un Proyecto con Django

Crea un nuevo proyecto de Django llamado "mi_blog".

```
# Ejecutar en la terminal o consola
django-admin startproject mi_blog
```

Paso 8: Crear la App dentro del Proyecto

Crea una nueva aplicación llamada "blog" dentro del proyecto "mi_blog".

```
# Ejecutar en la terminal o consola
cd mi_blog
python manage.py startapp blog
```

Paso 9: Agregar la App al Archivo settings.py del Proyecto

Abre el archivo "settings.py" en la carpeta "mi_blog" y agrega la aplicación "blog" a la lista de aplicaciones instaladas.

```
# settings.py

INSTALLED_APPS = [
    ...
    'blog',
]
```

Paso 10: Correr el Servidor de Pruebas

Para verificar que todo está configurado correctamente, inicia el servidor de pruebas de Django.

```
# Ejecutar en la terminal o consola
python manage.py runserver
```

Ahora podrás acceder a la página de inicio de Django en tu navegador web en la dirección "http://127.0.0.1:8000/".

Con estos pasos, has creado un nuevo proyecto de Blog en Django, configurado un entorno virtual, instalado Django y creado una nueva aplicación dentro del proyecto. Ahora estás listo para comenzar a desarrollar tu blog utilizando Django.

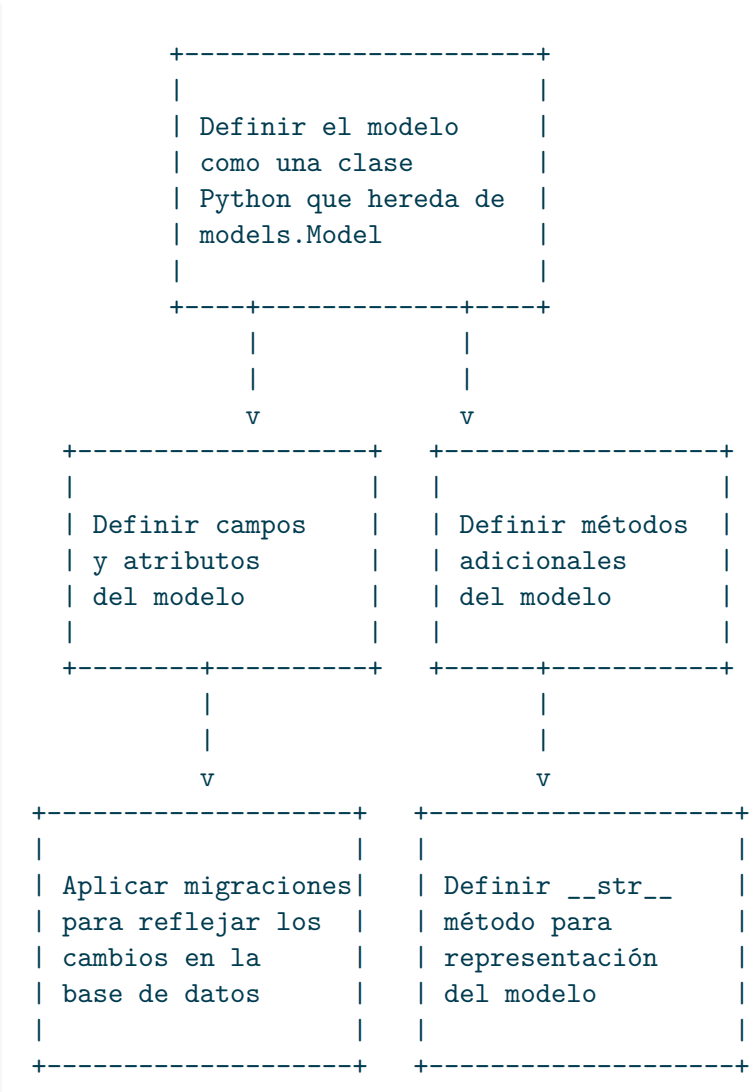
Part III

Modelos y Bases de Datos

7 Módulo 2: Modelos y Bases de Datos.

7.1 Diseño de Modelos en Django

En este diagrama, se muestra el flujo de diseño de modelos en Django.



7.2 1. Definimos el Modelo.

En Django, los modelos son la base para diseñar la estructura de la base de datos de nuestra aplicación web.

Cada modelo representa una tabla en la base de datos y define los campos que estarán presentes en dicha tabla.

Los modelos son definidos como clases Python que heredan de models.Model, lo que permite que Django maneje automáticamente la creación y gestión de la base de datos.

7.2.1 Creación de un Modelo.

Para crear un modelo en Django, primero definimos una clase Python que representa la tabla en la base de datos. Por ejemplo, si deseamos crear un modelo para representar las publicaciones en nuestro blog, podemos definirlo de la siguiente manera:

```
from django.db import models

class Publicacion(models.Model):
    titulo = models.CharField(max_length=200)
    contenido = models.TextField()
    fecha_publicacion = models.DateTimeField()
```

En este ejemplo, hemos definido el modelo Publicacion con tres campos: titulo, contenido y fecha_publicacion.

Cada campo se representa mediante un atributo de la clase, donde: **models.CharField** representa un campo de texto, **models.TextField** representa un campo de texto más largo y **models.DateTimeField** representa una fecha y hora.

7.3 2. Definir Métodos del Modelo.

Además de los campos, también podemos definir métodos en el modelo para realizar acciones específicas o para dar formato a los datos. Por ejemplo, podríamos agregar un método que nos devuelva una representación más legible de la publicación:

```
from django.db import models

class Publicacion(models.Model):
    titulo = models.CharField(max_length=200)
    contenido = models.TextField()
    fecha_publicacion = models.DateTimeField()

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

En este caso, hemos definido el método **str** que se ejecutará cuando necesitemos obtener una representación de texto del objeto Publicacion.

En este caso, hemos decidido que la representación será simplemente el título de la publicación.

Para poder probar los cambios que hemos realizado vamos a registrar nuestro modelo en el archivo **admin.py**

```
#  admin.py

from .models import Publicacion

admin.site.register(Publicacion)
```

7.4 3. Aplicar Migraciones.

Una vez que hemos definido nuestro modelo, necesitamos aplicar las migraciones para que los cambios se reflejen en la base de datos.

```
# Ejecutar en la terminal o consola
python manage.py makemigrations
python manage.py migrate
```

Con estos pasos, hemos diseñado nuestro modelo de Publicaciones en Django y aplicado las migraciones para crear la tabla correspondiente en la base de datos. Ahora estamos listos para utilizar nuestro modelo y almacenar datos en la base de datos.

Finalmente creamos un superusuario para acceder a la administración de nuestro proyecto.

```
python manage.py createsuperuser
```

Llenamos un pequeño formulario que nos pide: nombre de usuario, correo electrónico (no obligatorio), password, repeat again password.

Y listo para poder acceder a la administración de nuestro proyecto nos dirigimos a la siguiente url http://127.0.0.1:8000/admin

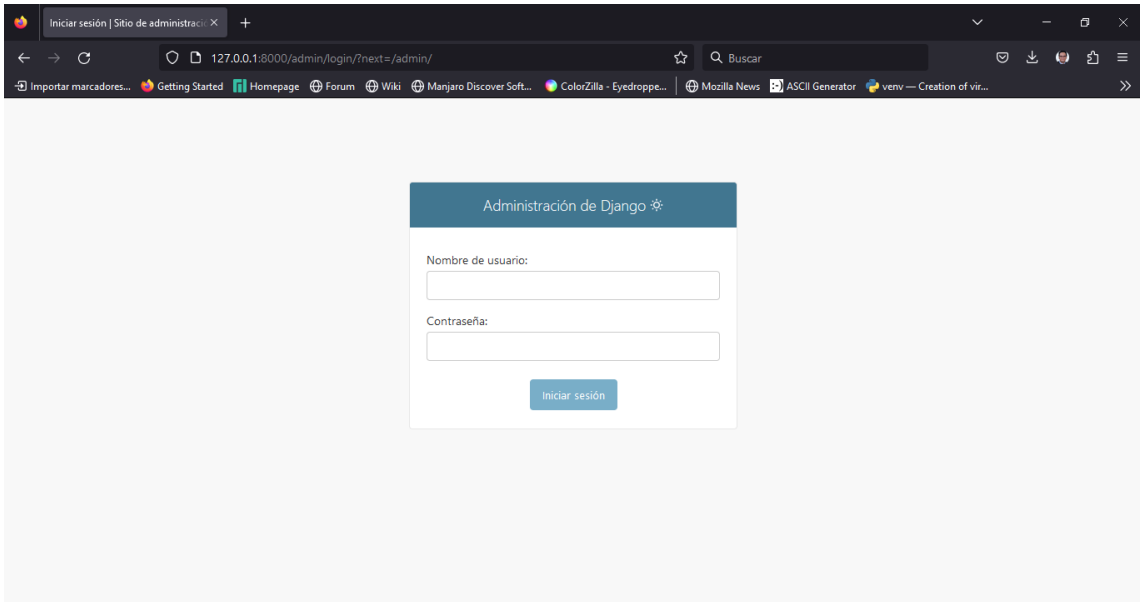
De forma gráfica ingresamos nuestro usuario y contraseña creado.

7.5 Ejemplo de las clases Publicación y Comentario en Django:

```
# En el archivo models.py de la aplicación "Publicaciones"

from django.db import models

class Publicacion(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()
    pub_date = models.DateTimeField(auto_now_add=True)
```




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

class Comentario(models.Model):
    publicacion = models.ForeignKey(Publicacion, on_delete=models.CASCADE)
    author = models.CharField(max_length=50)
    content = models.TextField()
    pub_date = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return f"Comentario de {self.author} en {self.publicacion}"
```

En este ejemplo, hemos definido dos clases:

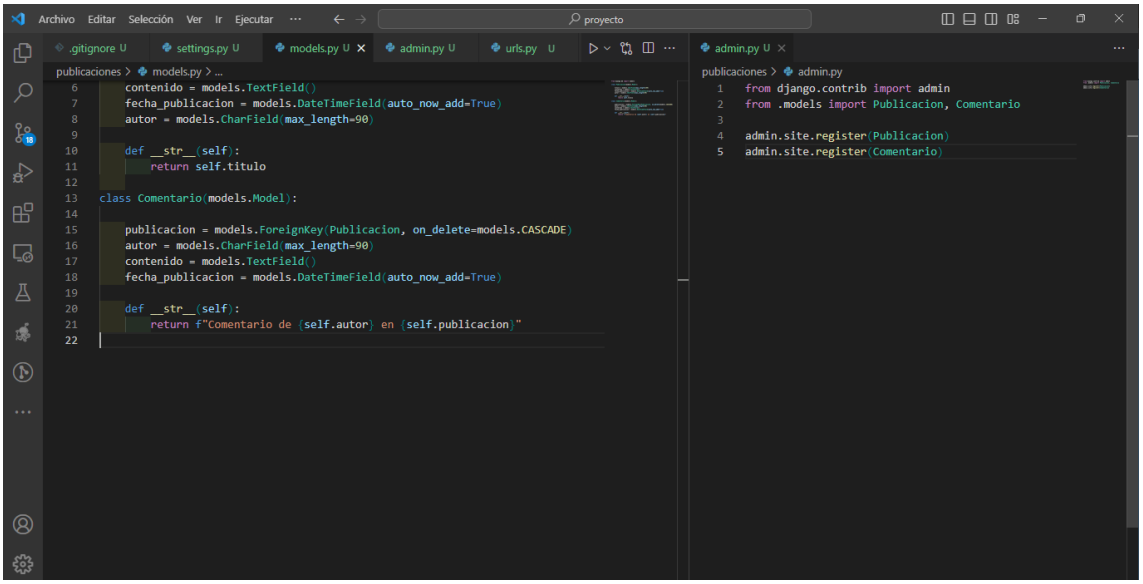
La clase "Publicacion": Representa una publicación en el blog y tiene tres campos: **title** (título de la publicación), **content** (contenido de la publicación) y **pub_date** (fecha de publicación).

La fecha de publicación se establece automáticamente utilizando la función **auto_now_add=True**.

También hemos definido un método "str" para que al imprimir una instancia de la clase, se muestre el título de la publicación.

La clase "Comentario": Representa un comentario en una publicación de publicacion específica y tiene cuatro campos: **publicacion** (clave externa que se relaciona con la publicacion al que pertenece el comentario), **author** (nombre del autor del comentario), **content** (contenido del comentario) y **pub_date** (fecha de publicación del comentario).

Al igual que en la clase **Publicacion**, hemos definido un método "str" para mostrar información útil al imprimir una instancia de la clase.



7.6 Migraciones de la Base de Datos

Las migraciones en Django son una forma de gestionar los cambios en la estructura de la base de datos de manera controlada y consistente.

Representan los cambios en la estructura de la base de datos en forma de archivos Python y se utilizan para crear, modificar o eliminar tablas y campos.

Cuando definimos nuestros modelos en Django (como se mostró en el ejemplo de la clase "Publicación" y "Comentario"), estamos describiendo la estructura de nuestras tablas en la base de datos.

Sin embargo, antes de que estos modelos se puedan utilizar, Django necesita traducirlos en el lenguaje específico del motor de base de datos que estamos utilizando (por ejemplo, PostgreSQL, MySQL, SQLite, etc.).

Es aquí donde entran en juego las migraciones.

Cuando creamos o modificamos modelos, Django genera automáticamente archivos de migración que contienen instrucciones para aplicar los cambios necesarios en la base de datos. Cada migración representa un paso en la evolución de la estructura de la base de datos.

Comandos para crear y aplicar migraciones:

```
python manage.py makemigrations
```

Este comando se utiliza para crear una nueva migración a partir de los cambios detectados en los modelos. Cuando ejecutamos este comando, Django analiza los modelos definidos en nuestra aplicación y compara la estructura actual con la estructura de la última migración aplicada.

Luego, genera una nueva migración que contiene las instrucciones para llevar la base de datos a su estado actual.

```
python manage.py migrate
```

Una vez que hemos creado una o varias migraciones, utilizamos este comando para aplicar esas migraciones pendientes y modificar la base de datos de acuerdo con los cambios en los modelos. Django realiza las operaciones necesarias en la base de datos para reflejar la estructura actual de los modelos definidos en nuestra aplicación.

Recuerda: "Es importante ejecutar estos comandos cada vez que realizamos cambios en los modelos para mantener la coherencia entre la estructura de la base de datos y la estructura definida en los modelos, garantizando así la integridad y consistencia de nuestros datos."

7.7 Consultas a la Base de Datos con el ORM de Django.

Cuando construimos aplicaciones web, a menudo necesitamos interactuar con una base de datos para almacenar y recuperar datos.

Django facilita esta tarea mediante su **Object-Relational Mapping (ORM)**.

Pero, **¿Qué es exactamente el ORM y cómo podemos hacer consultas a la base de datos utilizando esta funcionalidad?**

Object-Relational Mapping (ORM): Una forma amigable de interactuar con la base de datos.

Imagina que tienes una biblioteca y quieres mantener un registro de los libros y sus autores en una base de datos. Tradicionalmente, tendrías que escribir consultas SQL para agregar, actualizar o recuperar información. Sin embargo, con el ORM de Django, puedes utilizar objetos de Python para realizar estas tareas sin tener que escribir consultas SQL directamente. ¡Es como si el ORM tradujera automáticamente tus interacciones con objetos de Python a instrucciones SQL!

7.8 Ejemplo de Consulta con el ORM de Django

Supongamos que tienes dos modelos en Django: **Publicacion** con los campos **titulo**, **contenido** y **fecha_publicacion**, y **Comentario** con los campos **texto** y **fecha_comentario**.

Queremos obtener todas las publicaciones que contengan comentarios y mostrar sus títulos, fechas de publicación y la cantidad de comentarios que tienen.

```
from miapp.models import Publicacion, Comentario

# Realizamos una consulta utilizando el ORM de Django
publicaciones_con_comentarios = Publicacion.objects.filter(comentario__isnull=False)

# Mostramos los resultados
for publicacion in publicaciones_con_comentarios:
    cantidad_comentarios = Comentario.objects.filter(publicacion=publicacion).count()
```

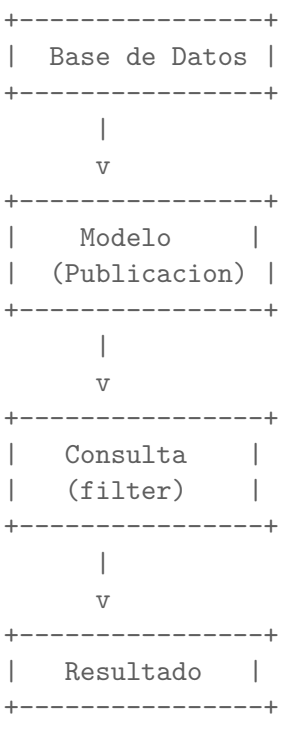
```
    print(f"Título: {publicacion.titulo}, Fecha de Publicación: {publicacion.fecha_publ
```

En este ejemplo, la función filter(comentario___isnull=False) nos permite obtener todas las publicaciones que tienen comentarios, es decir, filtramos aquellas publicaciones donde el campo **comentario** no es nulo.

Luego, utilizamos un bucle for para recorrer los resultados y, para cada publicación, realizamos una nueva consulta para contar la cantidad de comentarios asociados a esa publicación utilizando Comentario.objects.filter(publicacion=publicacion).count().

Así, podemos mostrar los títulos de las publicaciones, sus fechas de publicación y la cantidad de comentarios que tienen.

7.9 Diagrama de Consulta con ORM de Django.

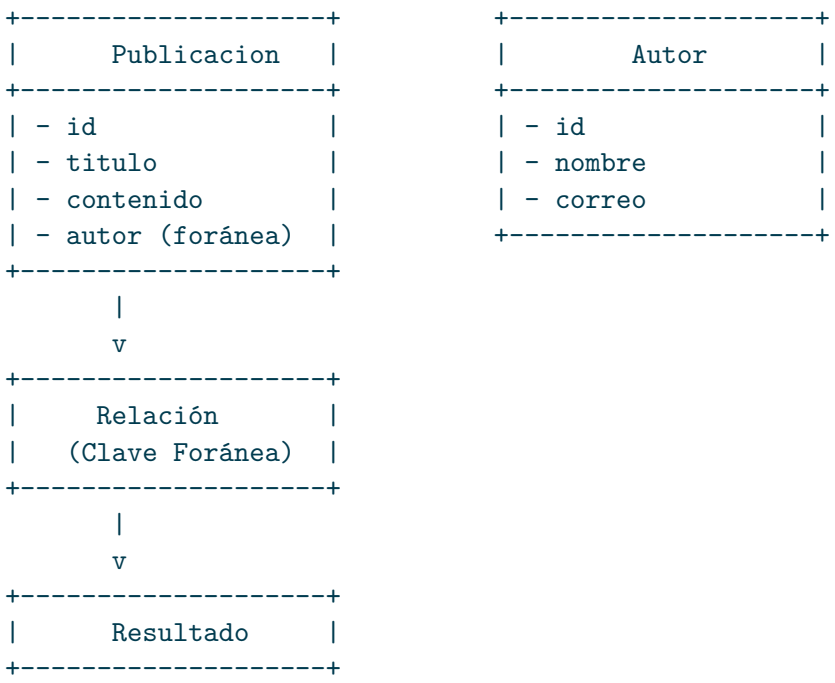


En este diagrama, el modelo **Publicacion** representa la estructura de la tabla en la base de datos.

La consulta se realiza a través del ORM de Django, que traduce la interacción con el modelo en una consulta SQL y devuelve el resultado.

El ORM de Django nos permite realizar consultas de una manera más legible y amigable, lo que facilita enormemente el manejo de datos en nuestras aplicaciones web.

7.10 Relaciones entre Modelos.



Django facilita el establecimiento de relaciones entre modelos mediante claves foráneas y claves primarias.

Por ejemplo,

En un blog, un modelo **Publicacion** podría tener una clave foránea a un modelo **Autor"**, lo que permitiría relacionar cada publicación con su respectivo **autor**.

Además, Django también soporta diferentes tipos de relaciones como Uno a Uno (por ejemplo, un modelo **Perfil** asociado a un modelo **Usuario**), Uno a Muchos (por ejemplo, un modelo **Categoría** que puede tener múltiples publicaciones) y Muchos a Muchos (por ejemplo, una relación de **Seguidores** entre usuarios).

Estas relaciones son fundamentales para organizar y acceder a los datos de manera estructurada en una aplicación Django.

7.11 Clave Primaria, Clave Foránea y Relaciones entre Modelos en Django.

En Django, los modelos representan la estructura de las tablas en la base de datos. Cada modelo se define como una clase Python que hereda de models.Model. Los atributos de la clase representan los campos de la tabla, y entre ellos, destacamos las claves primarias y las claves foráneas.

7.12 Clave Primaria.

Ejemplo:

```
from django.db import models

class Autor(models.Model):
    nombre = models.CharField(max_length=100)
    pais = models.CharField(max_length=50)

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

La clave primaria es un campo único que identifica de forma exclusiva cada registro en una tabla.

Por defecto, Django crea automáticamente un campo id como clave primaria para cada modelo, pero también es posible definir una clave primaria personalizada utilizando el atributo **primary_key=True**.

En este ejemplo, la tabla Autor tendrá una clave primaria id generada automáticamente. La columna id es un campo único que identificará de forma única a cada autor en la base de datos.

7.13 Clave Foránea.

Ejemplo:

```
from django.db import models

class Libro(models.Model):
    titulo = models.CharField(max_length=200)
    autor = models.ForeignKey(Autor, on_delete=models.CASCADE)

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

La clave foránea es un campo que hace referencia a la clave primaria de otra tabla, estableciendo una relación entre ambas tablas.

Esto indica que el valor del campo de la clave foránea en una tabla debe coincidir con el valor de la clave primaria en la otra tabla.

En este ejemplo, el modelo Libro tiene un campo autor que es una clave foránea que hace referencia al modelo Autor. Cada libro está relacionado con un autor específico, y la opción **on_delete=models.CASCADE** indica que si se elimina un autor, todos los libros asociados a ese autor también se eliminarán automáticamente.

7.14 Relaciones entre Modelos de Django:

Relación de Uno a Muchos (OneToMany): Un objeto de un modelo está relacionado con varios objetos de otro modelo.

Se logra utilizando el campo **ForeignKey**.

```
from django.contrib.auth.models import User
from django.db import models

class Publicacion(models.Model):
    titulo = models.CharField(max_length=100)
    contenido = models.TextField()
    autor = models.ForeignKey(User, on_delete=models.CASCADE)
```

Relación de Muchos a Muchos (ManyToMany): Varios objetos de un modelo están relacionados con varios objetos de otro modelo. Se logra utilizando el campo ManyToManyField.

```
from django.db import models

class Etiqueta(models.Model):
    nombre = models.CharField(max_length=50)

class Producto(models.Model):
    nombre = models.CharField(max_length=100)
    etiquetas = models.ManyToManyField(Etiqueta)
```

Relación de Uno a Uno (OneToOne): Un objeto de un modelo está relacionado con exactamente un objeto de otro modelo, y viceversa. Se logra utilizando el campo OneToOneField.

```
from django.contrib.auth.models import User
from django.db import models

class Perfil(models.Model):
    usuario = models.OneToOneField(User, on_delete=models.CASCADE)
    fecha_nacimiento = models.DateField()
```

Estas relaciones nos permiten asociar objetos de diferentes modelos en la base de datos, lo que es esencial para construir aplicaciones web más complejas.

Django facilita el manejo de estas relaciones, lo que nos permite desarrollar aplicaciones de forma más legible y amigable.

Django ofrece una potente forma de establecer relaciones entre modelos, lo que nos permite diseñar y construir aplicaciones web más complejas y ricas en datos.

La elección del tipo de relación dependerá de la lógica de negocio y los requerimientos específicos de la aplicación.

7.15 Ejemplo de Conexión del Proyecto Blog con Bases de Datos

A continuación, se presentará un ejemplo de configuración de Django para conectar el proyecto de blog a tres bases de datos diferentes: SQLite, MySQL, MongoDB y PostgreSQL.

7.16 SQLite:

SQLite es una base de datos incorporada en Django por defecto. No requiere configuración adicional para usarla, ya que Django creará automáticamente un archivo de base de datos SQLite en el directorio del proyecto.

```
# En el archivo settings.py del proyecto "mi_blog"

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
```

7.17 MySQL:

Para conectar la base de datos de MySQL de **forma local en tu máquina**, necesitarás seguir estos pasos:

1. **Instalar MySQL Server:** Lo primero que necesitas es tener instalado el servidor de MySQL en tu máquina. Puedes descargar la versión correspondiente para tu sistema operativo desde el sitio web oficial de MySQL: https://dev.mysql.com/downloads/mysql/
2. **Configurar el servidor:** Una vez que hayas instalado MySQL, debes configurar el servidor.

   Durante la instalación, se te pedirá establecer una contraseña para el usuario "root" que será el administrador del servidor.

   Asegúrate de recordar esta contraseña, ya que la necesitarás más adelante.
3. **Iniciar el servidor:** Después de instalar y configurar MySQL, debes iniciar el servidor.

   Esto puede variar dependiendo de tu sistema operativo, pero generalmente puedes hacerlo desde la línea de comandos o utilizando un programa específico para administrar servidores de bases de datos.

4. **Verificar la conexión:** Una vez que el servidor esté en funcionamiento, verifica que puedas conectarte a él.

   Puedes hacerlo desde la línea de comandos utilizando el cliente de MySQL o usando una herramienta de administración como phpMyAdmin.

5. **Crear una base de datos:** Antes de conectar Django a la base de datos, necesitas crear una base de datos vacía para tu proyecto.

   Puedes hacerlo desde el cliente de MySQL o desde una herramienta de administración.

6. **Configurar Django para usar MySQL:** Ahora que tienes el servidor de MySQL funcionando y una base de datos creada, puedes configurar Django para que utilice MySQL como base de datos, puedes instalar el cliente de MySQL MySQL Workbench.

   Ahora es necesario instalar el connector entre Django y nuestra base de datos, lo podemos hacer con el gestor de paquetes **pip**

```
pip install mysqlclient
```

Ahora vamos a configurar Django.

Para hacerlo, modifica el archivo settings.py de tu proyecto Django y ajusta la configura

```
# settings.py

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'nombre_de_la_base_de_datos',
        'USER': 'nombre_de_usuario',
        'PASSWORD': 'contraseña_del_usuario',
        'HOST': 'localhost',
        'PORT': '3306',
    }
}
```

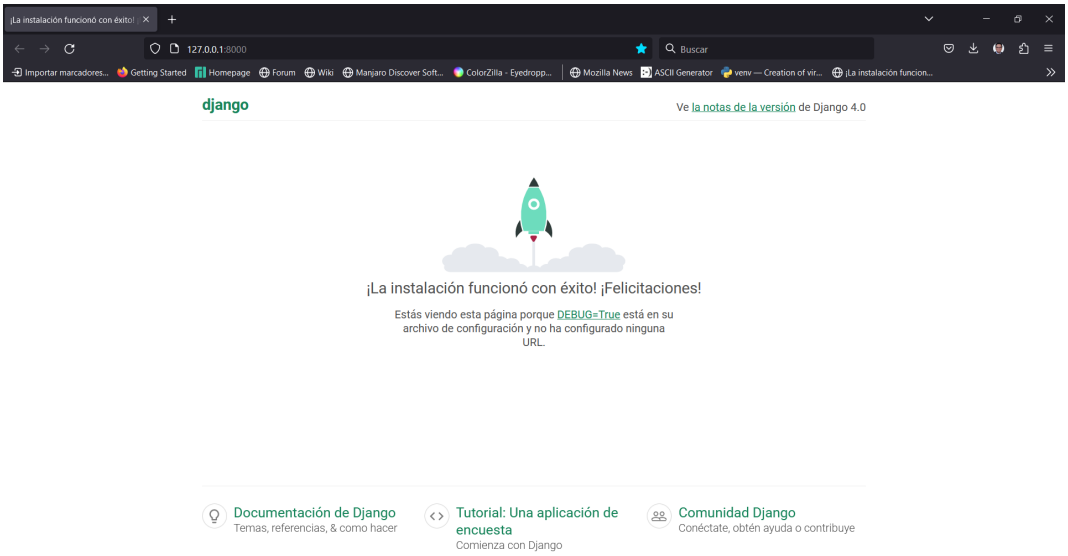
Asegúrate de reemplazar **nombre_de_la_base_de_datos**, **nombre_de_usuario** y **contraseña_del_usuario** con los valores adecuados para tu configuración de MySQL.

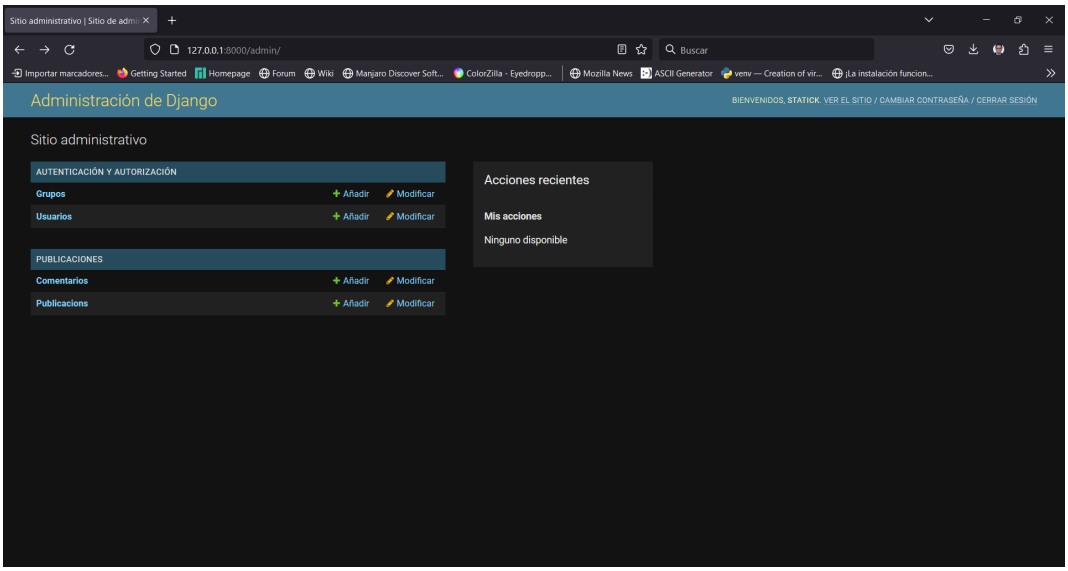
7. **Realizar migraciones:** Una vez configurada la base de datos en Django, es necesario aplicar las migraciones para crear las tablas y campos correspondientes. Ejecuta el siguiente comando para aplicar las migraciones:

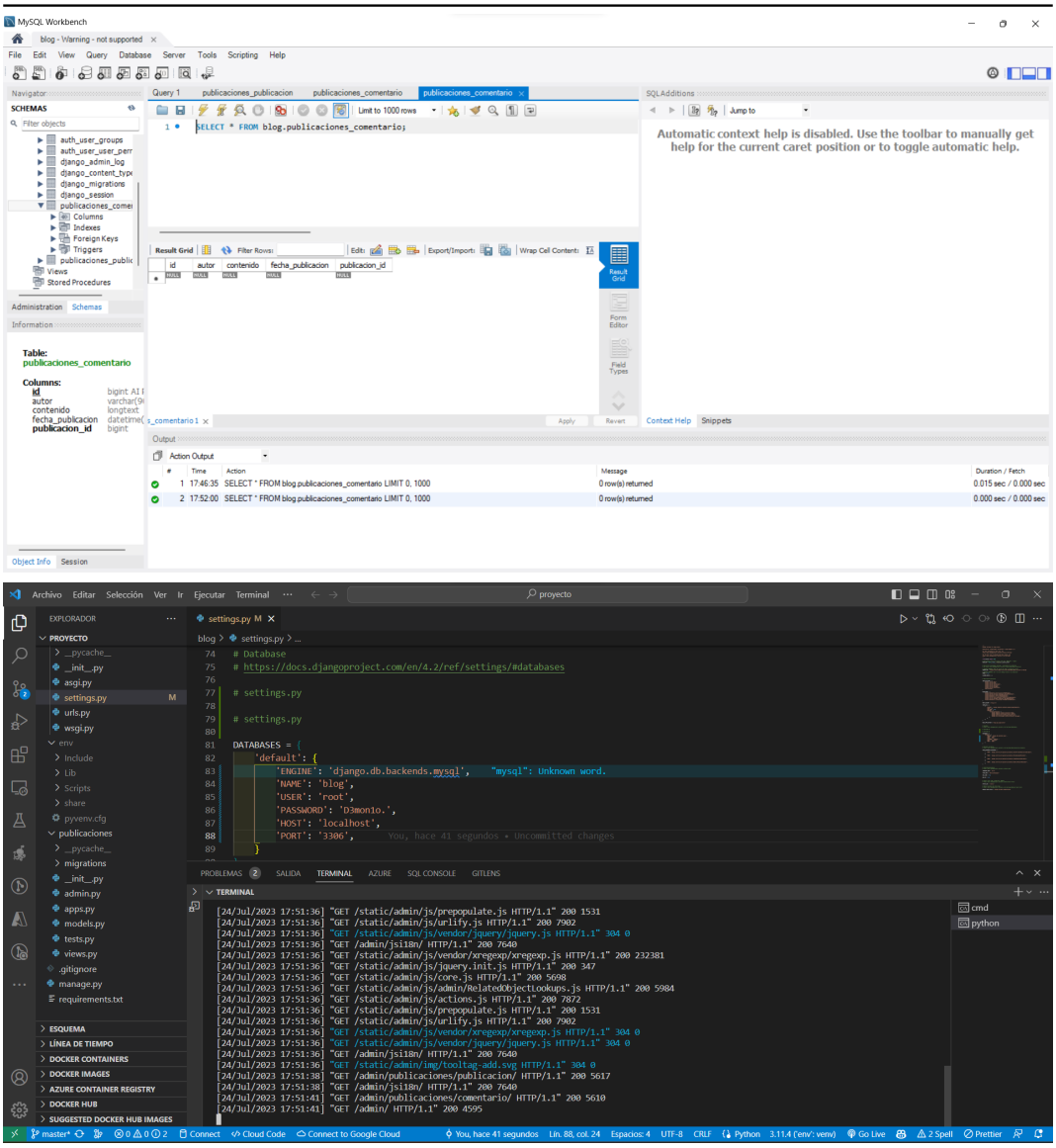
```
python manage.py migrate
```

8. **Verificar la conexión:** Finalmente, verifica que la conexión con la base de datos de MySQL se haya establecido correctamente y que puedas realizar consultas y operaciones desde tu proyecto Django.

   Con estos pasos, tendrás configurada la conexión con la base de datos de MySQL en tu proyecto de Django y podrás utilizarla para almacenar y recuperar datos de forma local en tu máquina.







7.18 PostgreSQL:

Lo primero que necesitas es instalar PostgreSQL en tu sistema operativo.

Puedes descargar la versión correspondiente para tu sistema desde el sitio web oficial de PostgreSQL: https://www.postgresql.org/download/

1. **Configurar PostgreSQL:** Durante la instalación, se te pedirá establecer una contraseña para el usuario "postgres" que será el administrador del servidor PostgreSQL. Asegúrate de recordar esta contraseña, ya que la necesitarás más adelante.

2. **Iniciar el servidor PostgreSQL:** Después de instalar PostgreSQL, debes iniciar el servidor. Esto puede variar dependiendo de tu sistema operativo, pero generalmente

puedes hacerlo desde la línea de comandos o utilizando una herramienta específica para administrar servidores de bases de datos.

3. **Verificar la conexión:** Una vez que el servidor esté en funcionamiento, verifica que puedas conectarte a él. Puedes hacerlo desde la línea de comandos utilizando el cliente psql o usando una herramienta de administración gráfica como pgAdmin.

4. **Instalar el controlador de PostgreSQL para Django:** Para conectar Django con PostgreSQL, necesitas instalar el controlador psycopg2. Puedes instalarlo utilizando el gestor de paquetes pip:

```
pip install psycopg2
```

5. **Configurar Django para usar PostgreSQL:**

   Modifica el archivo settings.py de tu proyecto Django y ajusta la configuración de la base de datos:

```
# settings.py

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'nombre_de_la_base_de_datos',
        'USER': 'nombre_de_usuario',
        'PASSWORD': 'contraseña_del_usuario',
        'HOST': 'localhost',  # Puedes cambiar esta dirección si PostgreSQL está en otr
        'PORT': '',           # Por defecto es el puerto 5432
    }
}
```

Asegúrate de reemplazar **nombre_de_la_base_de_datos**, **nombre_de_usuario** y **contraseña_del_usuario** con los valores adecuados para tu configuración de PostgreSQL.

8. **Realizar migraciones:**

   Una vez configurada la base de datos en Django, es necesario aplicar las migraciones para crear las tablas y campos correspondientes. Ejecuta el siguiente comando para aplicar las migraciones:

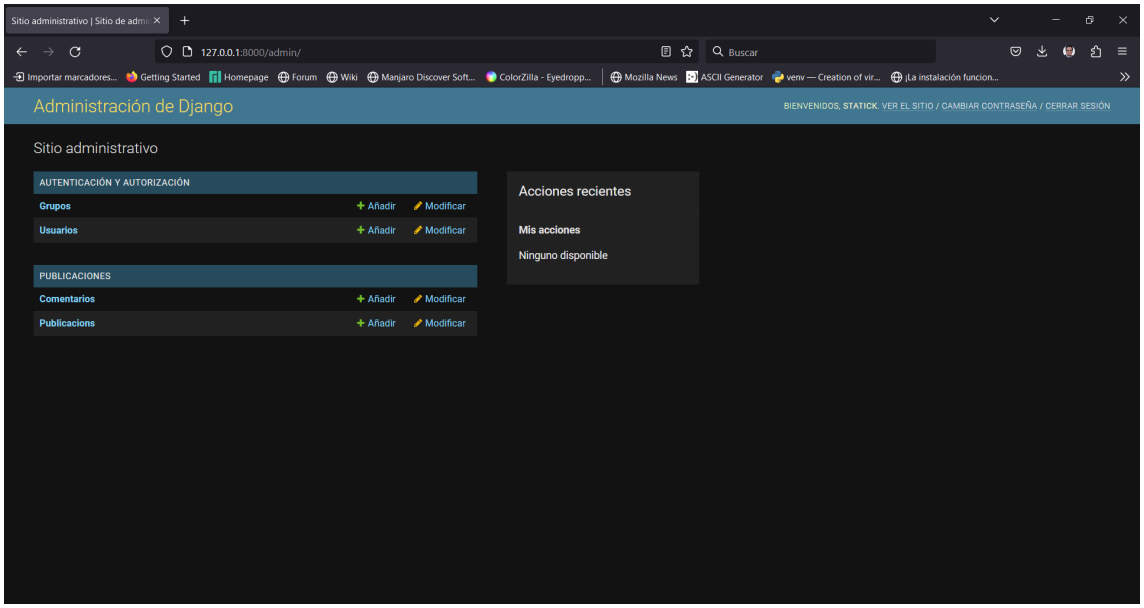
```
python manage.py migrate
```

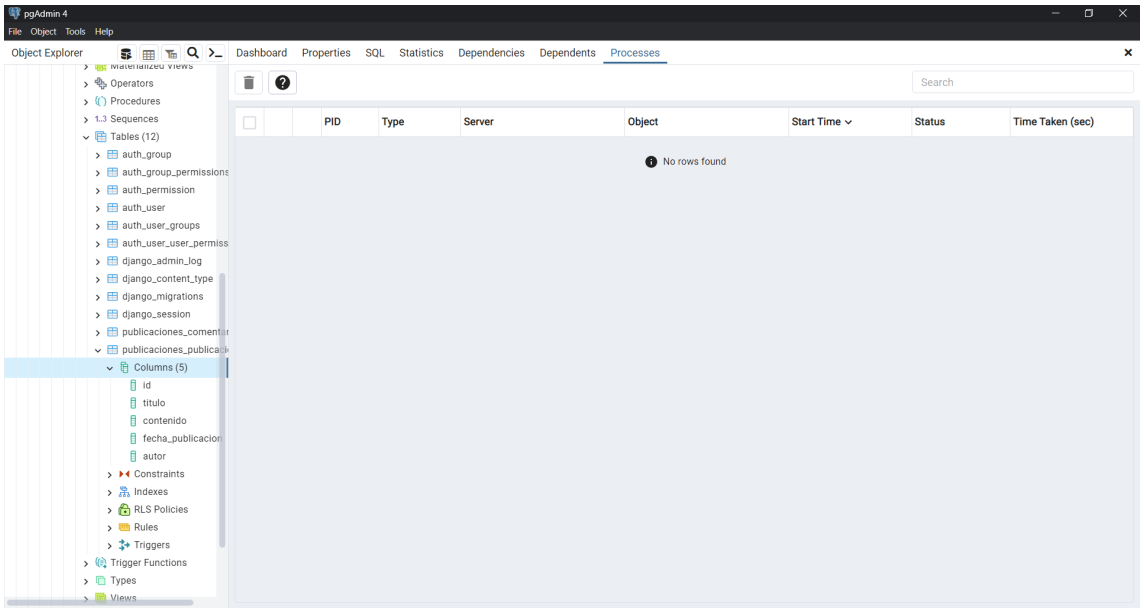
2. **Verificar la conexión:**

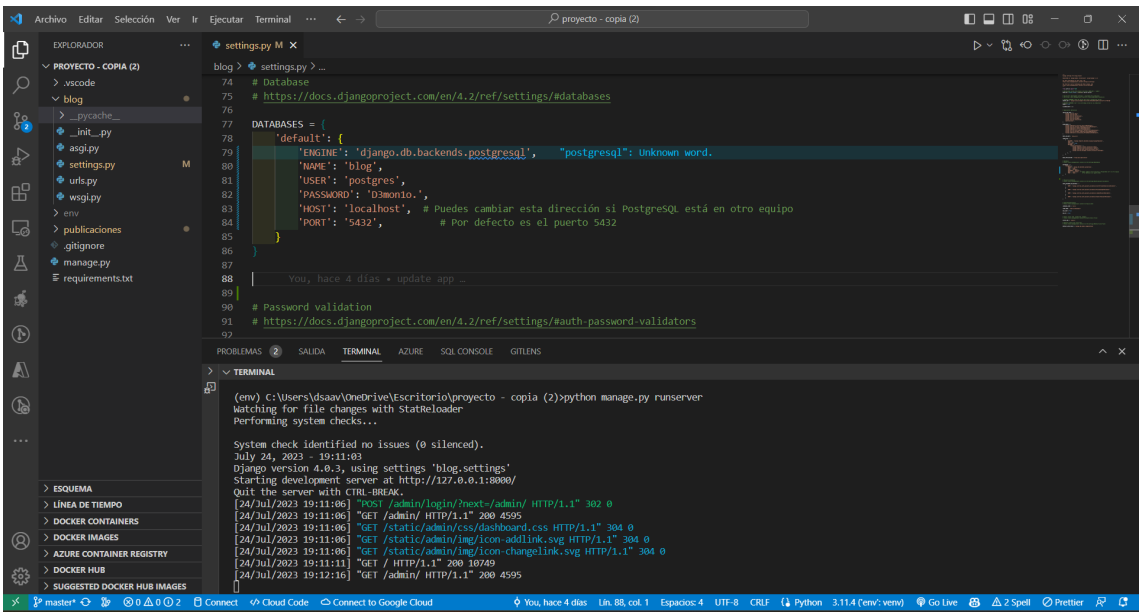
Finalmente, verifica que la conexión con PostgreSQL se haya establecido correctamente y que puedas realizar consultas y operaciones desde tu proyecto Django.

Con estos pasos, tendrás configurada la conexión con PostgreSQL en tu proyecto de Django y podrás utilizarla para almacenar y recuperar datos.

PostgreSQL es una base de datos relacional que ofrece características avanzadas y es ampliamente utilizada en aplicaciones web y proyectos de desarrollo.







7.19 MongoDB:

1. Lo primero que necesitas es instalar **MongoDB** en tu sistema operativo. Puedes descargar la versión correspondiente para tu sistema desde el sitio web oficial de MongoDB: https://www.mongodb.com/try/download/community

2. **Configurar el servidor MongoDB:** Una vez que hayas instalado MongoDB, es necesario configurar el servidor.

   Por defecto, MongoDB se ejecuta en el puerto 27017. Asegúrate de que el servidor MongoDB esté en funcionamiento antes de continuar.

3. **Instalar el controlador de MongoDB para Django:** Para conectar Django con MongoDB, necesitas instalar el controlador de MongoDB para Django, llamado "djongo". Puedes instalarlo utilizando el gestor de paquetes pip:

   ```
   pip install djongo==1.3.1
   ```

   Para que todo funcione con normalidad, es necesario incluir el paquete **pymongo**, para ello se sugiere utilizar el siguiente comando.

   ```
   pip install pymongo==3.12.1
   ```

4. **Configurar Django para usar MongoDB:**

   Modifica el archivo settings.py de tu proyecto Django y ajusta la configuración de la base de datos:

```
 # settings.py

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'publicaciones',
    'djongo',
]

DATABASES = {
    'default': {
        'ENGINE': 'djongo',
        'NAME': 'nombre_de_la_base_de_datos',
        'CLIENT': {
            'host': 'localhost', # Cambia esta dirección si tu servidor MongoDB está en
            'port': 27017,       # Puerto de MongoDB (por defecto es 27017)
        },
    }
}
```

Asegúrate de reemplazar **nombre_de_la_base_de_datos** con el nombre que deseas para tu base de datos MongoDB.

7. **Realizar migraciones:**

Una vez configurada la base de datos en Django, es necesario aplicar las migraciones para crear las colecciones y documentos correspondientes.

Ejecuta el siguiente comando para aplicar las migraciones:

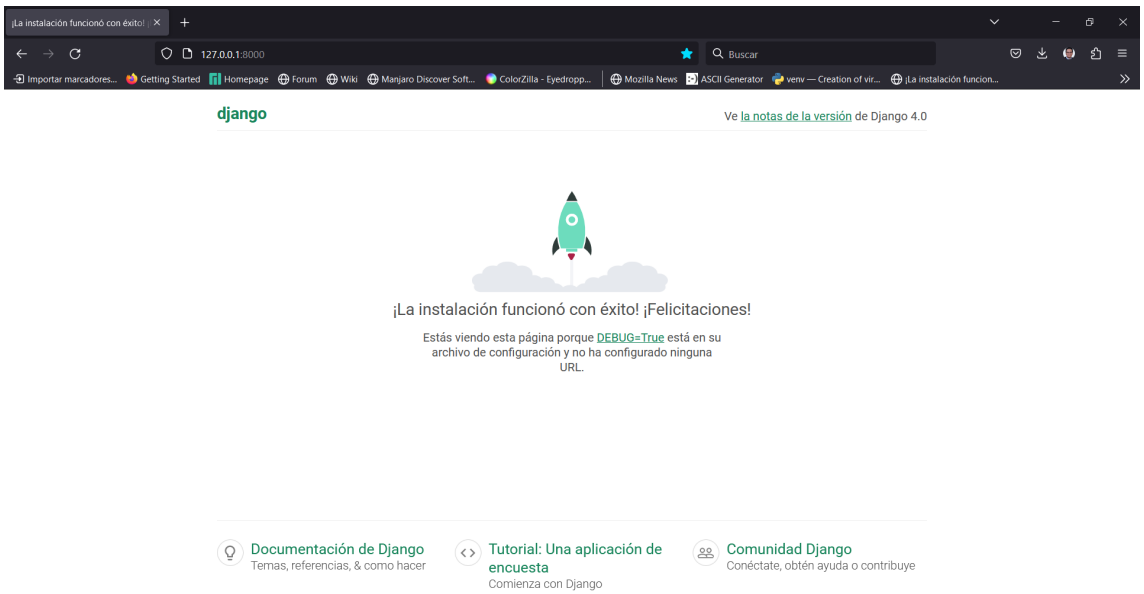
```
python manage.py makemigrations
python manage.py migrate
```

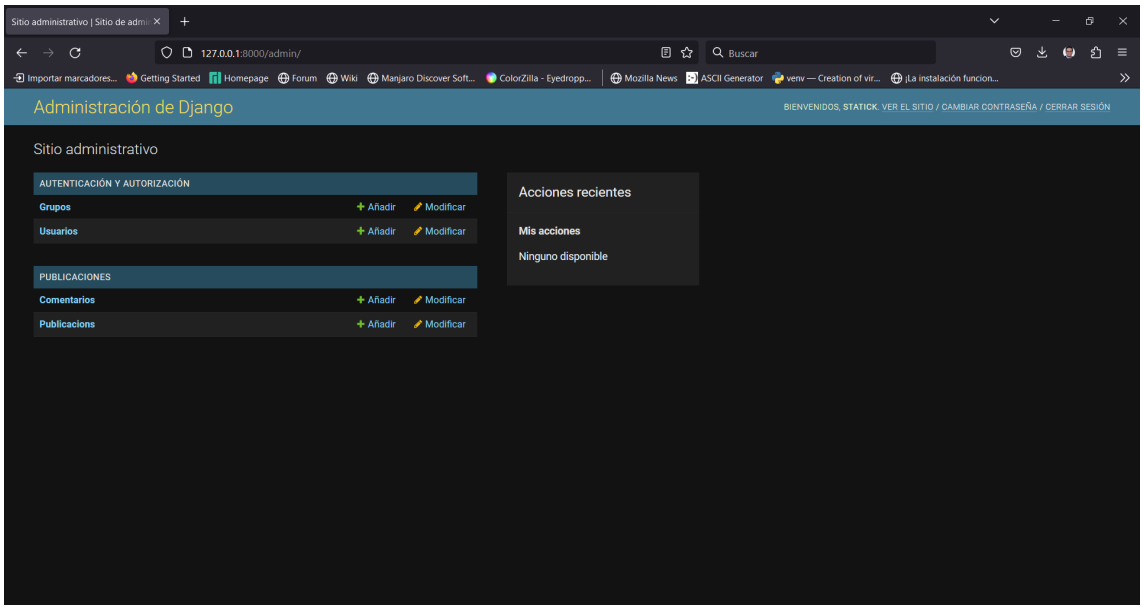
2. **Verificar la conexión:**

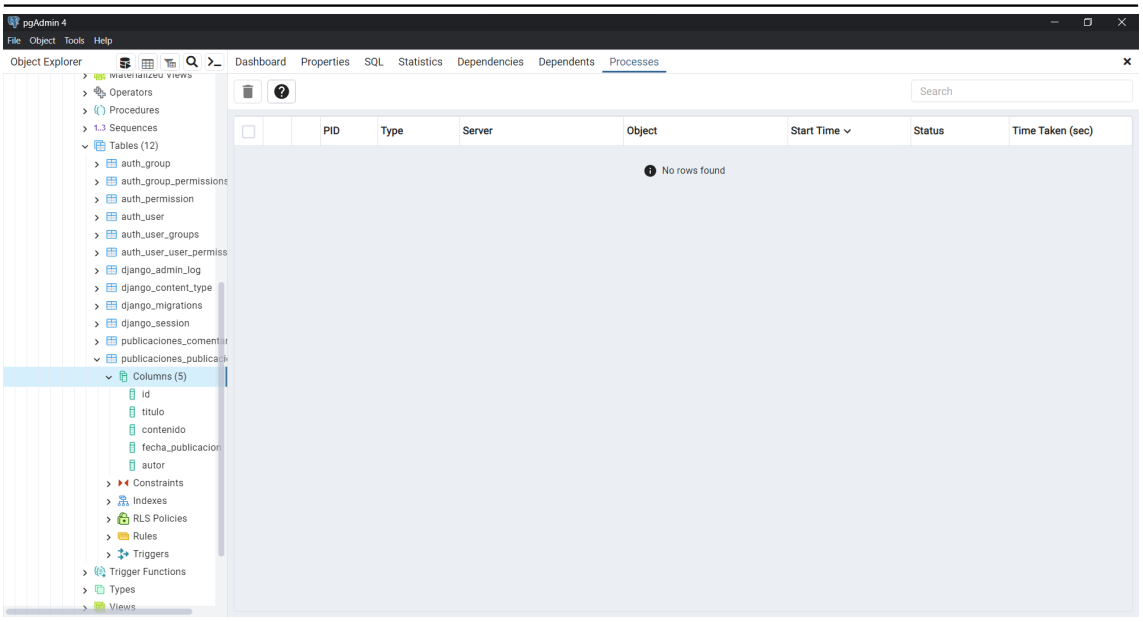
Finalmente, verifica que la conexión con MongoDB se haya establecido correctamente y que puedas realizar consultas y operaciones desde tu proyecto Django.

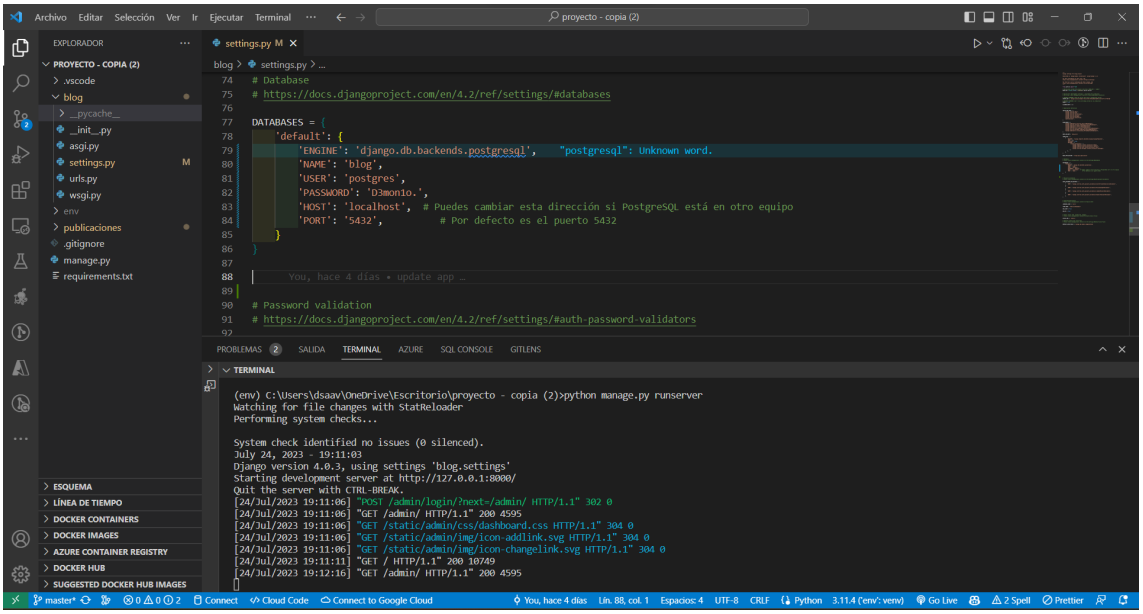
Con estos pasos, tendrás configurada la conexión con MongoDB en tu proyecto de Django y podrás utilizarla para almacenar y recuperar datos.

Es importante mencionar que MongoDB es una base de datos NoSQL, lo que significa que utiliza un modelo de datos basado en documentos en lugar de tablas y filas como las bases de datos relacionales. Esto permite una mayor flexibilidad y escalabilidad en el almacenamiento de datos.


¡La instalación funcionó con éxito!
127.0.0.1:8000
Buscar
Importar marcadores...
Getting Started
Homepage
Forum
Wiki
Manjaro Discover Soft...
ColorZilla - Eyedropp...
Mozilla News
ASCII Generator
venv — Creation of vir...
¡La instalación funcion...
django
Ve la notas de la versión de Django 4.0
¡La instalación funcionó con éxito! ¡Felicitaciones!
Estás viendo esta página porque DEBUG=True está en su archivo de configuración y no ha configurado ninguna URL.
Documentación de Django
Temas, referencias, & como hacer
Tutorial: Una aplicación de encuesta
Comienza con Django
Comunidad Django
Conéctate, obtén ayuda o contribuye


Sitio administrativo | Sitio de admi
127.0.0.1:8000/admin/
Buscar
Importar marcadores...
Getting Started
Homepage
Forum
Wiki
Manjaro Discover Soft...
ColorZilla - Eyedropp...
Mozilla News
ASCII Generator
venv — Creation of vir...
¡La instalación funcion...
Administración de Django
BIENVENIDOS, STATICK. VER EL SITIO / CAMBIAR CONTRASEÑA / CERRAR SESIÓN
Sitio administrativo
AUTENTICACIÓN Y AUTORIZACIÓN
Grupos
+ Añadir
Modificar
Usuarios
+ Añadir
Modificar
PUBLICACIONES
Comentarios
+ Añadir
Modificar
Publicacions
+ Añadir
Modificar
Acciones recientes
Mis acciones
Ninguno disponible


pgAdmin 4
File Object Tools Help
Object Explorer
Dashboard Properties SQL Statistics Dependencies Dependents Processes
Search
PID Type Server Object Start Time Status Time Taken (sec)
No rows found
Operators
Procedures
Sequences
Tables (12)
auth_group
auth_group_permissions
auth_permission
auth_user
auth_user_groups
auth_user_user_permiss
django_admin_log
django_content_type
django_migrations
django_session
publicaciones_coment
publicaciones_publica
Columns (5)
id
titulo
contenido
fecha_publicacion
autor
Constraints
Indexes
RLS Policies
Rules
Triggers
Trigger Functions
Types
Views


Archivo Editar Selección Ver Ir Ejecutar Terminal
proyecto - copia (2)
EXPLORADOR
PROYECTO - COPIA (2)
.vscode
blog
__pycache__
__init__.py
asgi.py
settings.py M
urls.py
wsgi.py
env
publicaciones
.gitignore
manage.py
requirements.txt
settings.py M
blog > settings.py > ...
# Database
# https://docs.djangoproject.com/en/4.2/ref/settings/#databases
DATABASES = {
'default': {
'ENGINE': 'django.db.backends.postgresql', "postgresql": Unknown word.
'NAME': 'blog',
'USER': 'postgres',
'PASSWORD': 'D3mon1o.',
'HOST': 'localhost', # Puedes cambiar esta dirección si PostgreSQL está en otro equipo
'PORT': '5432', # Por defecto es el puerto 5432
}
}
You, hace 4 días • update app ...
Password validation
https://docs.djangoproject.com/en/4.2/ref/settings/#auth-password-validators
PROBLEMAS 2 SALIDA TERMINAL AZURE SQL CONSOLE GITLENS
TERMINAL
(env) C:\Users\dsaav\OneDrive\Escritorio\proyecto - copia (2)>python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...
System check identified no issues (0 silenced).
July 24, 2023 - 19:11:03
Django version 4.0.3, using settings 'blog.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
[24/Jul/2023 19:11:06] "POST /admin/login/?next=/admin/ HTTP/1.1" 302 0
[24/Jul/2023 19:11:06] "GET /admin/ HTTP/1.1" 200 4595
[24/Jul/2023 19:11:06] "GET /static/admin/css/dashboard.css HTTP/1.1" 304 0
[24/Jul/2023 19:11:06] "GET /static/admin/img/icon-addlink.svg HTTP/1.1" 304 0
[24/Jul/2023 19:11:06] "GET /static/admin/img/icon-changelink.svg HTTP/1.1" 304 0
[24/Jul/2023 19:11:11] "GET / HTTP/1.1" 200 10749
[24/Jul/2023 19:12:16] "GET /admin/ HTTP/1.1" 200 4595
ESQUEMA
LÍNEA DE TIEMPO
DOCKER CONTAINERS
DOCKER IMAGES
AZURE CONTAINER REGISTRY
DOCKER HUB
SUGGESTED DOCKER HUB IMAGES
master*
Connect
Cloud Code
Connect to Google Cloud
You, hace 4 días
Lín. 88, col. 1
Espacios: 4
UTF-8
CRLF
Python
3.11.4 ('env': venv)
Go Live
2 Spell
Prettier

8 Ejemplo Práctico:

8.1 Diseño de un Modelo de Usuarios y Publicaciones en un Blog.

8.1.1 Importación de módulos necesarios:

Para definir los modelos en Django, primero importamos los módulos necesarios desde la biblioteca Django.

```
# models.py

from django.contrib.auth.models import AbstractUser
from django.db import models
```

8.1.2 Definición del modelo "Usuario":

El modelo de "Usuario" se crea mediante la herencia de la clase "AbstractUser" proporcionada por Django. Esto nos permite utilizar la funcionalidad de autenticación y autorización incorporada en Django. También podemos agregar campos adicionales según sea necesario.

```
# models.py

class Usuario(AbstractUser):
    # Agrega campos adicionales si es necesario
    pass
```

En el modelo "Usuario", podemos agregar campos adicionales según las necesidades específicas de nuestra aplicación. Al heredar de la clase "AbstractUser" proporcionada por Django, ya contamos con campos comunes para autenticación, como "username", "email" y "password".

Además de los campos heredados, algunos ejemplos de campos adicionales que podríamos agregar al modelo "Usuario" son:

Nombre completo: Podemos agregar un campo para almacenar el nombre completo del usuario.

```
class Usuario(AbstractUser):
    # Campos heredados de AbstractUser
```

```
    ...

    # Campo adicional
    nombre_completo = models.CharField(max_length=255)
```

Fecha de nacimiento: Podemos incluir un campo para registrar la fecha de nacimiento del usuario.

```
class Usuario(AbstractUser):
    # Campos heredados de AbstractUser
    ...

    # Campo adicional
    fecha_nacimiento = models.DateField()
```

Biografía: Podemos permitir que los usuarios agreguen una breve biografía sobre ellos mismos.

```
class Usuario(AbstractUser):
    # Campos heredados de AbstractUser
    ...

    # Campo adicional
    biografia = models.TextField(blank=True, null=True)
```

Imagen de perfil: Podemos agregar un campo para que los usuarios carguen una imagen de perfil.

```
class Usuario(AbstractUser):
    # Campos heredados de AbstractUser
    ...

    # Campo adicional
    imagen_perfil = models.ImageField(upload_to='imagenes_perfil/', blank=True, null=Tr
```

Estos son solo algunos ejemplos de los campos adicionales que podríamos agregar al modelo "Usuario". La elección de los campos depende de los requisitos específicos de la aplicación y qué información adicional deseamos almacenar para nuestros usuarios.

Django proporciona una amplia variedad de tipos de campos para adaptarse a diferentes tipos de datos y necesidades.

8.1.3 Definición del modelo "Publicacion":

El modelo "Publicacion" se crea como una subclase de "models.Model".

Aquí definimos los campos que representarán los atributos de una publicación en el blog, como "titulo", "contenido" y "fecha_publicacion". Cada campo es una instancia de una clase de campo de Django que define el tipo y las propiedades del campo.

```
# models.py

class Publicacion(models.Model):
    titulo = models.CharField(max_length=200)
    contenido = models.TextField()
    fecha_publicacion = models.DateTimeField()
```

Método “str”:

En el modelo “Publicacion”, hemos definido el método “str” que devuelve el título de la publicación cuando se imprime una instancia de la clase.

Esto hace que sea más fácil identificar las publicaciones en el administrador de Django y en cualquier otra parte donde se muestren objetos de la clase “Publicacion”.

```
# models.py

class Publicacion(models.Model):
    # Campos de la publicación...

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

Con estos pasos, hemos definido los modelos “Usuario” y “Publicacion” en Django. Estos modelos representarán las tablas “Usuario” y “Publicacion” en la base de datos y nos permitirán interactuar con ellos mediante el ORM de Django.

9 Actividad Práctica:

9.1 Agrega un Campo Adicional al Modelo de Publicaciones

[ ] Agrega un Campo Adicional al Modelo de Publicaciones y Realiza la Migración Correspondiente

[ ] En el modelo de Publicaciones, agrega un nuevo campo “imagen” de tipo “ImageField”.

[ ] Crea una migración para aplicar los cambios al modelo usando el comando

[ ] Aplica la migración a la base de datos.

9.2 Resolución de la Actividad Práctica.

Paso 1: Agregar un Campo Adicional al Modelo de Publicaciones

En el archivo “models.py” de la aplicación “blog”, agregamos un nuevo campo llamado “imagen” de tipo “ImageField” al modelo “Publicacion”. Esto nos permitirá almacenar imágenes relacionadas con cada publicación.

```
# models.py

from django.db import models

class Publicacion(models.Model):
    titulo = models.CharField(max_length=200)
    contenido = models.TextField()
    fecha_publicacion = models.DateTimeField()
    imagen = models.ImageField(upload_to='publicaciones/', blank=True, null=True)

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

Paso 2: Crear una Migración para Aplicar los Cambios

Después de agregar el nuevo campo “imagen” al modelo, necesitamos crear una migración para aplicar los cambios a la base de datos. Django nos proporciona un comando para generar automáticamente la migración.

```
# Ejecutar en la terminal o consola
python manage.py makemigrations
```

Paso 3: Aplicar la Migración a la Base de Datos

Una vez que se ha creado la migración, la aplicamos a la base de datos con el siguiente comando:

```
# Ejecutar en la terminal o consola
python manage.py migrate
```

Con estos pasos, hemos agregado con éxito un campo adicional "imagen" al modelo de Publicaciones y hemos aplicado la migración a la base de datos para reflejar el cambio. Ahora, cada publicación en el blog tendrá un campo para asociar una imagen, lo que mejorará la experiencia visual para los usuarios.

9.3 Extra.

Para poder manejar el modelo de Publicaciones desde la administración de Django, es necesario registrar el modelo en el archivo "admin.py" de la aplicación "blog". De esta manera, podremos acceder a las publicaciones y administrarlas desde la interfaz de administración de Django.

Paso 4: Registro del Modelo en el Administrador de Django

En el archivo "admin.py" de la aplicación "blog", registramos el modelo "Publicacion" para que sea accesible desde la interfaz de administración.

```
# admin.py

from django.contrib import admin
from .models import Publicacion

admin.site.register(Publicacion)
```

Con este registro, el modelo "Publicacion" estará disponible en la interfaz de administración de Django.

Ahora, cuando ingreses a la URL "/admin/" en tu aplicación, podrás iniciar sesión como superusuario y ver la sección de "Publicacions", donde podrás agregar, editar y eliminar publicaciones desde la interfaz de administración.

Esto facilitará el manejo de las publicaciones sin tener que acceder directamente a la base de datos.

9.4 Conclusión.

Con esto, hemos completado el Módulo 2 de Modelos y Bases de Datos en Django.

Has aprendido a diseñar modelos en Django, realizar migraciones de la base de datos, hacer consultas utilizando el ORM de Django, establecer relaciones entre modelos y usar formularios en Django.

En el próximo módulo, continuaremos explorando más características de Django, como vistas y plantillas.

Part IV

Vistas y Plantillas

10 Módulo 3: Vistas y Plantillas.

10.1 Creación de Vistas en Django

Las vistas en Django son funciones que procesan las solicitudes del usuario y devuelven una respuesta HTTP. Cada vista debe recibir una solicitud como argumento y devolver una respuesta.

Ejemplo de una vista que muestra un mensaje de bienvenida:

Paso 1: En el archivo "views.py" de la aplicación "publicaciones", agrega el código de la vista de bienvenida:

```
# publicaciones/views.py

from django.http import HttpResponse

def vista_bienvenida(request):
    return HttpResponse("¡Bienvenido al blog!")
```

Paso 3: Configura la URL para la vista en el archivo "urls.py" de la aplicación "publicaciones":

```
# publicaciones/urls.py

from django.urls import path
from . import views

urlpatterns = [
    path('bienvenida/', views.vista_bienvenida, name='vista_bienvenida'),
]
```

Paso 4: Configura la URL de la aplicación **publicaciones** en el archivo **urls.py** del proyecto:

```
# blog/urls.py

from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
```

```
    path('publicaciones/', include('publicaciones.urls')), # Agrega esta línea para inc
]
```

Paso 5: Ahora, ejecuta el servidor de desarrollo con el siguiente comando:

```
python manage.py runserver
```

Paso 6: Abre tu navegador web e ingresa a la siguiente dirección:

http://127.0.0.1:8000/publicaciones/bienvenida/

Deberías ver el mensaje "¡Bienvenido al blog!" en el navegador.

10.2 Otra forma de generar un Hola Mundo en Django.

Paso 1: Creación de la vista en views.py:

En esta etapa, se crea una vista llamada "HolaMundoView" utilizando la clase Template-View. Esta vista simplemente renderiza la plantilla "hola_mundo.html" que muestra un mensaje "Hola Mundo!".

```
# views.py

from django.shortcuts import render
from django.views.generic import TemplateView

class HolaMundoView(TemplateView):
    template_name = 'hola_mundo.html'
```

Paso 2: Configuración de las URLs en urls.py:

En el archivo urls.py de la aplicación "publicaciones", se define la URL para la vista "HolaMundoView". También se incluyen las URLs de la aplicación en las URLs globales del proyecto.

```
# urls.py de publicaciones

from django.urls import path
from .views import HolaMundoView

urlpatterns = [
    path('hola_mundo/', HolaMundoView.as_view(), name='hola_mundo'), ]
```

Paso 3 Creación de la plantilla hola_mundo.html:

La plantilla "hola_mundo.html" es un archivo HTML simple que muestra el mensaje "Hola Mundo!" en un encabezado h1.

```
<!DOCTYPE html>
<html>
<head>
<title>Hola Mundo</title>
</head>
<body>
<h1>Hola Mundo!</h1>
</body>
</html>
```

Paso 4: Configuración de las URLs globales del proyecto en urls.py:

En el archivo urls.py del proyecto principal, se incluye la URL de la aplicación "publicaciones" utilizando la función "include". Esto permitirá acceder a las URLs de la aplicación a través de la URL base "publicaciones/".

```
# urls.py del proyecto
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('publicaciones/', include('publicaciones.urls')),
    ]
```

Con estos pasos, hemos creado una aplicación simple que muestra el mensaje "Hola Mundo!" en la página cuando accedemos a la URL "publicaciones/hola_mundo/". Además, hemos configurado la conexión entre las URLs de la aplicación y las URLs globales del proyecto. A partir de aquí, podemos agregar más funcionalidades y vistas a nuestra aplicación utilizando los modelos "Publicacion" y "Comentario".

10.3 Mostrar Publicaciones y Comentarios

Para mostrar las publicaciones y comentarios agregados en los modelos, primero, asegúrate de que hayas definido correctamente los modelos "Publicacion" y "Comentario" en el archivo models.py de la aplicación "publicaciones" como se mostró en ejemplos anteriores.

10.4 Crea una vista para mostrar las publicaciones:

En el archivo views.py de la aplicación "publicaciones", crea una vista llamada "ListaPublicacionesView" para mostrar todas las publicaciones:

```
# publicaciones/views.py

from django.views.generic import ListView
from .models import Publicacion

class ListaPublicacionesView(ListView):
    model = Publicacion
    template_name = 'lista_publicaciones.html'
    context_object_name = 'publicaciones'
```

Crea una plantilla para mostrar la lista de publicaciones:

Crea un archivo llamado “lista_publicaciones.html” dentro de la carpeta “templates” de la aplicación “publicaciones”:

```
<!-- publicaciones/templates/lista_publicaciones.html -->

<!DOCTYPE html>
<html>
<head>
    <title>Lista de Publicaciones</title>
</head>
<body>
    <h1>Lista de Publicaciones</h1>
    <ul>
        {% for publicacion in publicaciones %}
            <li>{{ publicacion.titulo }}</li>
            <ul>
                {% for comentario in publicacion.comentarios.all %}
                    <li>{{ comentario.texto }}</li>
                {% endfor %}
            </ul>
        {% endfor %}
    </ul>
</body>
</html>
```

En este ejemplo, estamos utilizando una estructura de bucles for en la plantilla para mostrar las publicaciones y sus comentarios asociados.

Configura las URLs para mostrar la lista de publicaciones:

En el archivo urls.py de la aplicación “publicaciones”, agrega la configuración para mostrar la lista de publicaciones:

```
# publicaciones/urls.py

from django.urls import path
from .views import ListaPublicacionesView
```

```
urlpatterns = [
    path('publicaciones/', ListaPublicacionesView.as_view(), name='lista_publicaciones'
]
```

Actualiza las URLs del proyecto:

En el archivo urls.py del proyecto "blog", actualiza las URLs de la aplicación "publicaciones" para que se muestren en la ruta principal:

```
# blog/urls.py

from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('publicaciones.urls')),
]
```

Ejecuta el servidor de desarrollo:

Ahora, ejecuta el servidor de desarrollo nuevamente con el siguiente comando:

```
python manage.py runserver
```

Accede a la URL http://localhost:8000/publicaciones/ en tu navegador y deberías ver la lista de publicaciones y sus comentarios asociados.

Si no has agregado publicaciones o comentarios en la base de datos, es posible que no veas datos en la lista.

¡Listo! Ahora has configurado el proyecto "blog" para mostrar las publicaciones y comentarios agregados en los modelos "Publicacion" y "Comentario" utilizando el modelo Template View de Django.

10.5 Sistema de Plantillas de Django (Jinja2)

- Django utiliza el motor de plantillas Jinja2 para gestionar la presentación de los datos en las vistas.
- Jinja2 es un poderoso motor de plantillas que permite incrustar código Python y generar HTML de forma dinámica.
- En el contexto del proyecto "blog" que hemos estado desarrollando, el sistema de plantillas de Django se encargará de renderizar las vistas y mostrar las publicaciones con sus comentarios en la plantilla "lista_publicaciones.html".

10.6 Conceptos Principales del Sistema de Plantillas de Django:

Templates: Los templates son archivos HTML que contienen código Python que define cómo se mostrarán los datos en la interfaz de usuario.

En nuestro caso, el archivo "lista_publicaciones.html" será un template donde mostraremos la lista de publicaciones y sus comentarios.

Contexto: El contexto es un diccionario de Python que contiene los datos que se van a renderizar en el template.

En este contexto, proporcionaremos la lista de publicaciones y sus comentarios para que sean mostrados en la plantilla.

Variables de Plantilla: En los templates de Django, podemos utilizar variables de plantilla para acceder a los datos proporcionados en el contexto.

Por ejemplo, podemos utilizar la variable publicaciones para acceder a la lista de publicaciones y sus comentarios.

Directivas de Control: Jinja2 permite utilizar directivas de control, como bucles y condicionales, en los templates para generar contenido de forma dinámica.

Esto nos permite iterar sobre la lista de publicaciones y mostrar cada una de ellas con sus comentarios.

10.7 Pasos para Utilizar el Sistema de Plantillas de Django en el Proyecto "blog":

Pasos para Utilizar el Sistema de Plantillas de Django en el Proyecto "blog":

Paso 1: Crear el archivo "lista_publicaciones.html"

En el directorio "templates" de la aplicación "publicaciones", creamos el archivo "lista_publicaciones.html" donde definiremos la estructura HTML y utilizaremos las variables de plantilla para mostrar los datos.

```
<!-- lista_publicaciones.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Lista de Publicaciones</title>
</head>
<body>
    <h1>Lista de Publicaciones</h1>
    <ul>
        {% for publicacion in publicaciones %}
        <li>
            <h2>{{ publicacion.titulo }}</h2>
            <p>{{ publicacion.contenido }}</p>
```

```
            <h3>Comentarios:</h3>
            <ul>
                {% for comentario in publicacion.comentarios.all %}
                <li>{{ comentario.texto }}</li>
                {% endfor %}
            </ul>
        </li>
        {% endfor %}
    </ul>
</body>
</html>
```

Paso 2: Definir las Vistas

En el archivo "views.py" de la aplicación "publicaciones", definimos las vistas que serán responsables de obtener los datos de la base de datos (en este caso, las publicaciones y sus comentarios) y pasarlos al template.

```
# views.py
from django.shortcuts import render
from .models import Publicacion

def lista_publicaciones(request):
    publicaciones = Publicacion.objects.all()
    context = {'publicaciones': publicaciones}
    return render(request, 'lista_publicaciones.html', context)
```

Paso 3: Utilizar el Contexto

En las vistas, creamos un contexto que contiene los datos que queremos mostrar en el template. En nuestro caso, el contexto contendrá la lista de publicaciones y sus comentarios.

Paso 4: Renderizar el Template

Finalmente, en las vistas, utilizamos el método render() para renderizar el template "lista_publicaciones.html" con el contexto que creamos. Esto generará el contenido HTML dinámico que mostrará las publicaciones y sus comentarios.

Paso 5: Configurar las URLs

En el archivo "urls.py" de la aplicación "publicaciones", definimos la URL que se utilizará para acceder a la vista que renderiza el template "lista_publicaciones.html".

```
# urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('lista/', views.lista_publicaciones, name='lista_publicaciones'),
```

```
]
```

Con estos pasos, habremos integrado el sistema de plantillas de Django (Jinja2) en nuestro proyecto “blog” y podremos mostrar de forma dinámica las publicaciones y sus comentarios en la plantilla “lista_publicaciones.html”. Al acceder a la URL “/lista/”, se mostrará la lista de publicaciones con sus comentarios.

11 CRUD de Publicaciones

11.1 Crear Publicaciones

Paso 1: Crear el formulario de Publicación

En el archivo "forms.py" de la aplicación "publicaciones", creamos un formulario para la creación de publicaciones.

```
# forms.py
from django import forms
from .models import Publicacion

class PublicacionForm(forms.ModelForm):
    class Meta:
        model = Publicacion
        fields = ['titulo', 'contenido', 'autor']
```

Paso 2: Actualizar el archivo "views.py"

En el archivo "views.py" de la aplicación "publicaciones", definimos una vista para crear una nueva publicación y renderizar el formulario.

```
# views.py
from django.shortcuts import render, redirect
from .forms import PublicacionForm

def crear_publicacion(request):
    if request.method == 'POST':
        form = PublicacionForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect('lista_publicaciones')
    else:
        form = PublicacionForm()
    return render(request, 'crear_publicacion.html', {'form': form})
```

Paso 3: Crear la plantilla para el formulario de creación

En el directorio "templates" de la aplicación "publicaciones", creamos el archivo "crear_publicacion.html" que contendrá el formulario de creación de publicaciones.

```
<!-- crear_publicacion.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Crear Publicación</title>
</head>
<body>
    <h1>Crear Nueva Publicación</h1>
    <form method="post">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Crear</button>
    </form>
</body>
</html>
```

Paso 4: Configurar las URLs

En el archivo "urls.py" de la aplicación "publicaciones", agregamos una URL para la vista de creación de publicaciones.

```
# urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('lista/', views.lista_publicaciones, name='lista_publicaciones'),
    path('crear/', views.crear_publicacion, name='crear_publicacion'),
]
```

11.2 Leer Publicaciones

Paso 1: Actualizar el archivo "views.py"

En el archivo "views.py" de la aplicación "publicaciones", definimos una vista para mostrar la lista de publicaciones.

```
# views.py
from django.shortcuts import render
from .models import Publicacion

def lista_publicaciones(request):
    publicaciones = Publicacion.objects.all()
    return render(request, 'lista_publicaciones.html', {'publicaciones': publicaciones}
```

Paso 2: Actualizar el archivo "lista_publicaciones.html"

En la plantilla “lista_publicaciones.html”, podemos acceder a las publicaciones y mostrarlas en una lista.

```
<!-- lista_publicaciones.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Lista de Publicaciones</title>
</head>
<body>
    <h1>Lista de Publicaciones</h1>
    <ul>
        {% for publicacion in publicaciones %}
        <li>
            <h2>{{ publicacion.titulo }}</h2>
            <p>{{ publicacion.contenido }}</p>
            <h3>Comentarios:</h3>
            <ul>
                {% for comentario in publicacion.comentarios.all %}
                <li>{{ comentario.texto }}</li>
                {% endfor %}
            </ul>
        </li>
        {% endfor %}
    </ul>
</body>
</html>
```

11.3 Actualizar Publicaciones

Paso 1: Actualizar el archivo “views.py”

En el archivo “views.py” de la aplicación “publicaciones”, definimos una vista para actualizar una publicación existente.

```
# views.py
from django.shortcuts import render, get_object_or_404, redirect
from .forms import PublicacionForm
from .models import Publicacion

def actualizar_publicacion(request, pk):
    publicacion = get_object_or_404(Publicacion, pk=pk)
    if request.method == 'POST':
        form = PublicacionForm(request.POST, instance=publicacion)
        if form.is_valid():
            form.save()
```

```
            return redirect('lista_publicaciones')
    else:
        form = PublicacionForm(instance=publicacion)
    return render(request, 'actualizar_publicacion.html', {'form': form})
```

Paso 2: Crear la plantilla para el formulario de actualización

En el directorio "templates" de la aplicación "publicaciones", creamos el archivo "actualizar_publicacion.html" que contendrá el formulario de actualización de publicaciones.

```
<!-- actualizar_publicacion.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Actualizar Publicación</title>
</head>
<body>
    <h1>Actualizar Publicación</h1>
    <form method="post">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Guardar Cambios</button>
    </form>
</body>
</html>
```

Paso 3: Configurar las URLs

En el archivo "urls.py" de la aplicación "publicaciones", agregamos una URL para la vista de actualización de publicaciones.

```
# urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('lista/', views.lista_publicaciones, name='lista_publicaciones'),
    path('crear/', views.crear_publicacion, name='crear_publicacion'),
    path('actualizar/<int:pk>/', views.actualizar_publicacion, name='actualizar_publica
]
```

11.4 Eliminar Publicaciones

Paso 1: Actualizar el archivo "views.py"

En el archivo "views.py" de la aplicación "publicaciones", definimos una vista para eliminar una publicación existente.

```python
# views.py
from django.shortcuts import get_object_or_404, redirect
from .models import Publicacion

def eliminar_publicacion(request, pk):
    publicacion = get_object_or_404(Publicacion, pk=pk)
    if request.method == 'POST':
        publicacion.delete()
        return redirect('lista_publicaciones')
    return render(request, 'eliminar_publicacion.html', {'publicacion': publicacion})
```

Paso 2: Crear la plantilla para confirmar la eliminación

En el directorio "templates" de la aplicación "publicaciones", creamos el archivo "eliminar_publicacion.html" que contendrá la confirmación para eliminar la publicación.

```html
<!-- eliminar_publicacion.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Eliminar Publicación</title>
</head>
<body>
    <h1>Eliminar Publicación</h1>
    <p>¿Estás seguro de que deseas eliminar la publicación "{{ publicacion.titulo }}"?<
    <form method="post">
        {% csrf_token %}
        <button type="submit">Eliminar</button>
    </form>
</body>
</html>
```

Paso 3: Configurar las URLs

En el archivo "urls.py" de la aplicación "publicaciones", agregamos una URL para la vista de eliminación de publicaciones.

```python
# urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('lista/', views.lista_publicaciones, name='lista_publicaciones'),
    path('crear/', views.crear_publicacion, name='crear_publicacion'),
    path('actualizar/<int:pk>/', views.actualizar_publicacion, name='actualizar_publica
    path('eliminar/<int:pk>/', views.eliminar_publicacion, name='eliminar_publicacion')
]
```

11.5 CRUD de Comentarios

El CRUD de comentarios sigue un proceso similar al CRUD de publicaciones. A continuación, se describen los pasos para cada operación: Crear Comentarios

Paso 1: Crear el formulario de Comentario

En el archivo "forms.py" de la aplicación "publicaciones", creamos un formulario para la creación de comentarios.

```
# forms.py
from django import forms
from .models import Comentario

class ComentarioForm(forms.ModelForm):
    class Meta:
        model = Comentario
        fields = ['publicacion', 'autor', 'contenido']
```

Paso 2: Actualizar el archivo "views.py"

En el archivo "views.py" de la aplicación "publicaciones", definimos una vista para crear un nuevo comentario y renderizar el formulario.

```
# views.py
from django.shortcuts import render, redirect
from .forms import ComentarioForm

def crear_comentario(request):
    if request.method == 'POST':
        form = ComentarioForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect('lista_publicaciones')
    else:
        form = ComentarioForm()
    return render(request, 'crear_comentario.html', {'form': form})
```

Paso 3: Crear la plantilla para el formulario de creación

En el directorio "templates" de la aplicación "publicaciones", creamos el archivo "crear_comentario.html" que contendrá el formulario de creación de comentarios.

```
<!-- crear_comentario.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Crear Comentario</title>
</head>
```

```
<body>
    <h1>Crear Nuevo Comentario</h1>
    <form method="post">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Crear</button>
    </form>
</body>
</html>
```

Paso 4: Configurar las URLs

En el archivo "urls.py" de la aplicación "publicaciones", agregamos una URL para la vista de creación de comentarios.

```
# urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('lista/', views.lista_publicaciones, name='lista_publicaciones'),
    path('crear/', views.crear_publicacion, name='crear_publicacion'),
    path('actualizar/<int:pk>/', views.actualizar_publicacion, name='actualizar_publica
    path('eliminar/<int:pk>/', views.eliminar_publicacion, name='eliminar_publicacion')
    path('crear_comentario/', views.crear_comentario, name='crear_comentario'),
]
```

11.6 Leer Comentarios

`Paso 1:` Actualizar el archivo "views.py"

En el archivo "views.py" de la aplicación "publicaciones", definimos una vista para mostrar la lista de comentarios.

```
# views.py
from django.shortcuts import render
from .models import Comentario

def lista_comentarios(request):
    comentarios = Comentario.objects.all()
    return render(request, 'lista_comentarios.html', {'comentarios': comentarios})
```

Paso 2: Actualizar el archivo "lista_comentarios.html"

En la plantilla "lista_comentarios.html", podemos acceder a los comentarios y mostrarlos en una lista.

```
<!-- lista_comentarios.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Lista de Comentarios</title>
</head>
<body>
    <h1>Lista de Comentarios</h1>
    <ul>
        {% for comentario in comentarios %}
        <li>
            <p>Comentario de {{ comentario.autor }} en {{ comentario.publicacion.titulo
            <p>{{ comentario.contenido }}</p>
        </li>
        {% endfor %}
    </ul>
</body>
</html>
```

Paso 3: Configurar las URLs

En el archivo "urls.py" de la aplicación "publicaciones", agregamos una URL para la vista de lista de comentarios.

```
# urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('lista/', views.lista_publicaciones, name='lista_publicaciones'),
    path('crear/', views.crear_publicacion, name='crear_publicacion'),
    path('actualizar/<int:pk>/', views.actualizar_publicacion, name='actualizar_publica
    path('eliminar/<int:pk>/', views.eliminar_publicacion, name='eliminar_publicacion')
    path('crear_comentario/', views.crear_comentario, name='crear_comentario'),
    path('lista_comentarios/', views.lista_comentarios, name='lista_comentarios'),
]
```

11.7 Actualizar Comentarios

Paso 1: Actualizar el archivo "views.py"

En el archivo "views.py" de la aplicación "publicaciones", definimos una vista para actualizar un comentario existente.

```
# views.py
from django.shortcuts import render, get_object_or_404, redirect
from .forms import ComentarioForm
```

```python
from .models import Comentario

def actualizar_comentario(request, pk):
    comentario = get_object_or_404(Comentario, pk=pk)
    if request.method == 'POST':
        form = ComentarioForm(request.POST, instance=comentario)
        if form.is_valid():
            form.save()
            return redirect('lista_comentarios')
    else:
        form = ComentarioForm(instance=comentario)
    return render(request, 'actualizar_comentario.html', {'form': form})
```

Paso 2: Crear la plantilla para el formulario de actualización

En el directorio "templates" de la aplicación "publicaciones", creamos el archivo "actualizar_comentario.html" que contendrá el formulario de actualización de comentarios.

```html
<!-- actualizar_comentario.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Actualizar Comentario</title>
</head>
<body>
    <h1>Actualizar Comentario</h1>
    <form method="post">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Guardar Cambios</button>
    </form>
</body>
</html>
```

Paso 3: Configurar las URLs

En el archivo "urls.py" de la aplicación "publicaciones", agregamos una URL para la vista de actualización de comentarios.

```python
# urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('lista/', views.lista_publicaciones, name='lista_publicaciones'),
    path('crear/', views.crear_publicacion, name='crear_publicacion'),
    path('actualizar/<int:pk>/', views.actualizar_publicacion, name='actualizar_publica
    path('eliminar/<int:pk>/', views.eliminar_publicacion, name='eliminar_publicacion')
```

```
    path('crear_comentario/', views.crear_comentario, name='crear_comentario'),
    path('lista_comentarios/', views.lista_comentarios, name='lista_comentarios'),
    path('actualizar_comentario/<int:pk>/', views.actualizar_comentario, name='actualiz
]
```

11.8 Eliminar Comentarios

Paso 1: Actualizar el archivo "views.py"

En el archivo "views.py" de la aplicación "publicaciones", definimos una vista para eliminar un comentario existente.

```
# views.py
from django.shortcuts import get_object_or_404, redirect
from .models import Comentario

def eliminar_comentario(request, pk):
    comentario = get_object_or_404(Comentario, pk=pk)
    if request.method == 'POST':
        comentario.delete()
        return redirect('lista_comentarios')
    return render(request, 'eliminar_comentario.html', {'comentario': comentario})
```

Paso 2: Crear la plantilla para confirmar la eliminación

En el directorio "templates" de la aplicación "publicaciones", creamos el archivo "eliminar_comentario.html" que contendrá la confirmación para eliminar el comentario.

```
<!-- eliminar_comentario.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Eliminar Comentario</title>
</head>
<body>
    <h1>Eliminar Comentario</h1>
    <p>¿Estás seguro de que deseas eliminar el comentario de "{{ comentario.autor }}" e
    <form method="post">
        {% csrf_token %}
        <button type="submit">Eliminar</button>
    </form>
</body>
</html>
```

Paso 3: Configurar las URLs

En el archivo "urls.py" de la aplicación "publicaciones", agregamos una URL para la vista de eliminación de comentarios.

```
# urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('lista/', views.lista_publicaciones, name='lista_publicaciones'),
    path('crear/', views.crear_publicacion, name='crear_publicacion'),
    path('actualizar/<int:pk>/', views.actualizar_publicacion, name='actualizar_publica
    path('eliminar/<int:pk>/', views.eliminar_publicacion, name='eliminar_publicacion')
    path('crear_comentario/', views.crear_comentario, name='crear_comentario'),
    path('lista_comentarios/', views.lista_comentarios, name='lista_comentarios'),
    path('actualizar_comentario/<int:pk>/', views.actualizar_comentario, name='actualiz
    path('eliminar_comentario/<int:pk>/', views.eliminar_comentario, name='eliminar_com
]
```

11.9 Migrar y Ejecutar el Servidor

Paso 1: Aplicar las migraciones

Después de agregar los modelos y las vistas, es necesario aplicar las migraciones para crear las tablas correspondientes en la base de datos.

Ejecutamos el siguiente comando:

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

Paso 2: Ejecutar el servidor

Finalmente, para ver nuestro proyecto en funcionamiento, ejecutamos el servidor de desarrollo de Django.

Ejecutamos el siguiente comando:

```
python manage.py runserver
```

Con esto, podemos acceder a nuestro sistema CRUD de publicaciones y comentarios en el navegador, utilizando las URLs configuradas en las vistas y templates.

Por ejemplo,

1. Para ver la lista de publicaciones, accedemos a “/lista/”
2. Para crear una nueva publicación, accedemos a “/crear/”.
3. Para ver la lista de comentarios, accedemos a “/lista_comentarios/”
4. Para crear un nuevo comentario, accedemos a “/crear_comentario/”.

12 Correcciones

12.1 Integración de Botones.

Para integrar los botones de actualizar y eliminar en la lista de publicaciones y comentarios, necesitamos realizar algunos cambios en las plantillas y en las vistas.

A continuación, describo los pasos necesarios para cada uno:

12.2 Para las Publicaciones:

Paso 1: Actualizar "lista_publicaciones.html" en el directorio "templates" de la aplicación "publicaciones". Agregar los enlaces para actualizar y eliminar cada publicación.

```
<!-- lista_publicaciones.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Lista de Publicaciones</title>
</head>
<body>
    <h1>Lista de Publicaciones</h1>
    <ul>
        {% for publicacion in publicaciones %}
        <li>
            <h2>{{ publicacion.titulo }}</h2>
            <p>{{ publicacion.contenido }}</p>
            <a href="{% url 'detalle_publicacion' pk=publicacion.pk %}">Ver detalles</a
            <a href="{% url 'actualizar_publicacion' pk=publicacion.pk %}">Actualizar</
            <a href="{% url 'eliminar_publicacion' pk=publicacion.pk %}">Eliminar</a>
            <ul>
                {% for comentario in publicacion.comentarios %}
                <li>{{ comentario.texto }}</li>
                <a href="{% url 'actualizar_comentario' pk=comentario.pk %}">Actualizar
                <a href="{% url 'eliminar_comentario' pk=comentario.pk %}">Eliminar</a>
                {% endfor %}
            </ul>
        </li>
        {% endfor %}
    </ul>
```

```
</body>
</html>
```

Paso 2: Actualizar “views.py” en la aplicación “publicaciones”. Agregar las vistas para actualizar y eliminar las publicaciones.

```
# views.py
from django.shortcuts import render, get_object_or_404, redirect
from .forms import PublicacionForm, ComentarioForm
from .models import Publicacion

from django.shortcuts import render, redirect, get_object_or_404
from .forms import PublicacionForm, ComentarioForm
from .models import Publicacion, Comentario

def lista_publicaciones(request):
    publicaciones = Publicacion.objects.all()
    return render(request, 'lista_publicaciones.html', {'publicaciones': publicaciones}

def crear_publicacion(request):
    if request.method == 'POST':
        form = PublicacionForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect('lista_publicaciones')
    else:
        form = PublicacionForm()
    return render(request, 'crear_publicacion.html', {'form': form})

def detalle_publicacion(request, pk):
    publicacion = get_object_or_404(Publicacion, pk=pk)
    comentarios = Comentario.objects.filter(publicacion=publicacion)

    if request.method == 'POST':
        comentario_form = ComentarioForm(request.POST)
        if comentario_form.is_valid():
            comentario = comentario_form.save(commit=False)
            comentario.publicacion = publicacion
            comentario.save()
            return redirect('detalle_publicacion', pk=pk)
    else:
        comentario_form = ComentarioForm()

    context = {
        'publicacion': publicacion,
        'comentarios': comentarios,
        'comentario_form': comentario_form,
```

```
    }
    return render(request, 'detalle_publicacion.html', context)

def actualizar_publicacion(request, pk):
    publicacion = get_object_or_404(Publicacion, pk=pk)
    if request.method == 'POST':
        form = PublicacionForm(request.POST, instance=publicacion)
        if form.is_valid():
            form.save()
            return redirect('lista_publicaciones')
    else:
        form = PublicacionForm(instance=publicacion)
    return render(request, 'crear_publicacion.html', {'form': form})

def eliminar_publicacion(request, pk):
    publicacion = get_object_or_404(Publicacion, pk=pk)
    if request.method == 'POST':
        publicacion.delete()
        return redirect('lista_publicaciones')
    return render(request, 'eliminar_publicacion.html', {'publicacion': publicacion})
```

Paso 3: Actualizar "urls.py" en la aplicación "publicaciones". Agregar las URLs correspondientes para las vistas de actualizar y eliminar publicaciones.

```
# urls.py
from django.urls import path
from . import views

urlpatterns = [
    # Rutas de Publicaciones
    path('lista/', views.lista_publicaciones, name='lista_publicaciones'),
    path('crear/', views.crear_publicacion, name='crear_publicacion'),
    path('detalle/<int:pk>/', views.detalle_publicacion, name='detalle_publicacion'),
    path('actualizar/<int:pk>/', views.actualizar_publicacion, name='actualizar_publica
    path('eliminar/<int:pk>/', views.eliminar_publicacion, name='eliminar_publicacion')

    # Rutas de Comentarios
    path('<int:pk>/actualizar_comentario/', views.actualizar_comentario, name='actualiz
    path('<int:pk>/eliminar_comentario/', views.eliminar_comentario, name='eliminar_com
]
```

12.3 Para los Comentarios:

El proceso es similar al de las publicaciones, solo que debemos aplicarlo para los comentarios.

Paso 1: Actualizar “lista_publicaciones.html” en el directorio “templates” de la aplicación “publicaciones”. Agregar los enlaces para actualizar y eliminar cada comentario.

```
<!-- lista_publicaciones.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Lista de Publicaciones</title>
</head>
<body>
    <h1>Lista de Publicaciones</h1>
    <ul>
        {% for publicacion in publicaciones %}
        <li>
            <h2>{{ publicacion.titulo }}</h2>
            <p>{{ publicacion.contenido }}</p>
            <a href="{% url 'detalle_publicacion' pk=publicacion.pk %}">Ver detalles</a
            <a href="{% url 'actualizar_publicacion' pk=publicacion.pk %}">Actualizar</
            <a href="{% url 'eliminar_publicacion' pk=publicacion.pk %}">Eliminar</a>
            <ul>
                {% for comentario in publicacion.comentarios %}
                <li>{{ comentario.texto }}</li>
                <a href="{% url 'actualizar_comentario' pk=comentario.pk %}">Actualizar
                <a href="{% url 'eliminar_comentario' pk=comentario.pk %}">Eliminar</a>
                {% endfor %}
            </ul>
        </li>
        {% endfor %}
    </ul>
</body>
</html>
```

Tambien es necesario la creación de un nuevo template llamado detalle_publicacion.html en el directorio “templates” de la aplicación “publicaciones”. Agregar los enlaces para actualizar y eliminar cada comentario.

```
<!-- detalle_publicacion.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Detalle de Publicación</title>
</head>
<body>
    <h1>{{ publicacion.titulo }}</h1>
    <p>{{ publicacion.contenido }}</p>
    <h3>Comentarios:</h3>
    <ul>
        {% for comentario in comentarios %}
```

```
        <li>{{ comentario.contenido }}</li>
        <a href="{% url 'actualizar_comentario' comentario.pk %}">Actualizar Comentario
        <a href="{% url 'eliminar_comentario' comentario.pk %}">Eliminar Comentario</a>
        {% endfor %}
    </ul>

    <a href="{% url 'actualizar_publicacion' publicacion.pk %}">Actualizar Publicación<
    <a href="{% url 'eliminar_publicacion' publicacion.pk %}">Eliminar Publicación</a>

    <!-- Agregar formulario para agregar comentario -->
    <h3>Agregar Comentario:</h3>
    <form method="post">
        {% csrf_token %}
        {{ comentario_form.as_p }}
        <button type="submit">Enviar Comentario</button>
    </form>
</body>
</html>
```

Y finalmente actualizar el archivo actualizar_comentario.html en el directorio "templates" de la aplicación "publicaciones". Agregar los enlaces para actualizar y eliminar cada comentario.

```
<!-- actualizar_comentario.html -->
<!DOCTYPE html>
<head>
    <title>Actualizar Comentario</title>
</head>
<body>
    <h1>Actualizar Comentario</h1>
    <form method="post">
        {% csrf_token %}
        {{ comentario_form.as_p }}
        <button type="submit">Guardar Cambios</button>
    </form>
</body>
</html>
```

Paso 2: Actualizar "views.py" en la aplicación "publicaciones". Agregar las vistas para actualizar y eliminar los comentarios.

```
# views.py
from django.shortcuts import render, get_object_or_404, redirect
from .forms import PublicacionForm, ComentarioForm
from .models import Publicacion, Comentario

def lista_publicaciones(request):
```

```
    publicaciones = Publicacion.objects.all()
    return render(request, 'lista_publicaciones.html', {'publicaciones': publicaciones}

def crear_publicacion(request):
    # Código existente

def detalle_publicacion(request, pk):
    # Código existente

def actualizar_publicacion(request, pk):
    # Código existente

def eliminar_publicacion(request, pk):
    # Código existente

def crear_comentario(request):
    if request.method == 'POST':
        form = ComentarioForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect('lista_publicaciones')
    else:
        form = ComentarioForm()
    return render(request, 'crear_comentario.html', {'form': form})

def lista_comentarios(request):
    comentarios = Comentario.objects.all()
    return render(request, 'lista_comentarios.html', {'comentarios': comentarios})

def actualizar_comentario(request, pk):
    comentario = get_object_or_404(Comentario, pk=pk)

    if request.method == 'POST':
        comentario_form = ComentarioForm(request.POST, instance=comentario)
        if comentario_form.is_valid():
            comentario_form.save()
            return redirect('detalle_publicacion', pk=comentario.publicacion.pk)
    else:
        comentario_form = ComentarioForm(instance=comentario)

    context = {
        'comentario_form': comentario_form,
    }
    return render(request, 'actualizar_comentario.html', context)

def eliminar_comentario(request, pk):
    comentario = get_object_or_404(Comentario, pk=pk)
```

```
    publicacion_pk = comentario.publicacion.pk
    comentario.delete()
    return redirect('detalle_publicacion', pk=publicacion_pk)
```

Paso 3: Actualizar "urls.py" en la aplicación "publicaciones". Agregar las URLs correspondientes para las vistas de actualizar y eliminar comentarios.

```
# urls.py
from django.urls import path
from . import views

urlpatterns = [
    # Rutas de Comentarios
    path('<int:pk>/actualizar_comentario/', views.actualizar_comentario, name='actualiz
    path('<int:pk>/eliminar_comentario/', views.eliminar_comentario, name='eliminar_com
]
```

12.4 Corregir el Modelo

Finalmente para corregir el modelo y hacer que el campo autor herede los datos de **User** de Django, podemos usar un campo **ForeignKey** que apunte al modelo User.

Esto nos permitirá asociar cada publicación y comentario a un usuario específico. Aquí está el código corregido:

```
from django.contrib.auth.models import User
from django.db import models

class Publicacion(models.Model):
    titulo = models.CharField(max_length=200)
    contenido = models.TextField()
    fecha_publicacion = models.DateTimeField(auto_now_add=True)
    autor = models.ForeignKey(User, on_delete=models.CASCADE)

    def __str__(self):
        return self.titulo

class Comentario(models.Model):
    publicacion = models.ForeignKey(Publicacion, on_delete=models.CASCADE)
    autor = models.ForeignKey(User, on_delete=models.CASCADE)
    contenido = models.TextField()
    fecha_publicacion = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return f"Comentario de {self.autor.username} en {self.publicacion.titulo}"
```

En este código, hemos modificado el campo autor en ambos modelos para que sea un **ForeignKey** que apunta al modelo **User de Django**.

Con esto, cada publicación y comentario estará asociado a un usuario registrado en el sistema.

El argumento **on_delete=models.CASCADE** en el campo autor de Comentario asegura que si un usuario es eliminado, también se eliminarán todos sus comentarios relacionados, pero ten en cuenta que esto es opcional y depende de la lógica de negocio que desees implementar.

12.5 Backend: Construcción de una API con Django Rest Framework

En esta sección, dividiremos todo lo que hemos realizado hasta ahora en una API utilizando Django Rest Framework (DRF). Esto nos permitirá exponer nuestros modelos (Publicacion y Comentario) como puntos finales (endpoints) para realizar operaciones CRUD (Crear, Leer, Actualizar y Eliminar) a través de peticiones HTTP.

Paso 1: Instalar Django Rest Framework

Primero, debemos instalar Django Rest Framework en nuestro entorno virtual. Ejecuta el siguiente comando:

```
pip install djangorestframework
```

Paso 2: Configurar Django Rest Framework en el Proyecto

En el archivo settings.py del proyecto, agrega 'rest_framework' a la lista de aplicaciones instaladas:

```
INSTALLED_APPS = [
    # Otras aplicaciones...
    'rest_framework',
]
```

Paso 3: Serializadores

En DRF, los serializadores se utilizan para convertir nuestros modelos de Django en formatos JSON y viceversa. Vamos a crear los serializadores para los modelos Publicacion y Comentario en un archivo serializers.py dentro de la aplicación publicaciones.

```
# serializers.py
from rest_framework import serializers
from .models import Publicacion, Comentario

class ComentarioSerializer(serializers.ModelSerializer):
    class Meta:
        model = Comentario
```

```
        fields = '__all__'

class PublicacionSerializer(serializers.ModelSerializer):
    comentarios = ComentarioSerializer(many=True, read_only=True)

    class Meta:
        model = Publicacion
        fields = '__all__'
```

En este código, creamos dos serializadores, ComentarioSerializer y PublicacionSerializer, que utilizan el modelo correspondiente y definen los campos que queremos exponer en nuestra API. En el caso de la publicación, utilizamos comentarios para mostrar los comentarios relacionados.

Paso 4: Vistas

Vamos a modificar nuestras vistas para utilizar los serializadores y convertir nuestros datos en formato JSON. En el archivo views.py de la aplicación publicaciones, actualiza el contenido de las vistas de la siguiente manera:

```
# views.py
from rest_framework import generics
from .models import Publicacion, Comentario
from .serializers import PublicacionSerializer, ComentarioSerializer

class ListaPublicaciones(generics.ListCreateAPIView):
    queryset = Publicacion.objects.all()
    serializer_class = PublicacionSerializer

class DetallePublicacion(generics.RetrieveUpdateDestroyAPIView):
    queryset = Publicacion.objects.all()
    serializer_class = PublicacionSerializer

class ListaComentarios(generics.ListCreateAPIView):
    queryset = Comentario.objects.all()
    serializer_class = ComentarioSerializer

class DetalleComentario(generics.RetrieveUpdateDestroyAPIView):
    queryset = Comentario.objects.all()
    serializer_class = ComentarioSerializer
```

Paso 5: URLs

Ahora, vamos a configurar las URLs de nuestra API en el archivo urls.py de la aplicación publicaciones.

```
# urls.py
from django.urls import path
from . import views
```

```
urlpatterns = [
    path('publicaciones/', views.ListaPublicaciones.as_view(), name='lista_publicacione
    path('publicaciones/<int:pk>/', views.DetallePublicacion.as_view(), name='detalle_p
    path('comentarios/', views.ListaComentarios.as_view(), name='lista_comentarios'),
    path('comentarios/<int:pk>/', views.DetalleComentario.as_view(), name='detalle_come
]
```

En este código, configuramos las URLs de nuestras vistas utilizando las vistas basadas en clases proporcionadas por DRF. Creamos puntos finales (endpoints) para listar, crear, ver, actualizar y eliminar publicaciones y comentarios.

¡Hemos construido una API básica para nuestro proyecto "Blog" utilizando Django Rest Framework!

Ahora podemos realizar operaciones CRUD a través de las peticiones HTTP en nuestros modelos de Publicacion y Comentario.

12.6 Documentación de las API

Para documentar las API, podemos utilizar la herramienta **drf-yasg**. La información la vamos a obtener de la documentación oficial en el siguiente link

Documentación de drf-yasg

Paso 1: Instalar drf-yasg

Primero, debemos instalar drf-yasg en nuestro entorno virtual. Ejecuta el siguiente comando:

```
pip install -U drf-yasg
```

Paso 2: Configurar drf-yasg en el Proyecto

En el archivo settings.py del proyecto, agrega 'drf_yasg' a la lista de aplicaciones instaladas:

```
INSTALLED_APPS = [
    # Otras aplicaciones...
    'drf_yasg',
]
```

Paso 3: Configurar drf-yasg en el archivo urls.py

En el archivo urls.py del proyecto, agrega las siguientes líneas de código:

```
# urls.py
...
from django.urls import re_path
from rest_framework import permissions
```

```
from drf_yasg.views import get_schema_view
from drf_yasg import openapi

...

schema_view = get_schema_view(
    openapi.Info(
        title="Snippets API",
        default_version='v1',
        description="Test description",
        terms_of_service="https://www.google.com/policies/terms/",
        contact=openapi.Contact(email="contact@snippets.local"),
        license=openapi.License(name="BSD License"),
    ),
    public=True,
    permission_classes=(permissions.AllowAny,),
)

urlpatterns = [
    path('swagger<format>/', schema_view.without_ui(cache_timeout=0), name='schema-jso
    path('swagger/', schema_view.with_ui('swagger', cache_timeout=0), name='schema-swag
    path('redoc/', schema_view.with_ui('redoc', cache_timeout=0), name='schema-redoc'),
    ...
]
```

Paso 4: Agregar la variable autenticacion_classes.

En el archivo url.py del proyecto agregar la variable authentication_classes para que no nos pida autenticación para poder ver la documentación de las API.

```
# urls.py
...
authentication_classes = []
...
```

Paso 5: Ejecutar el servidor

Ejecuta el servidor y en el navegador ingresa a http://localhost:8000/swagger o http://localhost:8000/redoc

```
python manage.py runserver
```

Tambien puedes probar los plugins de VSCode RappidApi Client o Thunder Client para poder ver la documentación de las API desde el editor de código.

En la siguiente sección, construiremos el frontend para consumir esta API.

13 FRONTEND: Herencia de Plantillas con Django.

13.1 Herencia de Plantillas

Para iniciar con el tema de herencia de plantillas de django, es necesario generar un archivo base.html que será nuestra plantilla base para la generación de las demás plantillas.

En esta plantilla base incluiremos una barra de navegación con accesos a archivos estáticos de **Home**, **Contacto**, **Quienes somos** y **Crear Publicación**.

- Abre el archivo base.html en el directorio templates de tu proyecto.
- Agrega el siguiente código para definir la barra de navegación:

```
<!DOCTYPE html>
<html>
<head>
    <title>{% block title %}Mi Blog{% endblock %}</title>
    <!-- Agregar enlaces a Bootstrap CSS -->
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/css/bo
</head>
<body>
    <nav class="navbar navbar-expand navbar-dark bg-dark">
        <div class="container">
            <a class="navbar-brand" href="{% url 'home' %}">Mi Blog</a>
            <ul class="navbar-nav ml-auto">
                <li class="nav-item">
                    <a class="nav-link" href="{% url 'home' %}">Home</a>
                </li>
                <li class="nav-item">
                    <a class="nav-link" href="{% url 'home' %}">Creditos</a>
                </li>
                <li class="nav-item">
                    <a class="nav-link" href="{% url 'contacto' %}">Contacto</a>
                </li>
                <li class="nav-item">
                    <a class="nav-link" href="{% url 'quienes_somos' %}">Quienes Somos<
                </li>
                <li class="nav-item">
                    <a class="nav-link" href="{% url 'crear_publicacion' %}">Crear</a>
                </li>
```

```
            </ul>
        </div>
    </nav>

    <div class="container mt-4">
        {% block content %}
        {% endblock %}
    </div>

    <!-- Agregar enlaces a Bootstrap JS y jQuery -->
    <script src="https://code.jquery.com/jquery-3.5.1.slim.min.js"></script>
    <script src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.9.1/dist/umd/popper.min.
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"><
</body>
</html>
```

En este código, hemos definido una etiqueta

que contiene enlaces para “Home”, “Contacto”, “Quienes Somos” y “Crear”. Los enlaces utilizan las etiquetas {% url %} de Django para generar las URLs correspondientes a cada vista.

- En cada plantilla que extienda de base.html, asegúrate de agregar bloques de contenido para el título y el contenido específico de cada página:

Por ejemplo, en la plantilla lista_publicaciones.html:

```
{% extends 'base.html' %}

{% block title %}Lista de Publicaciones{% endblock %}

{% block content %}
<h1>Lista de Publicaciones</h1>
<!-- Contenido de la lista de publicaciones -->
{% endblock %}
```

En la plantilla detalle_publicacion.html:

```
{% extends 'base.html' %}

{% block title %}Detalle de Publicación{% endblock %}

{% block content %}
<h1>Detalle de Publicación</h1>
<!-- Contenido del detalle de la publicación -->
{% endblock %}
```

Y así sucesivamente para las demás plantillas que extiendan de base.html.

13.2 Modificación de Views.

Vista y URL para la página “Home”:

```
# views.py
from django.shortcuts import render

def home_view(request):
    return render(request, 'home.html')

# urls.py del proyecto
from django.urls import path
from . import views

urlpatterns = [
    path('home/', views.home_view, name='home'),
    # otras URLs de tu proyecto
]
```

- Vista y URL para la página “Contacto”:

```
# views.py
from django.shortcuts import render

def contacto_view(request):
    return render(request, 'contacto.html')

# urls.py del proyecto
from django.urls import path
from . import views

urlpatterns = [
    # otras URLs de tu proyecto
    path('contacto/', views.contacto_view, name='contacto'),
]
```

Vista y URL para la página “Quienes Somos”:

```
# views.py
from django.shortcuts import render

def quienes_somos_view(request):
    return render(request, 'quienes_somos.html')

# urls.py del proyecto
from django.urls import path
from . import views
```

```
urlpatterns = [
    # otras URLs de tu proyecto
    path('quienes_somos/', views.quienes_somos_view, name='quienes_somos'),
]
```

- Vista y URL para la página “Crear”:

```
# views.py
from django.shortcuts import render

def crear_view(request):
    return render(request, 'crear.html')

# urls.py del proyecto
from django.urls import path
from . import views

urlpatterns = [
    # otras URLs de tu proyecto
    path('crear/', views.crear_view, name='crear_publicacion'),
]
```

13.3 Bootstrap en base.html

Para incluir Bootstrap en el archivo base.html, debes agregar las etiquetas y

```
<script src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.0.7/dist/umd/popper.min.js"></
<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"></sc

Este código incluye las hojas de estilos y scripts de Bootstrap desde sus respectivos CDN

Si prefieres tener los archivos locales en tu proyecto, simplemente descárgalos desde la

Con esta modificación, tu archivo base.html utilizará Bootstrap y tendrás una barra de na

## Inclusión de Herencia de Plantillas en las demás plantillas.

1. Template de home.html.

``` html
{% extends 'base.html' %}

{% block title %}Home - Mi Blog{% endblock %}

{% block content %}
<h1>Bienvenido a Mi Blog</h1>
```
```

```
<p>Este es el contenido de la página de inicio.</p>
{% endblock %}
```

2. Template de contacto.html.

```
{% extends 'base.html' %}

{% block title %}Contacto - Mi Blog{% endblock %}

{% block content %}
<h1>Contacto</h1>
<p>Información de contacto y formulario de contacto.</p>
{% endblock %}
```

3. Template de quienes_somos.html.

```
{% extends 'base.html' %}

{% block title %}Quienes Somos - Mi Blog{% endblock %}

{% block content %}
<h1>Quienes Somos</h1>
<p>Información sobre nosotros y nuestra historia.</p>
{% endblock %}
```

En cada uno de estos templates, utilizamos la etiqueta {% extends 'base.html' %} para indicar que estamos heredando el contenido y la estructura del archivo base.html. Luego, en el bloque {% block content %}, agregamos el contenido específico de cada página.

Cada template tiene su título personalizado en el bloque {% block title %}.

Cuando se acceda a las URLs correspondientes a cada template, se mostrará el contenido específico de cada página dentro del contenedor proporcionado por el archivo base.html, y la barra de navegación con los enlaces a Home, Contacto, Quienes Somos y Crear se mantendrá visible en todas las páginas gracias a la herencia.

13.4 Agregando bootstrap a los templates.

ara agregar Bootstrap a las plantillas anteriores, necesitaremos incluir los enlaces a los archivos CSS y JS de Bootstrap en el archivo base.html. Además, podemos utilizar las clases de Bootstrap para mejorar la apariencia de los elementos en cada página.

A continuación, se muestran las plantillas actualizadas con Bootstrap:

1. Template de base.html

```
<!DOCTYPE html>
<html>
<head>
```

```
    <title>{% block title %}Mi Blog{% endblock %}</title>
    <!-- Agregar enlaces a Bootstrap CSS -->
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/css/bo
</head>
<body>
    <nav class="navbar navbar-expand navbar-dark bg-dark">
        <div class="container">
            <a class="navbar-brand" href="{% url 'home' %}">Mi Blog</a>
            <ul class="navbar-nav ml-auto">
                <li class="nav-item">
                    <a class="nav-link" href="{% url 'home' %}">Home</a>
                </li>
                <li class="nav-item">
                    <a class="nav-link" href="{% url 'contacto' %}">Contacto</a>
                </li>
                <li class="nav-item">
                    <a class="nav-link" href="{% url 'quienes_somos' %}">Quienes Somos<
                </li>
                <li class="nav-item">
                    <a class="nav-link" href="{% url 'crear_publicacion' %}">Crear</a>
                </li>
            </ul>
        </div>
    </nav>

    <div class="container mt-4">
        {% block content %}
        {% endblock %}
    </div>

    <!-- Agregar enlaces a Bootstrap JS y jQuery -->
    <script src="https://code.jquery.com/jquery-3.5.1.slim.min.js"></script>
    <script src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.9.1/dist/umd/popper.min.
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"><
</body>
</html>
```

2. Template de home.html.

```
{% extends 'base.html' %}

{% block title %}Home - Mi Blog{% endblock %}

{% block content %}
<div class="jumbotron">
    <h1 class="display-4">Bienvenido a Mi Blog</h1>
    <p class="lead">Este es el contenido de la página de inicio.</p>
```

```
</div>
{% endblock %}
```

3. Template de contacto.html.

```
{% extends 'base.html' %}

{% block title %}Contacto - Mi Blog{% endblock %}

{% block content %}
<div class="row">
    <div class="col-md-6 mx-auto">
        <h1>Contacto</h1>
        <p>Información de contacto y formulario de contacto.</p>
    </div>
</div>
{% endblock %}
```

4. Template de quienes_somos.html

```
{% extends 'base.html' %}

{% block title %}Quienes Somos - Mi Blog{% endblock %}

{% block content %}
<div class="container">
    <h1>Quienes Somos</h1>
    <p>Información sobre nosotros y nuestra historia.</p>
</div>
{% endblock %}
```

En estas plantillas, hemos utilizado las clases de Bootstrap para dar estilo a los elementos. Por ejemplo, en el archivo base.html, hemos utilizado la clase “navbar” para crear la barra de navegación y la clase “jumbotron” para resaltar el contenido de la página de inicio. Además, en los templates de contacto.html y quienes_somos.html, hemos utilizado la clase “container” para centrar el contenido y la clase “row” para crear una fila que contenga el contenido.

14 Reactjs

14.1 Frontend con Reactjs

Para crear un proyecto desde cero en ReactJS que consuma los datos de la API creada con Django Rest Framework, sigue estos pasos:

Paso 1: Verificar versiones de Node.js y npm.

Antes de comenzar, asegúrate de tener Node.js y npm instalados en tu sistema. Abre la terminal y ejecuta los siguientes comandos para verificar las versiones:

```
node -v
npm -v
```

Paso 2: Crear un nuevo proyecto de React

Crea un nuevo proyecto de React utilizando create-react-app. Ejecuta el siguiente comando en la terminal:

```
npx create-react-app frontend
```

Esto creará una nueva carpeta llamada "frontend" con una estructura de proyecto de React preconfigurada.

Paso 3: Instalar axios

En la carpeta "frontend", instala la librería axios para realizar peticiones HTTP a la API de Django. Ejecuta el siguiente comando en la terminal:

```
cd frontend
npm install axios
```

Paso 4: Crear el componente ListaPublicaciones.js

Crea el componente **ListaPublicaciones.js** en la carpeta "src/components" con el siguiente contenido:

```
import React, { useState, useEffect } from 'react';
import axios from 'axios';

const ListaPublicaciones = () => {
  const [publicaciones, setPublicaciones] = useState([]);
```

```
  useEffect(() => {
    const fetchPublicaciones = async () => {
      try {
        const response = await axios.get('http://localhost:8000/publicaciones/');
        setPublicaciones(response.data.results);
      } catch (error) {
        console.error('Error al obtener las publicaciones:', error);
      }
    };

    fetchPublicaciones();
  }, []);

  return (
    <div>
      <h1>Lista de Publicaciones</h1>
      <ul>
        {publicaciones.map((publicacion) => (
          <li key={publicacion.id}>
            <h2>{publicacion.titulo}</h2>
            <p>{publicacion.contenido}</p>
          </li>
        ))}
      </ul>
    </div>
  );
};

export default ListaPublicaciones;
```

Paso 5: Crear el componente CrearPublicacion.js

Crea el componente CrearPublicacion.js en la carpeta "src/components" con el siguiente contenido:

```
import React from 'react';
import axios from 'axios';

const CrearPublicacion = () => {
  const [titulo, setTitulo] = React.useState('');
  const [contenido, setContenido] = React.useState('');

  // Replace "1" with the ID of the author you want to associate with the publicación
  const autorId = 1;

  const handleCrear = async () => {
    try {
      await axios.post('http://localhost:8000/publicaciones/', {
```

```
        titulo,
        contenido,
        autor: autorId,
      });
      alert('Publicación creada correctamente.');
      setTitulo('');
      setContenido('');
    } catch (error) {
      console.error('Error al crear la publicación:', error);
    }
  };

  return (
    <div>
      <h1>Crear Publicación</h1>
      <label>
        Título:
        <input type="text" value={titulo} onChange={(e) => setTitulo(e.target.value)} /
      </label>
      <br />
      <label>
        Contenido:
        <textarea value={contenido} onChange={(e) => setContenido(e.target.value)} />
      </label>
      <br />
      <button onClick={handleCrear}>Crear</button>
    </div>
  );
};

export default CrearPublicacion;
```

Paso 6: Crear el componente ActualizarPublicacion.js

Crea el componente ActualizarPublicacion.js en la carpeta "src/components" con el siguiente contenido:

```
import React, { useState, useEffect } from 'react';
import axios from 'axios';

const ActualizarPublicacion = () => {
  const [titulo, setTitulo] = useState('');
  const [contenido, setContenido] = useState('');
  const [autor, setAutor] = useState('');

  // Replace "3" with the ID of the publicación you want to update
  const publicacionId = 3;
```

```
useEffect(() => {
  const fetchPublicacion = async () => {
    try {
      const response = await axios.get(`http://localhost:8000/publicaciones/${publica
      const { titulo, contenido, autor } = response.data;
      setTitulo(titulo);
      setContenido(contenido);
      setAutor(autor);
    } catch (error) {
      console.error('Error al obtener la publicación:', error);
    }
  };

  fetchPublicacion();
}, [publicacionId]);

const handleActualizar = async () => {
  try {
    await axios.put(`http://localhost:8000/publicaciones/${publicacionId}/`, {
      titulo,
      contenido,
      autor,
    });
    alert('Publicación actualizada correctamente.');
  } catch (error) {
    console.error('Error al actualizar la publicación:', error);
  }
};

return (
  <div>
    <h1>Actualizar Publicación</h1>
    <label>
      Título:
      <input type="text" value={titulo} onChange={(e) => setTitulo(e.target.value)} /
    </label>
    <br />
    <label>
      Contenido:
      <textarea value={contenido} onChange={(e) => setContenido(e.target.value)} />
    </label>
    <br />
    <label>
      Autor:
      <input type="number" value={autor} onChange={(e) => setAutor(Number(e.target.va
    </label>
    <br />
```

```
      <button onClick={handleActualizar}>Actualizar</button>
    </div>
  );
};

export default ActualizarPublicacion;
```

Paso 7: Crear el componente EliminarPublicacion.js

Crea el componente EliminarPublicacion.js en la carpeta "src/components" con el siguiente contenido:

```
import React, { useEffect } from 'react';
import axios from 'axios';

const EliminarPublicacion = () => {
  // Replace "2" with the ID of the publicación you want to delete
  const publicacionId = 2;

  useEffect(() => {
    const fetchPublicacion = async () => {
      try {
        const response = await axios.get(`http://localhost:8000/publicaciones/${publica
        const { titulo, contenido, autor } = response.data;
        console.log('Publicación a eliminar:', { titulo, contenido, autor });
      } catch (error) {
        console.error('Error al obtener la publicación:', error);
      }
    };

    fetchPublicacion();
  }, [publicacionId]);

  const handleEliminar = async () => {
    try {
      await axios.delete(`http://localhost:8000/publicaciones/${publicacionId}/`);
      alert('Publicación eliminada correctamente.');
    } catch (error) {
      console.error('Error al eliminar la publicación:', error);
    }
  };

  return (
    <div>
      <h1>Eliminar Publicación</h1>
      <p>¿Estás seguro de que deseas eliminar esta publicación?</p>
      <button onClick={handleEliminar}>Eliminar</button>
    </div>
```

```
  );
};

export default EliminarPublicacion;
```

Paso 8: Crear el componente Home.js

Crea el componente Home.js en la carpeta "src/components" con el siguiente contenido:

```
import React from 'react';

const Home = () => {
  return (
    <div>
      <h1>Bienvenido a la página de inicio</h1>
    </div>
  );
};

export default Home;
```

Paso 9: Crear el componente NavBar.js

Crea el componente NavBar.js en la carpeta "src/components" con el siguiente contenido:

```
import React from 'react';
import { Link } from 'react-router-dom';

const NavBar = () => {
  return (
    <nav className="navbar navbar-expand navbar-dark bg-dark">
      <div className="container">
        <Link className="navbar-brand" to="/">Mi Blog</Link>
        <ul className="navbar-nav ml-auto">
          <li className="nav-item">
            <Link className="nav-link" to="/">Home</Link>
          </li>
          <li className="nav-item">
            <Link className="nav-link" to="/crear">Crear</Link>
          </li>
        </ul>
      </div>
    </nav>
  );
};

export default NavBar;
```

Paso 10: Actualizar el archivo App.js

Actualiza el archivo App.js en la carpeta "src" con el siguiente contenido:

```
import React from 'react';
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';
import NavBar from './components/NavBar';
import ListaPublicaciones from './components/ListaPublicaciones';
import CrearPublicacion from './components/CrearPublicacion';
import DetallePublicacion from './components/DetallePublicacion';
import ActualizarPublicacion from './components/ActualizarPublicacion';
import EliminarPublicacion from './components/EliminarPublicacion';

const App = () => {
  return (
    <Router>
      <div>
        <NavBar />
        <Routes>
          <Route path="/" element={<ListaPublicaciones />} />
          <Route path="/crear" element={<CrearPublicacion />} />
          <Route path="/publicaciones/:id" element={<DetallePublicacion />} />
          <Route path="/publicaciones/:id/actualizar" element={<ActualizarPublicacion /
          <Route path="/publicaciones/:id/eliminar" element={<EliminarPublicacion />} /
        </Routes>
      </div>
    </Router>
  );
};

export default App;
```

Paso 11: Estilos CSS con los archivos index.css y App.css

En una aplicación React, los estilos CSS se aplican a los componentes para mejorar la apariencia y el diseño de la interfaz de usuario. Por defecto, React crea dos archivos para estilos: index.css y App.css.

index.css: Este archivo contiene estilos globales que se aplican a toda la aplicación.

App.css: Este archivo contiene estilos específicos para el componente principal de la aplicación (en este caso, el componente App.js).

A continuación, se muestra cómo puedes usar estos archivos CSS en tu aplicación React:

14.1.1 1. index.css

En el directorio src, crea un archivo llamado index.css y agrega los estilos globales que deseas aplicar a toda la aplicación. Por ejemplo:

```css
/* index.css */
body {
  font-family: Arial, sans-serif;
  margin: 0;
  padding: 0;
}

.container {
  max-width: 1200px;
  margin: 0 auto;
}
```

14.1.2 2. App.css

En el directorio src, ya deberías tener un archivo llamado App.css. Puedes agregar estilos específicos para el componente App.js aquí. Por ejemplo:

```css
/* App.css */
.header {
  background-color: #007bff;
  color: #ffffff;
  padding: 1rem;
  text-align: center;
}

.nav-link {
  color: #ffffff;
  text-decoration: none;
  margin-right: 1rem;
}

.nav-link:hover {
  color: #0056b3;
}
```

14.1.3 3. Importar los archivos CSS en App.js

En el componente App.js, importa los archivos CSS para que los estilos se apliquen a la aplicación. Debes importar index.css al principio del archivo src/index.js, y App.css en App.js:

```javascript
// src/index.js
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
```

```
import './index.css'; // Importar index.css aquí

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);
```

```
// src/App.js
import React from 'react';
import './App.css'; // Importar App.css aquí

function App() {
  // Código de la aplicación
  return (
    <div className="container">
      <header className="header">
        <nav>
          <a className="nav-link" href="/">Home</a>
          <a className="nav-link" href="/contacto">Contacto</a>
          <a className="nav-link" href="/quienes_somos">Quienes Somos</a>
          <a className="nav-link" href="/crear_publicacion">Crear</a>
        </nav>
      </header>
      {/* Resto del contenido de la aplicación */}
    </div>
  );
}

export default App;
```

Con esto, tus estilos CSS definidos en index.css se aplicarán a toda la aplicación, y los estilos definidos en App.css se aplicarán específicamente al componente App.js. Recuerda adaptar los estilos a tus necesidades y preferencias.

Paso 12: Importar estilos CSS

En el archivo **index.js** en la carpeta **"src"**, importa el archivo **styles.css** para aplicar los estilos personalizados:

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

ReactDOM.render(
```

```
    <React.StrictMode>
      <App />
    </React.StrictMode>,
    document.getElementById('root')
);

reportWebVitals();
```

Con esto, has creado un proyecto de React independiente que consume los datos de la API creada con Django Rest Framework. Puedes ejecutar el proyecto de React con el siguiente comando:

```
npm start
```

Luego, podrás ver la aplicación en el navegador accediendo a http://localhost:3000/.

Los componentes ListaPublicaciones, CrearPublicacion, ActualizarPublicacion y EliminarPublicacion estarán disponibles en sus respectivas rutas y consumirán datos de la API de Django.

14.2 Inconvenientes

En caso de tener inconvenientes con las peticiones por parte de la aplicacion de React, se debe agregar la siguiente linea en el archivo settings.py de la carpeta “backend”:

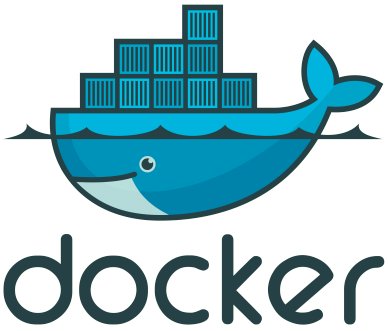
```
CORS_ORIGIN_ALLOW_ALL = True
```

Part V

Docker

15 Módulo 5:Docker.

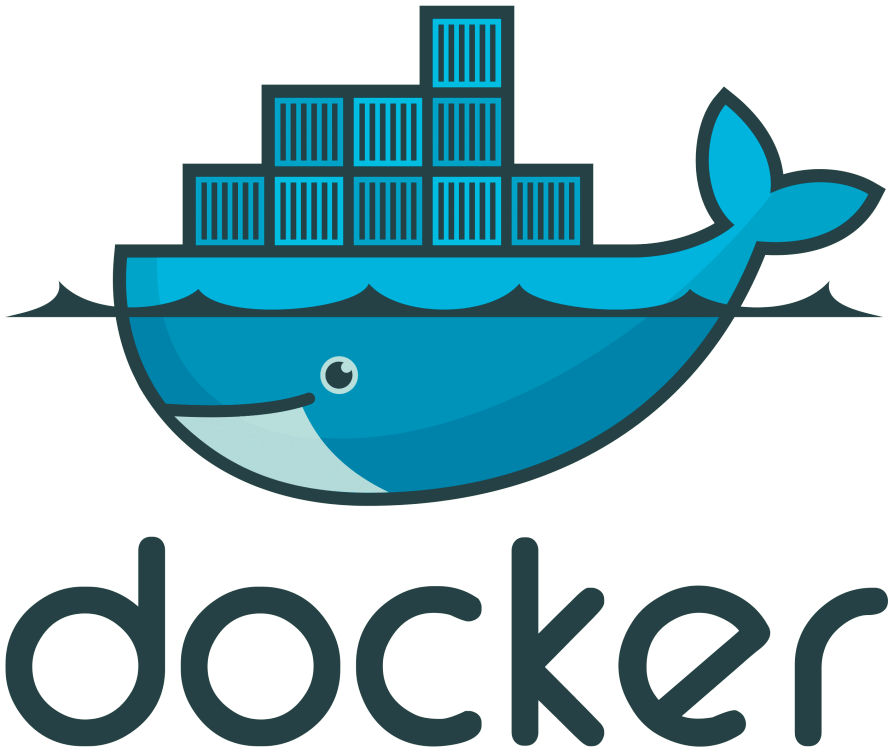
15.1 Fundamentos de Docker.



15.1.1 ¿Qué es Docker?

Docker es una tecnología que se ha convertido en el estandar para desarrollar, distribuir y ejecutar aplicaciones a nivel productivo en todo el mundo, en la nube, un nuestras máquinas, es decir en todos lados.

15.1.2 ¿Por qué utilizar Docker?



Imaginemos un proyecto en el que muchas personas (decenas de miles) puedan colaborar en una especie de pizarra digital para compartir nuestras ideas y colaborar, para poder escalar a cientos de miles o millones de usuarios existen muchos desafíos, esto se refiere a lo que sucede con el software cuando lo escribimos en nuestras máquinas y lo que debe hacer cuando este en los dispositivos, los servidores, la nube, etc.

Estos son los problemas a los que nos enfrenteamos los desarrolladores en todo el mundo, sin importar el lenguaje de programación, framework, base de datos o servidor que utilicemos.

Docker nos permite resolver este tipo de problemas triviales de forma sencilla, rápida y eficiente. Ayuda a trabajar de forma rápida, mejor con mayor confianza y seguridad.

15.2 Problemas que están presentes en el Desarrollo de Software.

1. **Construir:** Necesitamos saber como se construye el software que estamos desarrollando.
2. **Distribuir:** Como se va a distribuir a los usuarios que van a utilizar el software
3. **Ejecutar:** Como va a funcionar el sofware que hemos desarrollado.

1. Construir.- Escribir código en la máquina del desarrollador. (Compile, que no compile, arreglar el bug, compartir código, etc.)

Problemática:

- Entorno de desarrollo (paquetes)
- Dependencias (Frameworks, bibliotecas)
- Versiones de entornos de ejecución (runtime, versión Node)
- Equivalencia de entornos de desarrollo (compartir el código)
- Equivalencia con entornos productivos (pasar a producción)
- Servicios externos (integración con otros servicios ej: Base de Datos)

2. Distribuir.- Llevar la aplicación donde se va a desplegar (Transformarse en un artefacto).

Problemática:

- Output de build heterogeneo (múltiples compilaciones)
- Acceso a servidores productivos (No tenemos acceso al servidor)
- Ejecución nativa vs virtualizada

- Entornos Serverless

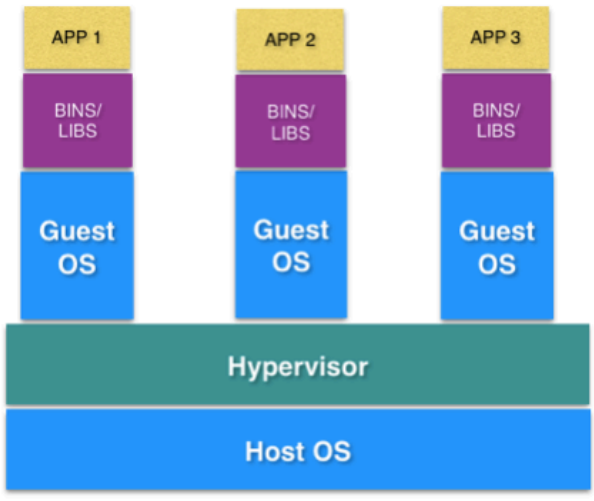
3. Ejecutar.- Implementar la solución en el ambiente de producción (Subir a producción).

Hacer que funcione como debería funcionar.

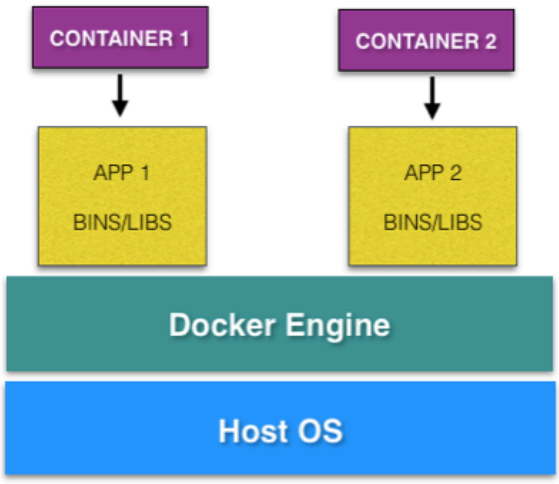
Problemática:

- Dependencia de aplicación (paquetes, runtime)
- Compatibilidad con el entorno productivo (sistema operativo poco amigable con la solución)
- Disponibilidad de servicios externos (Acceso a los servicios externos)
- Recursos de hardware (Capacidad de ejecución - Menos memoria, procesador más debil).

15.3 Maquinas Virtuales vs Docker.



VIRTUAL MACHINE ARCHITECTURE



DOCKER ARCHITECTURE

15.3.1 Virtualización

Version virtual de algún recurso tecnológico, como un servidor, un dispositivo de almacenamiento, un sistema operativo o recurso de red.

La virtualización resuelve los tres problemas de la sección anterior.

15.3.2 Máquinas Virtuales.

15.3.3 Problemas de las VMs

Peso:

- En el orden de los Gb. Repiten archivos en común.
- Inicio Lento

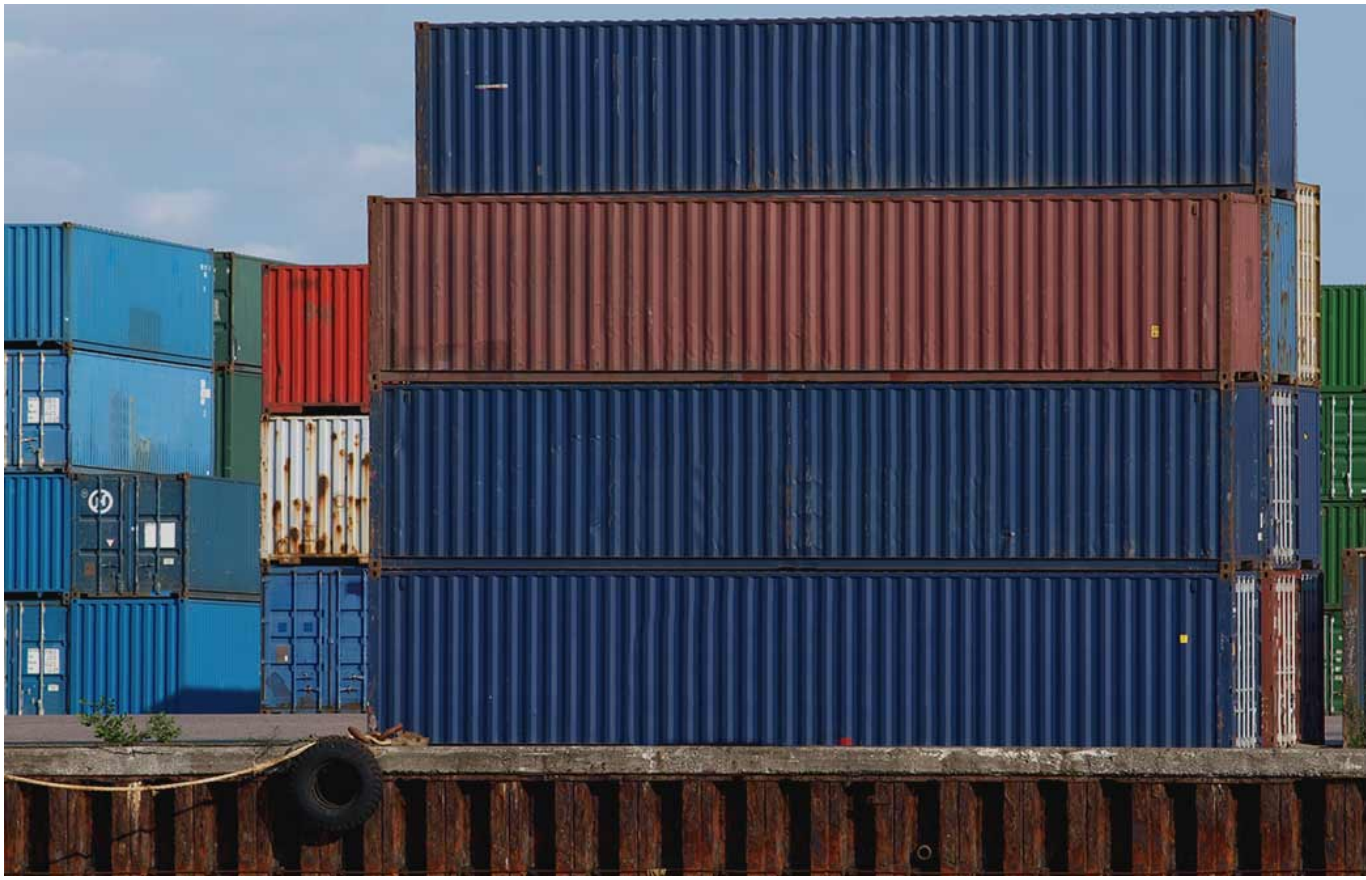
Costo de administración:

- Necesita mantenimiento igual que cualquier otra comptuadora.

Múltiples formatos.

- VDI, VMDK, VHD, raw, entre otros.

15.3.4 Contenedores.



¿Qué es un contenedor?

- Es una agrupación de procesos.
- Es una entidad lógica, no tiene el limite estricto de las máquinas virtuales, emulación del sistema operativo simulado por otra más abajo.
- Ejecuta sus procesos de forma nativa.
- Los procesos que se ejecutan adentro de los contenedores ven su universo como el contenedor lo define, no pueden ver mas allá del contenedor, a pesar de estar corriendo en una maquina más grande.
- No tienen forma de consumir más recursos que los que se les permite. Si esta restringido en memoria ram por ejemplo, es la única que pueden usar.
- A fines prácticos los podemos imaginar cómo maquinas virtuales, pero NO lo son. Máquinas virtuales livianas.
- Docker corre de forma nativa solo en Linux.
- Sector del disco: Cuando un contenedor es ejecutado, el daemon de docker le dice, a partir de acá para arriba este disco es tuyo, pero no puedes subir mas arriba.
- Docker hace que los procesos adentro de un contenedor este aislados del resto del sistema, no le permite ver más allá.

- Cada contenedor tiene un ID único, también tiene un nombre.

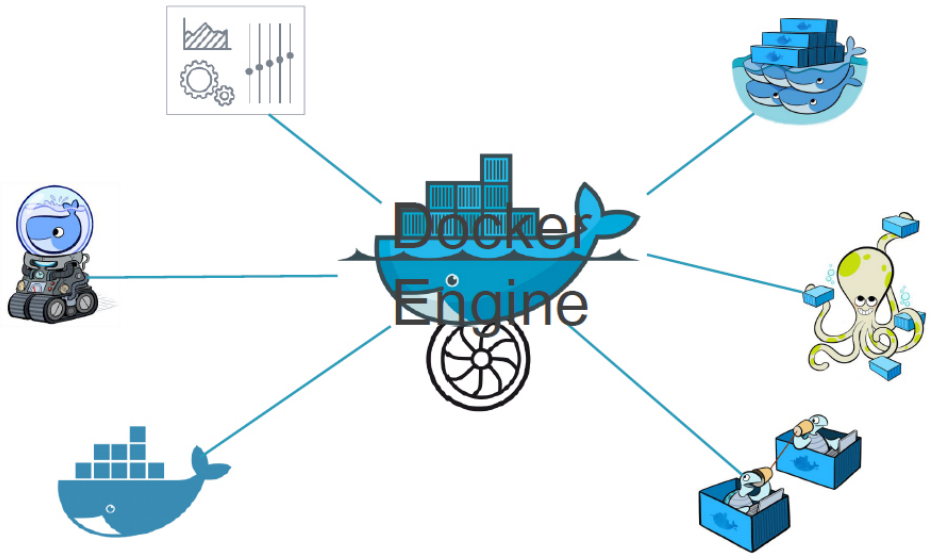
15.3.5 Containerización.

Empleo de **contenedores** para construir y desplegar software.

Caracteristicas.

- Flexibles
- Livianos
- Portables
- Bajo acoplamiento
- Escalables
- Seguros

15.4 Arquitectura de Docker.



Docker es una Plataforma que permite construir, ejecutar y compartir aplicaciones mediante **contenedores**.

Container: Es una instancia de una imagen.

Image: Es un paquete de software que incluye todo lo necesario para ejecutar una aplicación.

Network: Es una red virtual que permite la comunicación entre contenedores y con el exterior.

Data Volumes: Es un directorio que se monta en un contenedor para que este pueda acceder a el.

15.5 Trabajar con Docker.

Paso 1: Instalar Docker

Lo primero que debemos hacer es instalar Docker en nuestro sistema operativo. Para ello, visita el sitio web oficial de Docker y sigue las instrucciones de instalación para tu sistema operativo específico.

Paso 2: Verificar la Instalación

Una vez instalado Docker, verifica que esté funcionando correctamente ejecutando el siguiente comando:

```
docker --version
```

Este comando mostrará la versión de Docker instalada en tu sistema.

Ahora empecemos a probar Docker.

```
docker run hello-world
```

Este comando descargará la imagen de Docker **hello-world** y la ejecutará en un contenedor. Cuando el contenedor se ejecute, imprimirá un mensaje y saldrá.

Paso 3: Descargar una Imagen de Docker

Docker utiliza imágenes para crear contenedores. Una imagen es una plantilla de solo lectura que contiene el sistema operativo, el software y los archivos necesarios para ejecutar una aplicación. Puedes descargar imágenes de Docker Hub, que es un registro de imágenes públicas disponibles para su uso.

Para descargar una imagen, utiliza el siguiente comando:

```
docker pull nombre_de_la_imagen
```

Por ejemplo, para descargar la imagen oficial de Ubuntu, puedes ejecutar:

```
docker pull ubuntu
```

Paso 4: Ver Imágenes Descargadas

Para ver la lista de imágenes descargadas en tu sistema, puedes utilizar el siguiente comando:

```
docker images
```

Paso 5: Crear un Contenedor

Una vez que tengas una imagen descargada, puedes crear un contenedor basado en esa imagen. Para ello, utiliza el siguiente comando:

```
docker run nombre_de_la_imagen
```

Por ejemplo, para crear un contenedor basado en la imagen de Ubuntu, puedes ejecutar:

```
docker run ubuntu
```

Paso 6: Ver Contenedores en Ejecución

Para ver la lista de contenedores en ejecución en tu sistema, puedes utilizar el siguiente comando:

```
docker ps
```

Paso 7: Ver Todos los Contenedores

Para ver la lista de todos los contenedores, incluidos los que no están en ejecución, puedes utilizar el siguiente comando:

```
docker ps -a
```

Ahora probemos el docker id con el comando inxpect.

```
docker inspect ID_del_contenedor
```

Con este comando vemos toda la información del contenedor.

Ahora probemos dar un NAME al contenedor.

```
docker run --name nombre_del_contenedor nombre_de_la_imagen
```

Muy bien, por favor hagamos una pequeña práctica para probar lo que hemos aprendido, por favor creemos 10 contenedores con nombres de paises, por ejemplo: Colombia, Argentina, Brasil, Chile, Ecuador, España, Francia, Italia, México, Perú.

```
docker run --name Colombia hello-world
docker run --name Argentina hello-world
docker run --name Brasil hello-world
docker run --name Chile hello-world
'''etc'''
```

Ahora probemos el comando inxpect con el nombre del contenedor.

```
docker inspect nombre_del_contenedor
```

Paso 8: Detener un Contenedor

Para detener un contenedor en ejecución, puedes utilizar el siguiente comando:

```
docker stop ID_del_contenedor
```

Reemplaza `ID_del_contenedor` con el ID real del contenedor que deseas detener.

Paso 9: Eliminar un Contenedor

Antes de continuar probemos el comando docker container prune, este comando elimina todos los contenedores que no estén en ejecución.

```
docker container prune
```

Para eliminar un contenedor que no está en ejecución, puedes utilizar el siguiente comando:

```
docker rm ID_del_contenedor
```

Reemplaza `ID_del_contenedor` con el ID real del contenedor que deseas eliminar.

15.6 Docker desde Visual Studio Code.

los pasos para instalar la extensión de Docker en Visual Studio Code y cómo utilizarla para trabajar con Docker en tu proyecto.

Paso 1: Instalar la extensión de Docker para Visual Studio Code

Abre Visual Studio Code.

Haz clic en el icono de extensiones en el panel lateral izquierdo o presiona Ctrl+Shift+X (o Cmd+Shift+X en macOS) para abrir la sección de extensiones.

En el cuadro de búsqueda, escribe “Docker” y selecciona la extensión “Docker” desarrollada por Microsoft.

Haz clic en el botón “Instalar” para instalar la extensión.

Una vez instalada, verás un nuevo icono de Docker en la barra lateral izquierda de VS Code.

Paso 2: Utilizar la extensión de Docker en tu proyecto

Una vez que la extensión de Docker está instalada, puedes utilizarla para gestionar imágenes y contenedores de Docker en tu proyecto.

Abre la carpeta de tu proyecto en Visual Studio Code.

Haz clic en el icono de Docker en la barra lateral izquierda para abrir la vista de Docker.

En la vista de Docker, verás diferentes secciones como “Imagenes”, “Contenedores”, “Volúmenes” y “Redes”.

Para construir una imagen de Docker para tu proyecto, haz clic con el botón derecho en la carpeta raíz de tu proyecto y selecciona “Build Image…” en el menú contextual. Esto abrirá una ventana donde puedes configurar los detalles de la imagen y su etiqueta.

Para crear y ejecutar un contenedor basado en la imagen que acabas de construir, haz clic con el botón derecho en la imagen en la sección "Imagenes" y selecciona "Run". Esto abrirá una ventana donde puedes configurar las opciones del contenedor, como los puertos y las variables de entorno.

Una vez que el contenedor esté en ejecución, puedes ver sus registros y otros detalles haciendo clic en el contenedor en la sección "Contenedores".

Puedes detener y eliminar contenedores desde la vista de Docker haciendo clic con el botón derecho en el contenedor y seleccionando "Stop" o "Remove".

Estos son algunos de los pasos básicos para utilizar Docker desde Visual Studio Code con la extensión oficial de Docker.

La extensión proporciona una forma sencilla de gestionar imágenes y contenedores de Docker directamente desde el entorno de desarrollo de VS Code, lo que facilita el trabajo con Docker en tus proyectos.

Práctica: Dockerizar el Proyecto Django "Blog"

A continuación, vamos a dockerizar el proyecto Django **Blog** que hemos estado desarrollando. Para ello, vamos a seguir los siguientes pasos:

1. Crear un archivo `Dockerfile` en el directorio raíz del proyecto.

Este archivo contendrá las instrucciones para construir la imagen de Docker.

2. En el `Dockerfile`, especificar la imagen base que utilizaremos.

Por ejemplo, podemos usar la imagen oficial de Python para Django:

```
FROM python:3.8
```

3. Copiar el código fuente de nuestra aplicación al contenedor:

```
COPY . /app
```

4. Establecer el directorio de trabajo dentro del contenedor:

```
WORKDIR /app
```

5. Instalar las dependencias de Python necesarias para nuestro proyecto:

```
RUN pip install -r requirements.txt
```

6. Exponer el puerto en el que se ejecutará nuestro servidor de Django:

```
EXPOSE 8000
```

7. Especificar el comando para ejecutar nuestra aplicación:

```
CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]
```

8. Guardar y cerrar el archivo `Dockerfile`.
9. Desde la línea de comandos, navegar al directorio raíz del proyecto y ejecutar el siguiente comando para construir la imagen de Docker:

```
docker build -t nombre_de_la_imagen .
```

Reemplaza `nombre_de_la_imagen` con un nombre significativo para tu imagen.

10. Una vez que la imagen se haya construido correctamente, puedes crear un contenedor basado en esa imagen:

```
docker run -p 8000:8000 nombre_de_la_imagen
```

Reemplaza `nombre_de_la_imagen` con el nombre de la imagen que creaste en el paso anterior.

11. Ahora, deberías poder acceder a tu aplicación Django desde tu navegador ingresando `http://localhost:8000`.

Con estos pasos, habrás dockerizado exitosamente tu proyecto Django **Blog** y estará listo para ser desplegado en cualquier entorno que tenga Docker instalado.

15.7 Introducción a Docker Compose

Docker Compose es una herramienta que permite definir y gestionar aplicaciones multi-contenedor. Permite definir las dependencias, servicios, redes y volúmenes que componen una aplicación en un archivo YAML, lo que facilita la configuración y el despliegue de la aplicación en diferentes entornos.

A diferencia de Docker, que se enfoca en la construcción y ejecución de contenedores individuales, Docker Compose se centra en la orquestación de múltiples contenedores y su interacción.

15.8 Ventajas de usar Docker Compose

Definición clara de servicios: Con Docker Compose, podemos definir todos los servicios necesarios para nuestra aplicación en un solo archivo YAML, lo que facilita la gestión de la infraestructura de la aplicación.

Configuración simplificada: Docker Compose permite establecer todas las configuraciones necesarias para cada servicio en un solo lugar, lo que facilita la gestión de la configuración y el despliegue en diferentes entornos.

Interconexión de servicios: Docker Compose facilita la conexión de diferentes servicios y la comunicación entre ellos, lo que es especialmente útil para aplicaciones que requieren una arquitectura de microservicios.

Escalabilidad: Docker Compose permite escalar servicios individuales según las necesidades de la aplicación, lo que facilita el escalado horizontal de la aplicación.

15.8.1 Cómo usar Docker Compose en nuestro proyecto Django "Blog"

Para utilizar Docker Compose en nuestro proyecto Django **Blog**, seguiremos los siguientes pasos:

1. Crear un archivo docker-compose.yml en el directorio raíz del proyecto.
2. Este archivo contendrá la configuración de los servicios y contenedores de nuestra aplicación.

En el archivo docker-compose.yml, definir los servicios necesarios para nuestra aplicación. Por ejemplo, podemos definir dos servicios: uno para el servidor de Django y otro para la base de datos.

```
version: '3'

services:
  db:
    image: postgres
    environment:
      POSTGRES_DB: blog
      POSTGRES_USER: myuser
      POSTGRES_PASSWORD: mypassword

  web:
    build: .
    command: python manage.py runserver 0.0.0.0:8000
    ports:
      - "8000:8000"
    depends_on:
      - db
```

Guardar y cerrar el archivo docker-compose.yml.

Desde la línea de comandos, navegar al directorio raíz del proyecto y ejecutar el siguiente comando para construir las imágenes de los servicios definidos en el archivo docker-compose.yml:

```
docker-compose build
```

Una vez que las imágenes se hayan construido correctamente, puedes ejecutar el siguiente comando para iniciar los contenedores definidos en el archivo docker-compose.yml:

```
docker-compose up
```

Ahora, deberías poder acceder a tu aplicación Django desde tu navegador ingresando http://localhost:8000.

Con estos pasos, habrás utilizado Docker Compose para gestionar los servicios y contenedores de tu aplicación Django "Blog". Docker Compose simplifica el despliegue y la gestión de aplicaciones multi-contenedor, lo que lo convierte en una herramienta muy útil para el desarrollo de aplicaciones complejas y escalables.