

Taller básico de Django

Introducción a la Programación - 1^{er} semestre del 2024 

 ggodoy@campus.ungs.edu.ar

 @gonzag2023

django



Agenda

Preliminares

1. ¿Qué es Django?
2. Conocimientos necesarios
3. Programas requeridos
4. Estructura de un proyecto
5. Patrón MVT
6. ORMs
7. ¿Qué limitaciones posee el *framework*?

Agenda

Conociendo la estructura

1. Vista global
2. *View (views.py)*
3. *URLs (urls.py)*
4. *Model (models.py)*
5. *HTMLs (templates) + T.I.*

Agenda

Primeros pasos: parte I

1. *Virtual Environments*
2. *Instalación Django*
3. *Params*
4. *Render*
5. **Hola Mundo**

Agenda

Primeros pasos: parte II

1. Estructuras para iterar: CICLO **FOR**
2. Estructuras condicionales: **IF-ELSE**
3. *Forms*
4. *URL names*
5. *Static Files*

Agenda

Adicionales

1. *Bootstrap Library*
2. ¿Qué son las **APIs**?

Agenda

Finalizando...

1. Actividad integradora
2. Otros proyectos (Open Source)
3. Bibliografía

INICIO PARTE I: TEORÍA

PRELIMINARES

¿Qué es Django?



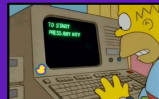
¿Qué es Django?

- Un ***framework*** o entorno de trabajo, **es un conjunto de herramientas, librerías y buenas prácticas.**
- Django es un *framework* web gratuito, *open source*, escrito en Python.
- Con Django, es posible:
 - **Crear aplicaciones web complejas en forma rápida y sencilla**, reutilizando y exportando tareas de una app a otra. Ejs.: blogs, redes sociales (Instagram), plataformas *e-commerce*, CMS (Dropbox), entre otros.
 - Crear aplicaciones de tiempo real como *chats* o mensajería instantánea.
 - Crear plataformas educativas (LMS) o de gestión de cursos, evaluaciones, usuarios y más.

Conocimientos necesarios



Conocimientos necesarios



- **Saber programar** en Python:
 - Variables
 - Condicionales
 - Ciclo FOR, funciones
 - Listas / listas de listas* (ocasionalmente)
 - POO** (**sugerido**, bases/conceptos básicos)
- Un poco de [HTML](#) + [CSS](#).
- Algo de **tiempo** y **ganas**.

Programas requeridos



Programas requeridos



- Python + PIP.
- Django.
- Navegador web (Chrome, Opera, Safari, Firefox).
- DB Browser for SQLite.
- S.O.: GNU/Linux, Microsoft Windows o MacOS.
- PC Specs:
 - CPU: 1Ghz. o más.
 - RAM: 4GB como mínimo.
 - Espacio en disco: 5GB mínimo.
- Conexión a internet.

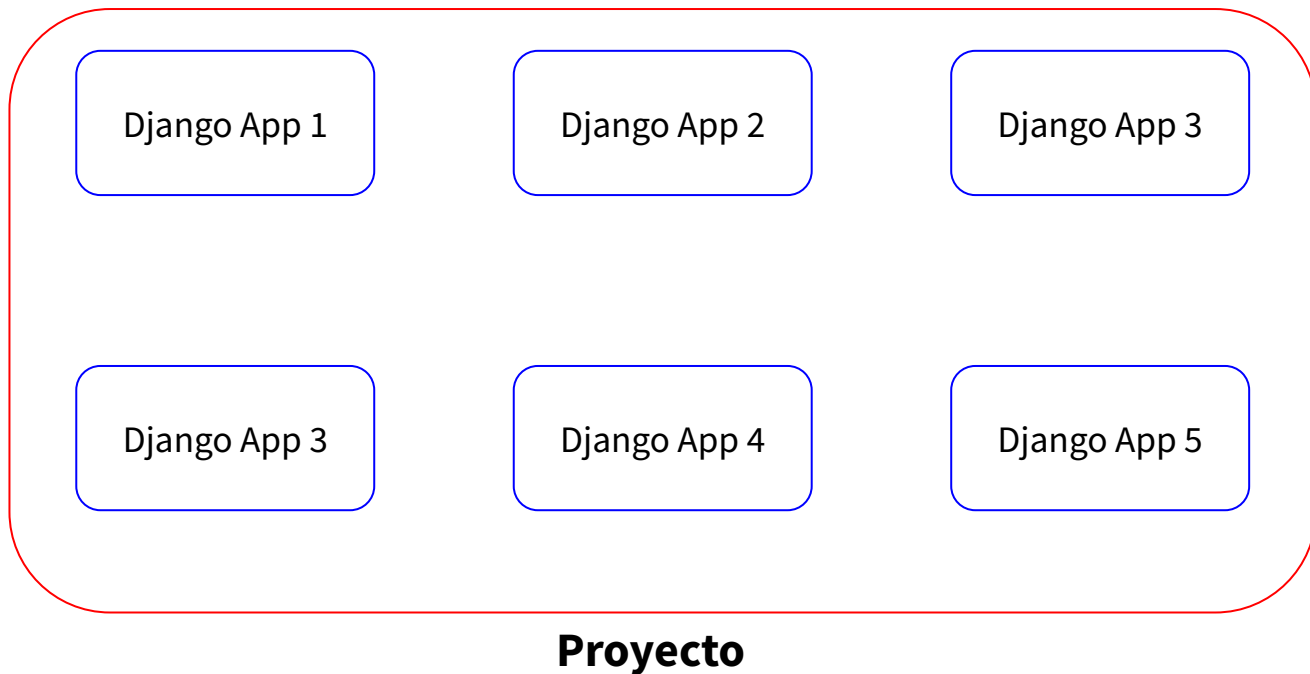
Estructura de un proyecto



Estructura de un proyecto



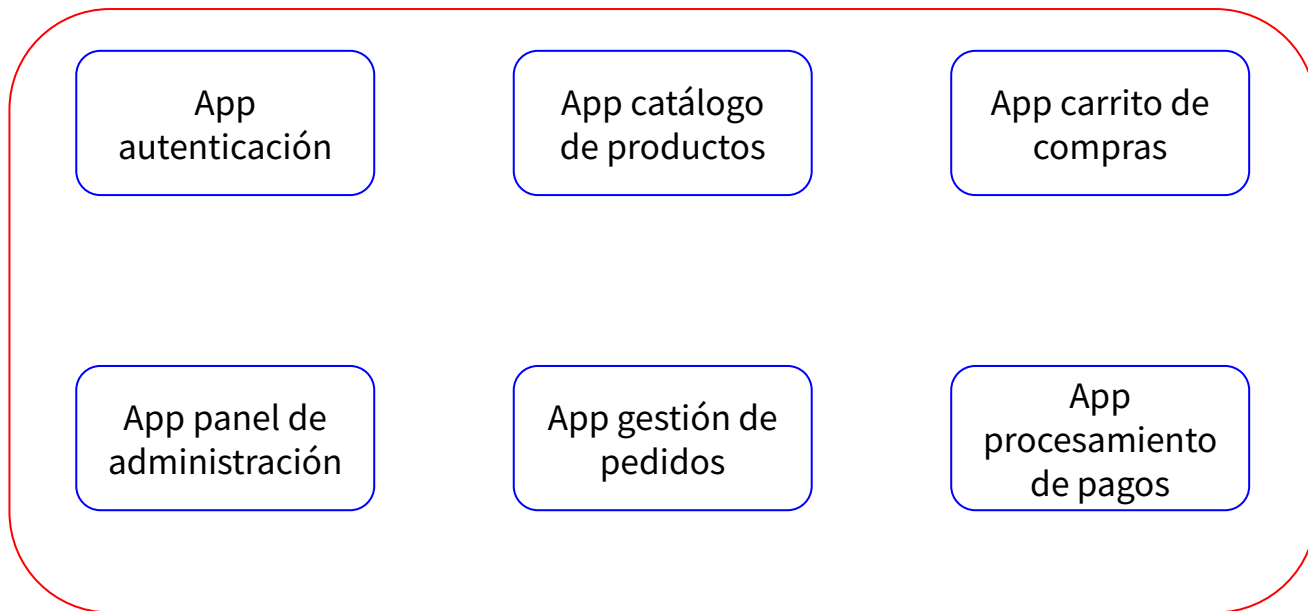
Se genera **1 proyecto**, que puede contener **1 o más aplicaciones**.



Estructura de un proyecto



Ejemplo: **tienda e-commerce**

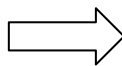


Proyecto e-commerce

Estructura de un proyecto



Gestor de
dependencias.



install **django**

Instalar Django.

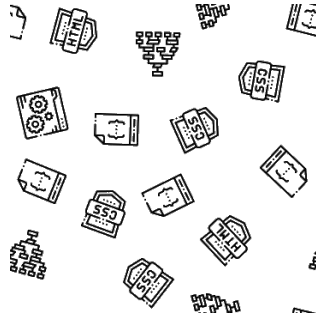
django-admin startproject <nombre_proyecto>

Crear un proyecto.

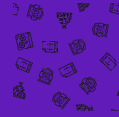
python **manage.py startapp** <nombre_app>

Crear una app.

Patrón MVT



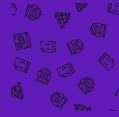
Patrón MVT



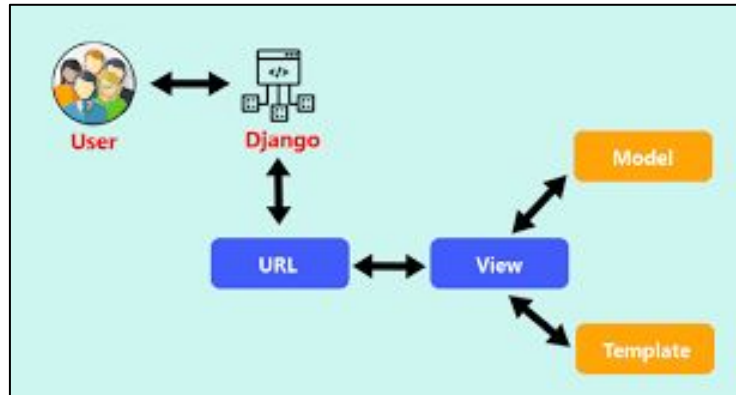
- Está basado en el patrón MVC (**M**odel - **V**iew - **C**ontroller):
 - El **modelo** es quién gestiona los datos en la base de datos.
 - La **vista** es quién muestra la info. al usuario, lo que vé, con lo que se maneja.
 - El **controlador** es un “puente” entre la vista y el modelo (lógica de negocio).



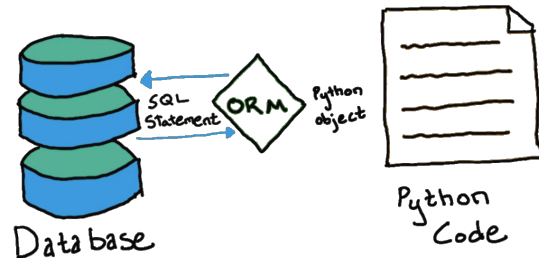
Patrón MVT



- Django utiliza este mismo patrón, pero lo redefine como MVT (**m**odel - **v**iew - **t**emplate):
 - El **modelo** es quién se comunica con la base de datos.
 - Los **templates** son las “páginas” o info. que se muestra a los usuarios.
 - Las **vistas** son los controladores o “puentes” entre el modelo y los templates.



ORMs



ORMs



- Un **ORM** (*object relational mapping*) es una técnica que **permite interactuar con una base de datos relacional** utilizando objetos Python **en lugar de escribir consultas SQL directamente.**
- En Django, el ORM se encarga de *mapear* las clases de modelos de Django (que definen la estructura de la base de datos) a las tablas en la base de datos y viceversa. **Esto significa que es posible manipular los datos en la base de datos utilizando objetos Python SIN tener que escribir queries SQL.**

ORMs



table with cats in database

id	name	weight_g
1	Tigger	3200
2	Luna	2900
3	Jasper	3300

Model
Cat

operations with all cats

QuerySet (.objects)

- `.create(...)`
- `.get(...)`
- `.filter(...)`
- `.all()`

operations with one cat

instance

- `.id / .name / .weight_g`
- `.save()`
- `.delete()`

ORMs

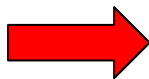


- Ejemplo: supongamos que queremos crear un modelo para representar **libros** en una **biblioteca**. Cada libro tiene por atributos un título, un autor y una fecha de publicación. Una posible alternativa es la siguiente:

```
# models.py
from django.db import models

class Libro(models.Model):
    titulo = models.CharField(max_length=100)
    autor = models.CharField(max_length=100)
    año_publicacion = models.IntegerField()

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



Esta función es muy similar al **toString()** de Java.

ORMs



- Ahora, supongamos que queremos recuperar todos los libros de la base de datos que fueron publicados **después del año 2000**:

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

def libros_recientes(request):
    libros = Libro.objects.filter(año_publicacion__gt=2000)
    return render(request, 'libros.html', {'libros': libros})
```

ORMs



- Finalmente, **pasamos los resultados de la consulta al *template* libros.html, donde mostraremos la lista de libros:**

```
<!-- libros.html -->
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>Libros Recientes</title>
</head>
<body>
  <h1>Libros Recientes</h1>
  <ul>
    {% for libro in libros %}
      <li>{{ libro.titulo }} - {{ libro.autor }} ({{ libro.año_publicacion }})</li>
    {% endfor %}
  </ul>
</body>
</html>
```

Recorre cada libro del listado de libros que recibió el template.

Atributos de los libros (YA definidos por el modelo)

Fin recorrido.

¿Y si, además, quiero mostrar aquellos libros **cuyo título comienza con “a” (mayús/minús)?**



ORMs



- 1^{ra} forma: usamos la función `__istartswith` (*case insensitive*), aplicándola al título y verificando que comience con “a”/”A”:

```
from django.shortcuts import render
from .models import Libro
```

```
def libros_recientes(request):
```

```
    libros = Libro.objects.filter(año_publicacion__gt=2000, titulo__istartswith='a')
```

```
    return render(request, 'libros.html', {'libros': libros})
```



ORMs



- 2^{da} forma: utilizando un bloque condicional **IF-ELSE** en el template, haciendo que se muestren aquellos libros cuyo título comience con “a” o “A”:

```
<!-- libros.html -->
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>Libros Recientes con Título que Empieza por "A"</title>
</head>
<body>
  <h1>Libros Recientes con Título que Empieza por "A"</h1>
  <ul>
    {% for libro in libros %}
      {% if libro.titulo|lower|startswith:"a" %}
        <li>{{ libro.titulo }} - {{ libro.autor }} ({{ libro.año_publicacion }})</li>
      {% endif %}
    {% endfor %}
  </ul>
</body>
</html>
```

Versión A

Filtros en cadenas.

➔

ORMs



- 2^{da} forma: utilizando un bloque condicional **IF-ELSE** en el template, haciendo que se muestren aquellos libros cuyo título comience con “a” o “A”:

```
<!-- libros.html -->
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>Libros Recientes con Título que Empieza por "A"</title>
</head>
<body>
  <h1>Libros Recientes con Título que Empieza por "A"</h1>
  <ul>
    {% for libro in libros %}
      {% if libro.titulo.lower().startswith("a") %}
        <li>{{ libro.titulo }} - {{ libro.autor }} ({{ libro.año_publicacion }})</li>
      {% endif %}
    {% endfor %}
  </ul>
</body>
</html>
```

Versión B

Funciones en cadenas.

¿Qué limitaciones posee el *framework*?



¿Qué limitaciones posee el *framework*?



- Es un **monolito**.

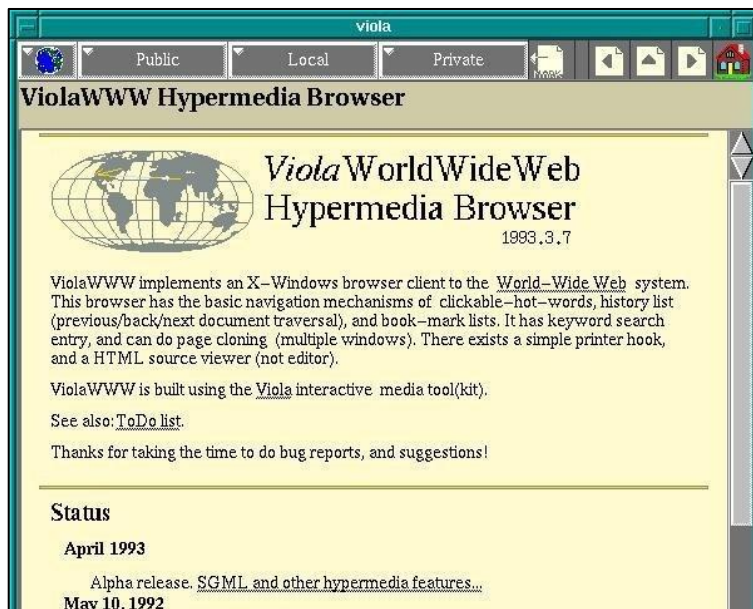
Una **aplicación monolítica** es una unidad donde todas las partes están en una misma base de código. Es fácil de desarrollar y de testear, y puede haber varias copias de la misma aplicación en distintos servidores. Es una arquitectura ideal para proyectos simples.



¿Qué limitaciones posee el *framework*?



- Los *templates* no son “particularmente” interactivos. **A cada petición del browser, una respuesta HTML del server.**



¿Qué limitaciones posee el *framework*?



- Django es un *framework* completo de desarrollo. Dependiendo del proyecto, **la curva de aprendizaje puede ser leve o moderada.**

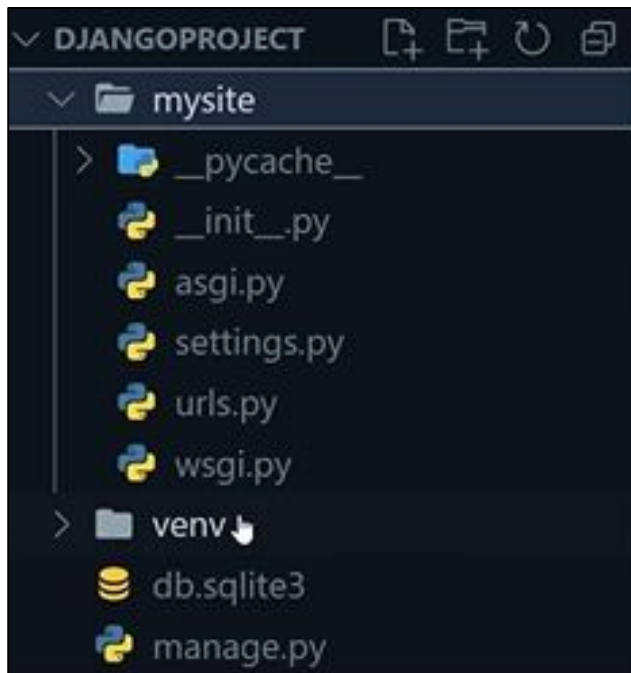


LA MENTE ES COMO UN PARACAÍDAS,
SOLO FUNCIONA SI SE ABRE.

CONOCIENDO LA ESTRUCTURA

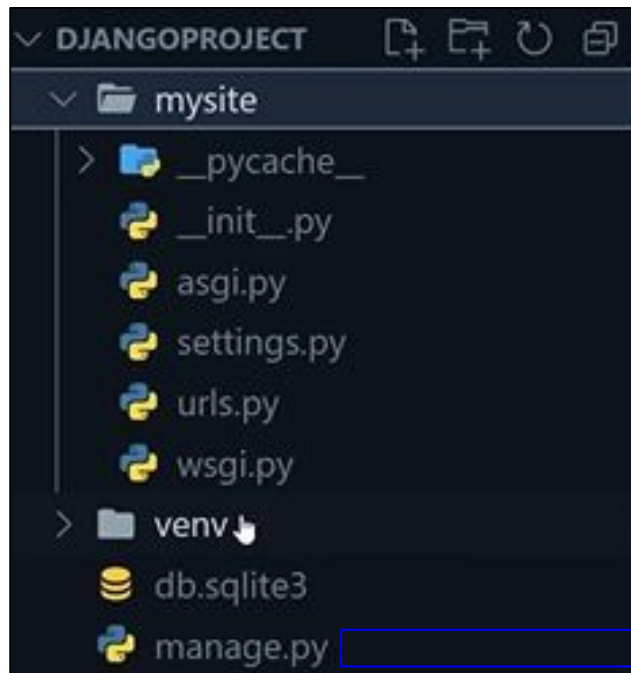


Conociendo la estructura: vista global



Vista del proyecto.

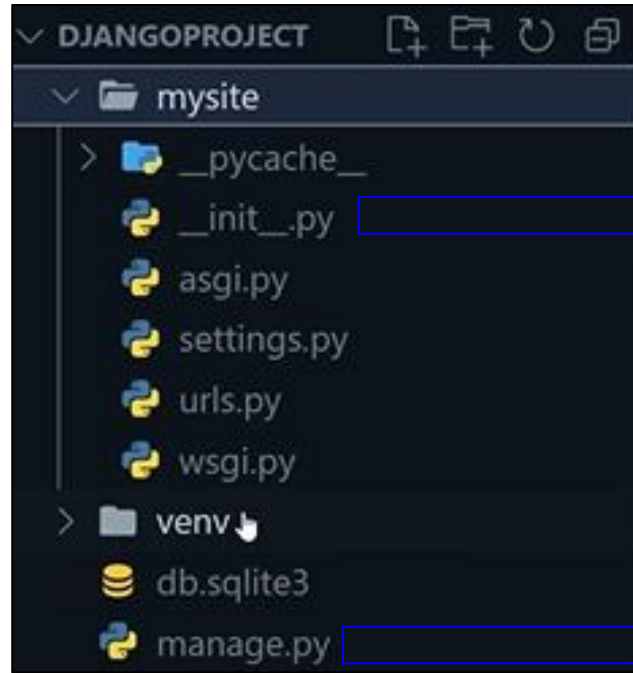
Conociendo la estructura: vista global



Vista del proyecto.

Ejecución de comandos de administración, ayuda, testing, compilación de mensajes.

Conociendo la estructura: vista global

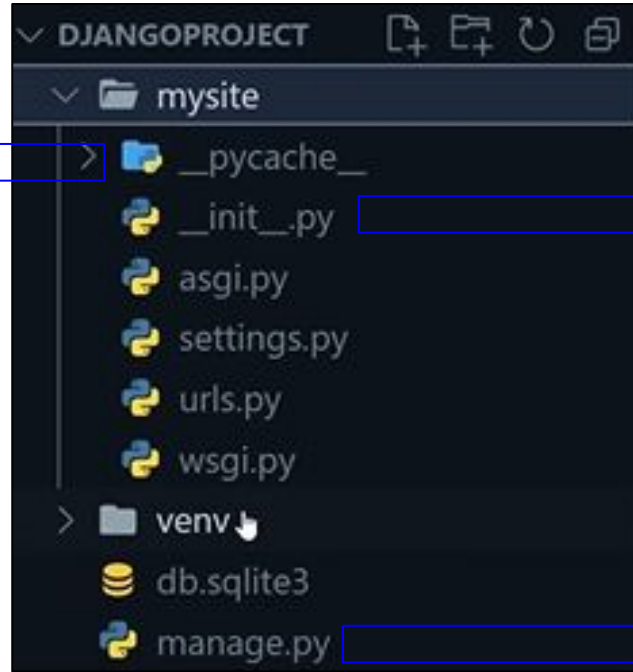


Identificación de módulo de Python (indica que *mysite* es un módulo de Django).

Ejecución de comandos de administración, ayuda, testing, compilación de mensajes.

Vista del proyecto.

Conociendo la estructura: vista global



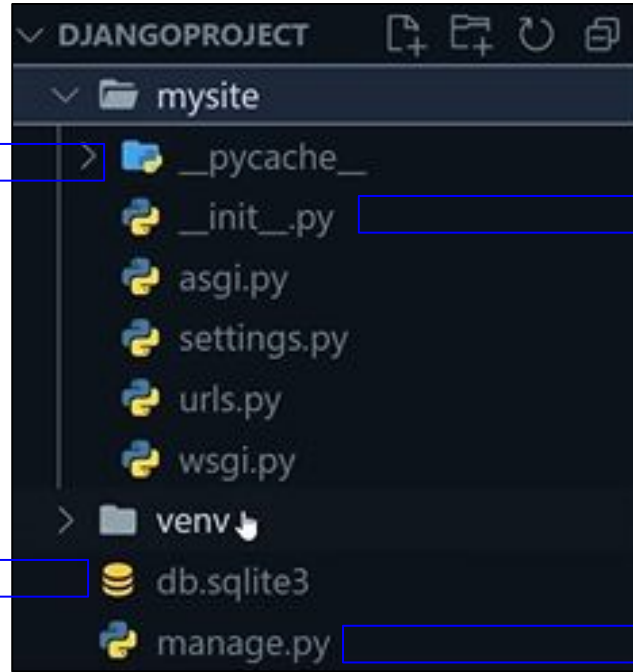
Archivos compilados del proyecto.

Identificación de módulo de Python (indica que *mysite* es un módulo de Django).

Ejecución de comandos de administración, ayuda, testing, compilación de mensajes.

Vista del proyecto.

Conociendo la estructura: vista global



Archivos compilados del proyecto.

Identificación de módulo de Python (indica que *mysite* es un módulo de Django).

Base de datos por *default* (SQLite). **Funciona desde un archivo.**

Ejecución de comandos de administración, ayuda, testing, compilación de mensajes.

Vista del proyecto.

Conociendo la estructura: vista global

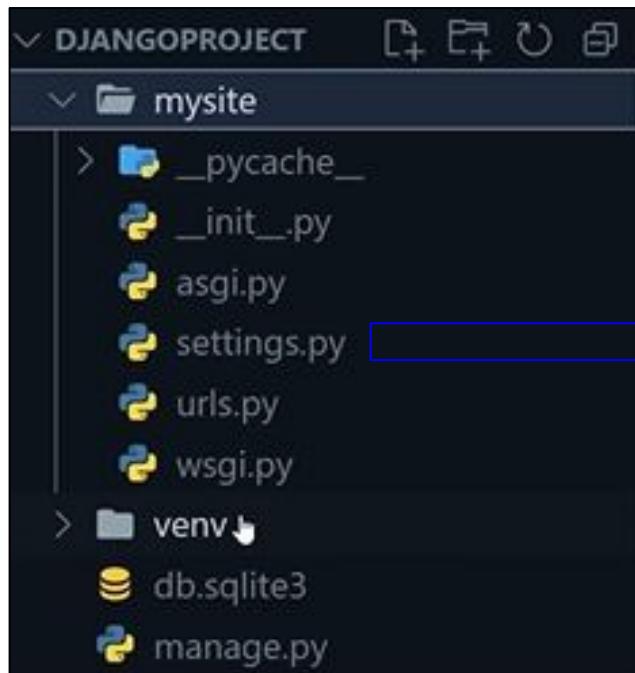


python **manage.py** **-help** (doble gui3n)

Manual de comandos disponibles en manage.py.

```
(venv) C:\Users\fazt\Desktop\django project>
```

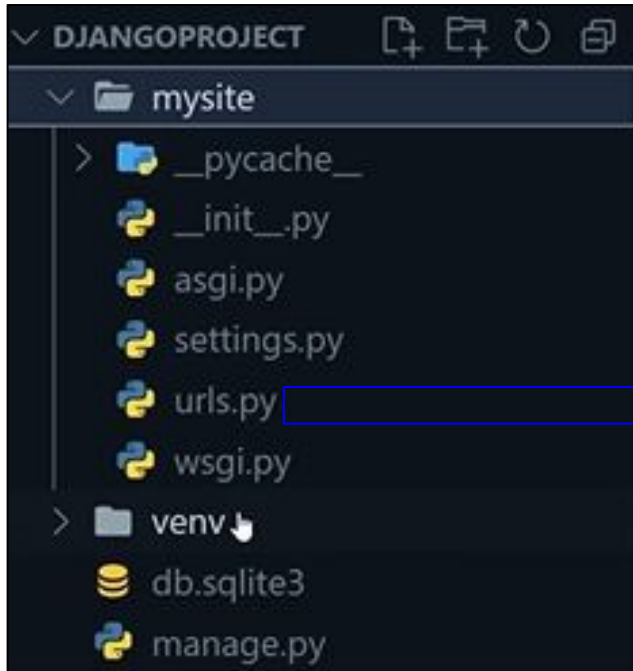
Conociendo la estructura: vista global



Configuración global del proyecto:

- DEBUG (*testing*).
- ALLOWED_HOSTS (quiénes pueden acceder).
- SECRET_KEY (encriptación y cifrado).
- BASE_DIR (directorio principal).
- **INSTALLED_APPS (aplicaciones instaladas).**
- STATIC_FILES (ubicación de archivos html/css/js).
- (...).

Conociendo la estructura: vista global

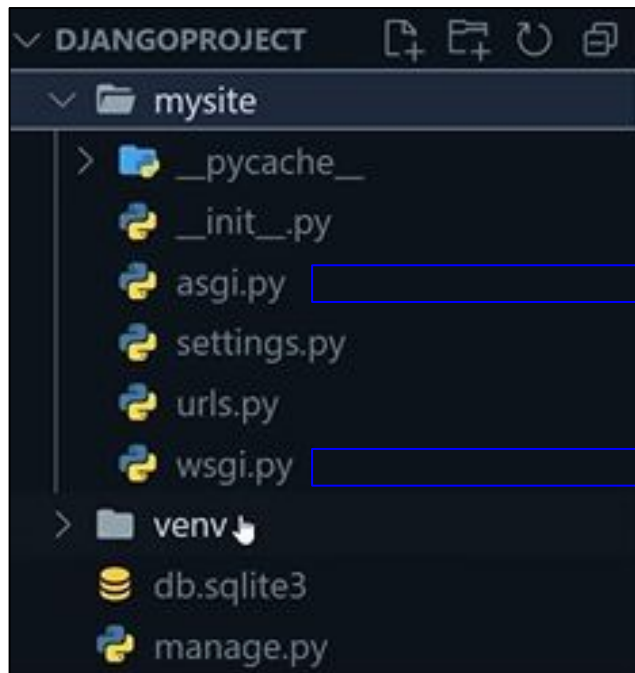


Enrutamiento/rutas de acceso:

- Representan las URLs que los usuarios pueden visitar. **Se vé más adelante (URL names).**

```
urlpatterns = [  
    path('', home_view, name='home'),  
    path('contact/', contact_view),  
    path('about/', about_view),  
    path('social_view/', social_view),  
    path('product/', product_detail_view),  
    path('admin/', admin.site.urls),  
]
```

Conociendo la estructura: vista global



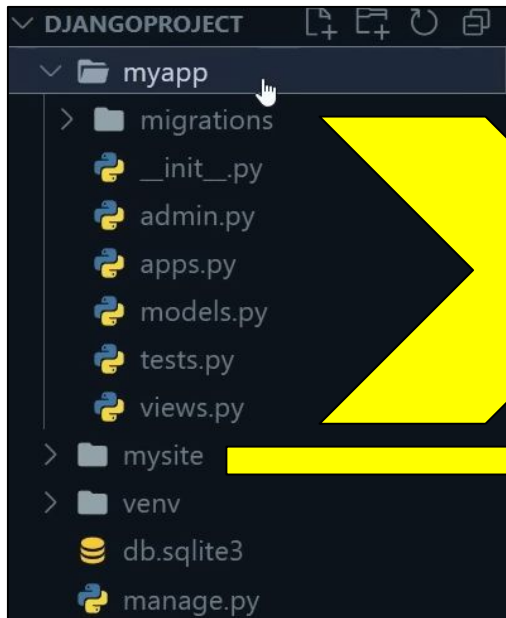
- Módulos encargados del *deploy* de la aplicación. **NO los usamos, por el momento, en Intro. a la Programación.**

Conociendo la estructura: vista global



```
python manage.py startapp myapp
```

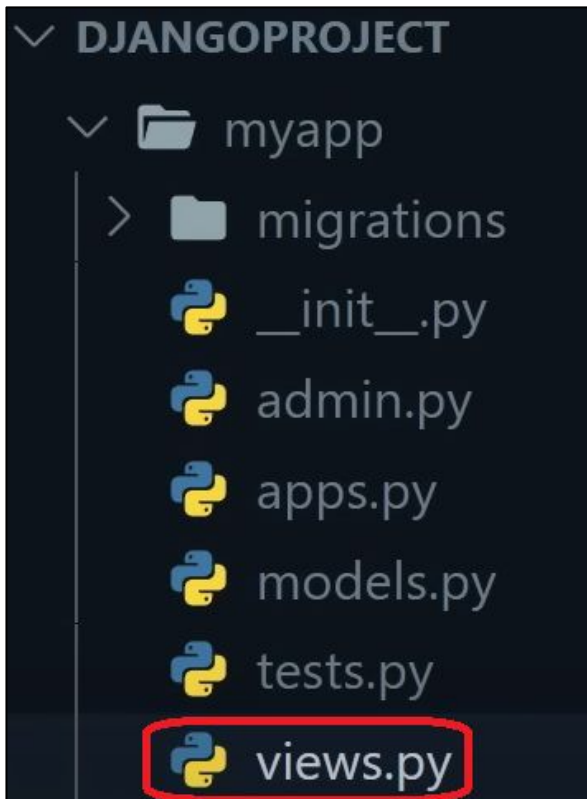
Creación de una aplicación llamada “myapp” en el proyecto “mysite”.



Aplicación “myapp” creada.

**Configuración global
del proyecto **para todas
las apps.****

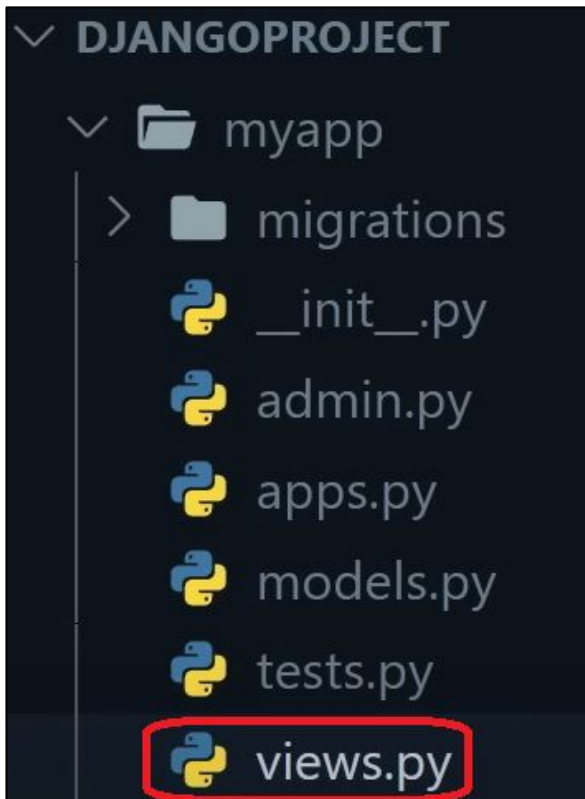
Conociendo la estructura: views.py



views.py

- Representa un *controller* en la lógica del patrón MVC.
- Contiene, en su versión general, toda la **lógica de negocio que permite capturar datos desde la BD (u otras fuentes), procesarlos y emitir esa información.**
- La información procesada con anterioridad, típicamente es devuelta al usuario mediante un **template** (“html”).

Conociendo la estructura: views.py

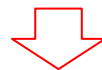


views.py (ejemplo)

```
index.html  views.py  X
justColor > views.py > home
1  from django.http import HttpResponse
2  from django.shortcuts import render
3
4  def home(request):
5  |  return render(request, 'index.html')
```

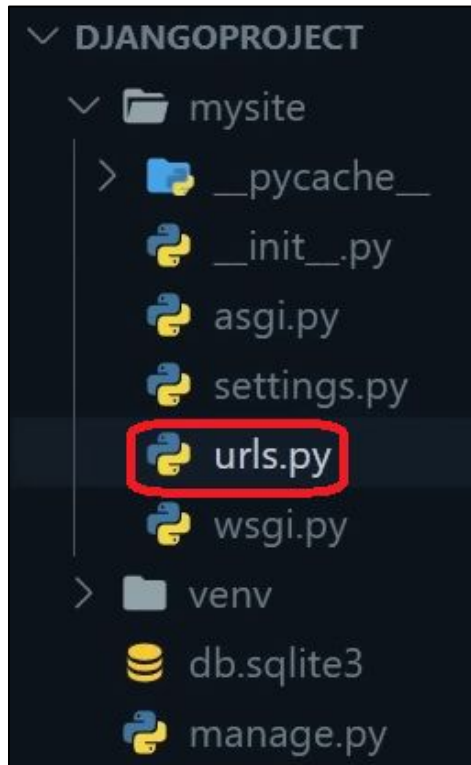


Por def., siempre se envía un
request.



Template a
mostrar.

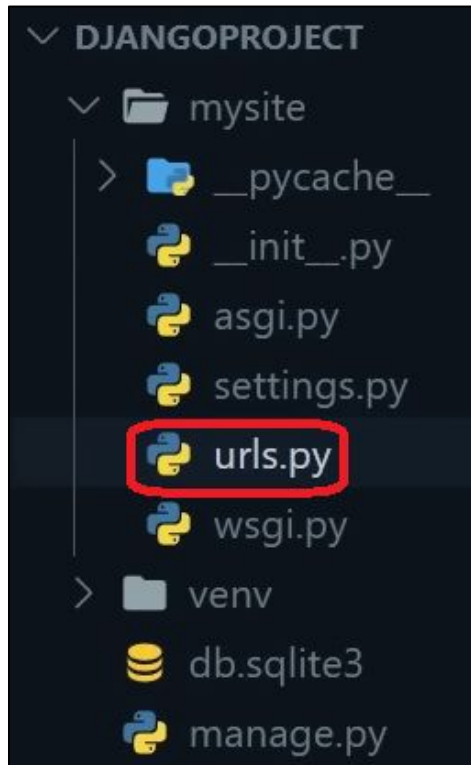
Conociendo la estructura: urls.py



urls.py

- **Sirve para definir qué URLs se podrán visitar.** Si no lo hacemos, no se podrán acceder (clásico error 404 -NOT FOUND-).
- *urls.py* es un módulo de Python que se encarga del **routing**, es decir, de vincular cada página con una URL para que el usuario pueda entrar sin problemas. Ej. miapp.com/clientes.
- Por defecto, Django genera un único *urls.py* para todo el proyecto. **Es recomendable tener 1 *urls.py* por cada aplicación del sistema**, y usar al del proyecto como agrupador de los demás (**`include()`**).

Conociendo la estructura: urls.py



urls.py (ejemplo)

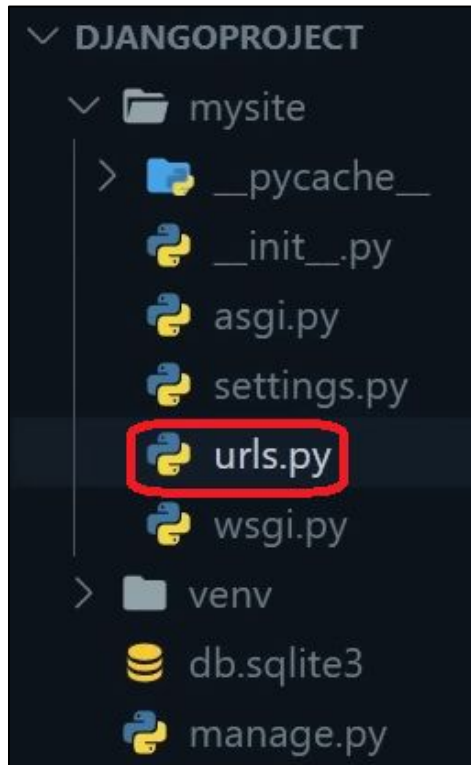
**URL de
acceso.**

**Función dentro de
views.py que hará el
render de un template.**

```
urlpatterns = [
    path('articles/2003/', views.special_case_2003),
    path('articles/<int:year>', views.year_archive),
    path('articles/<int:year>/<int:month>', views.month_archive),
    path('articles/<int:year>/<int:month>/<slug:slug>', views.article_detail),
]
```

**Otros parámetros
que recibe la URL.**

Conociendo la estructura: urls.py



urls.py (ejemplo)

**URL de
acceso.**

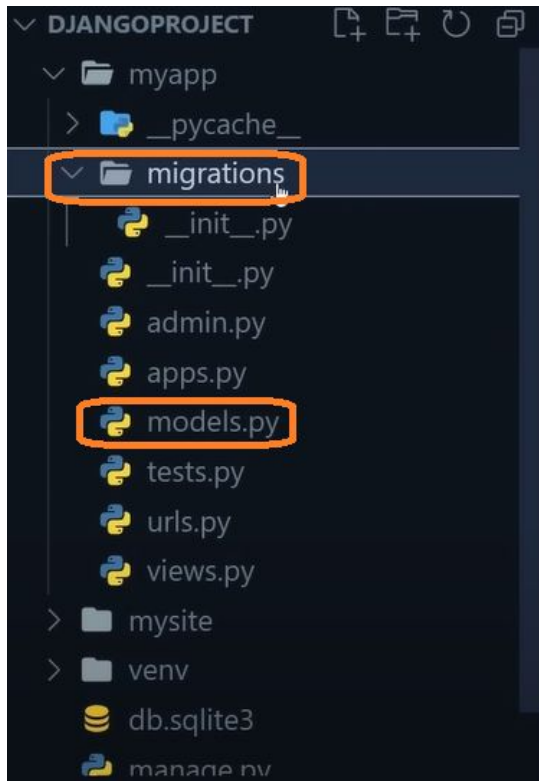
**Función dentro de
views.py que hará el
render de un template.**

```
urlpatterns = [
    path('articles/2003/', views.special_case_2003),
    path('articles/<int:year>', views.year_archive),
    path('articles/<int:year>/<int:month>', views.month_archive),
    path('articles/<int:year>/<int:month>/<slug:slug>', views.article_detail),
]
```

**Otros parámetros
que recibe la URL.**

 **Las URLs pueden tener
un nombre asociado.**

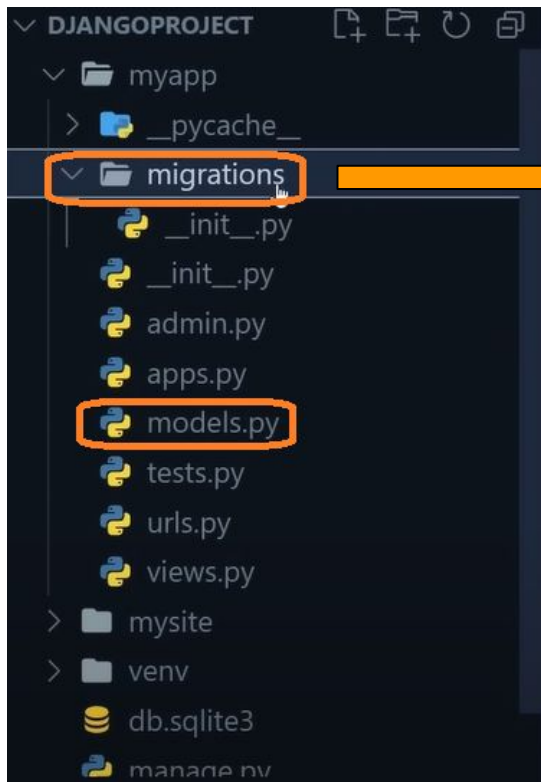
Conociendo la estructura: models.py



models.py

- Sirve para **crear clases**, que luego se convertirán en **tablas en la base de datos**.
- Existen 2 comandos principales:
 - **python manage.py makemigrations**: genera los compilados/"scripts" para la base de datos.
 - **python manage.py migrate**: ejecuta las migraciones/"scripts" del comando anterior, en la base de datos.

Conociendo la estructura: models.py



- El directorio *migrations* se usa para guardar los “scripts” generados con **makemigrations**, que luego se ejecutarán con **migrate**.

```
C:\Users\fazt\Desktop\django\project>python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

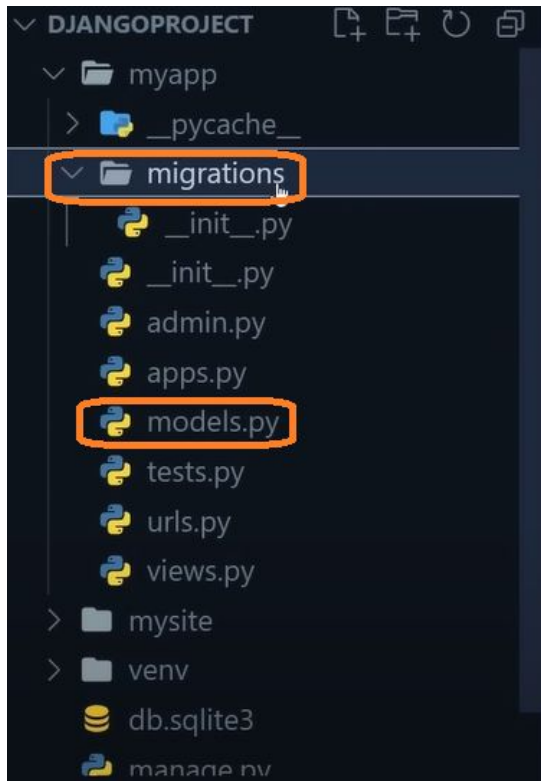
System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly until you
apply the migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
August 30, 2022 - 20:28:27
Django version 4.1, using settings 'mysite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.

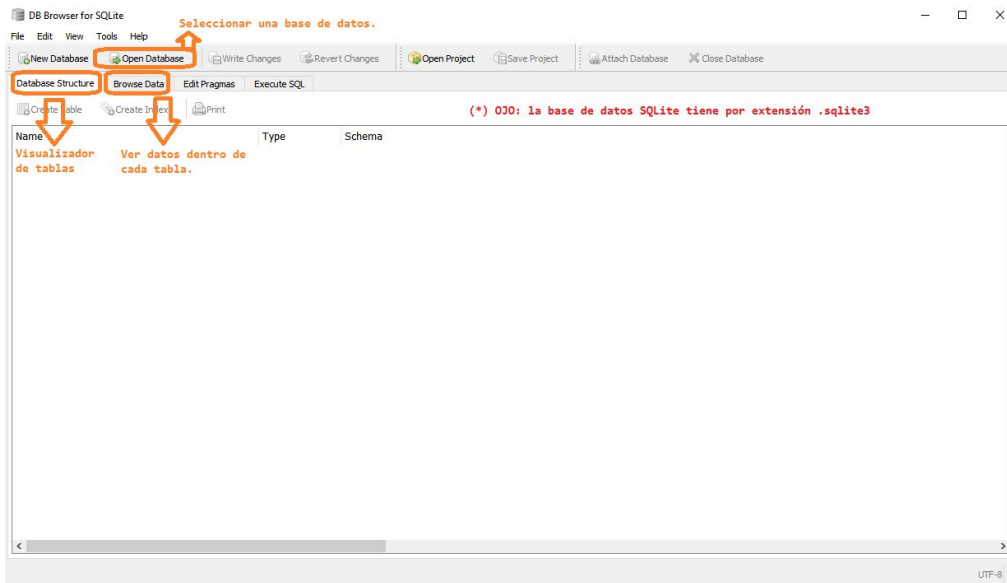
C:\Users\fazt\Desktop\django\project>python manage.py makemigrations
No changes detected

C:\Users\fazt\Desktop\django\project>
```

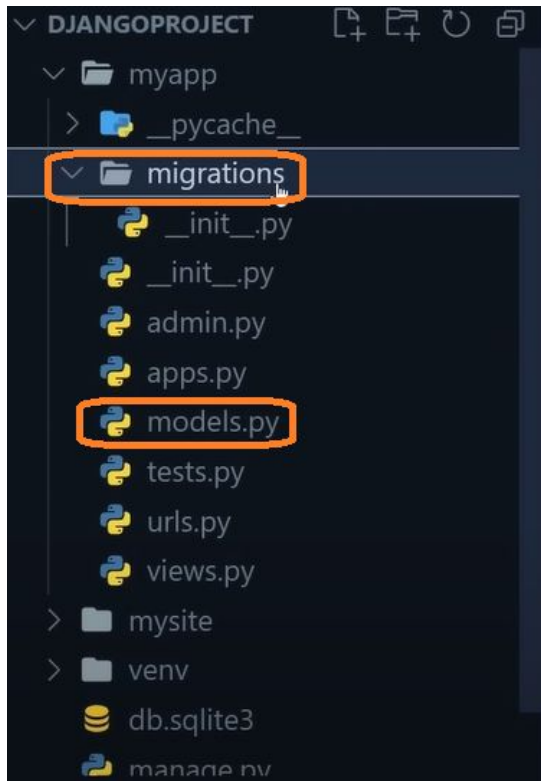
Conociendo la estructura: models.py



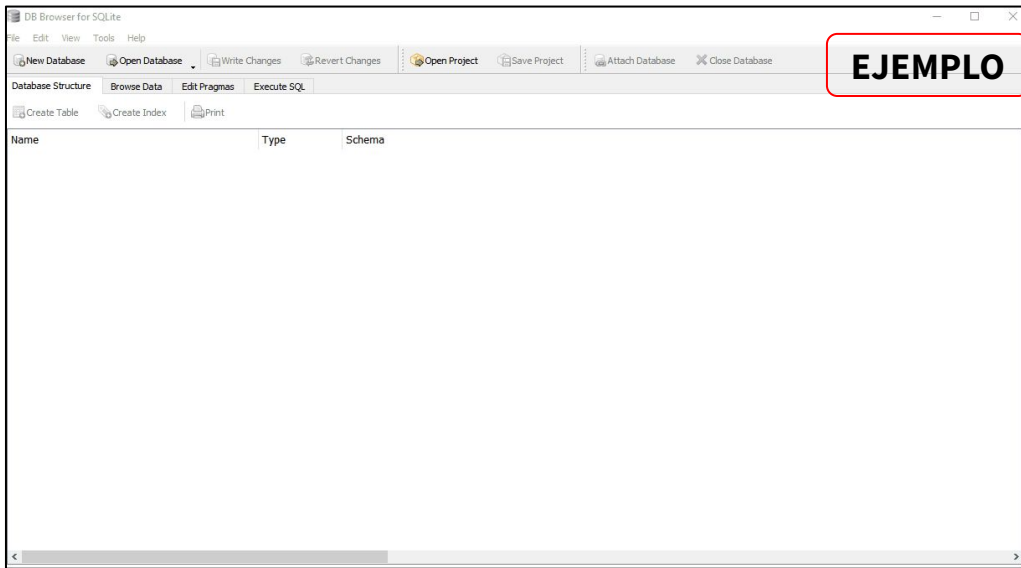
- Para ver las tablas de la base de datos (en este caso, SQLite3), debemos usar un cliente SQL. Uno muy fácil de utilizar es [DB Browser for SQLite](#).



Conociendo la estructura: models.py



- Para ver las tablas de la base de datos (en este caso, SQLite3), debemos usar un cliente SQL. Uno muy fácil de utilizar es [DB Browser for SQLite](#).



Conociendo la estructura: templates

- En Django, un **template** es básicamente **un archivo de texto que contiene código HTML mezclado con etiquetas especiales de Django**. Estas etiquetas permiten la inserción dinámica de datos desde la base de datos o desde el contexto proporcionado por las vistas.
- **OJO: no se trata de HTML únicamente**, sino que dentro de cada template se puede operar con la lógica que se necesite, según los datos que reciba. Esto significa que **podremos utilizar ciclos, condicionales, expresiones regulares, operadores booleanos y un largo etc.**

Conociendo la estructura: templates



```
from django.shortcuts import render
from .models import Post

def post_list(request):
    posts = Post.objects.all()
    return render(request, 'blog/post_list.html',
    {
        'posts': posts
    })
```

views.py

```
<!DOCTYPE html>
<html>
<head>
    <title>Lista de Publicaciones</title>
</head>
<body>
    <h1>Lista de Publicaciones</h1>
    <ul>
        {% for post in posts %}
            <li>{{ post.title }}</li>
        {% endfor %}
    </ul>
</body>
</html>
```

post_list.html

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

urlpatterns = [
    path('posts/', post_list, name='post_list'),
]
```

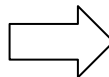
urls.py

Conociendo la estructura: *template inheritance*



- A medida que un proyecto escala, es importante recordar 2 conceptos clave: **DRY** y **KISS**.
- **DRY** (Don't Repeat Yourself -no te repitas-) es un principio que promueve la reducción en la duplicación de código. **El objetivo es escribir código de manera que cada parte del conocimiento o lógica esté representada en un único lugar en el sistema.**

```
def calcular_area_circulo(radio):  
    return math.pi * radio ** 2  
  
radio = 5  
area = calcular_area_circulo(radio)  
print("El área del círculo es:", area)
```



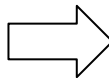
Una forma de representarlo es mediante las **funciones.**

Conociendo la estructura: *template inheritance*



- **KISS** (Keep It *Short* and Simple**) es otro principio de diseño que aboga por mantener las soluciones simples y evitar la complejidad innecesaria. **La idea es que las soluciones simples son más fáciles de entender, mantener y depurar (“si no podés explicarlo en forma sencilla, es que no lo entendés.”).**

```
texto = "Este es un TEXTO de Ejemplo"  
texto_minusculas = texto.lower()  
print("Texto en minúsculas:", texto_minusculas)
```



Supongamos que debemos **escribir un programa que convierta una cadena a minúsculas**. En lugar de utilizar métodos complicados o bibliotecas externas, **usamos el método `lower()`**.

Conociendo la estructura: *template inheritance*



- La **herencia de *templates*** (*template inheritance*) es una característica que permite definir un *template* **base** con elementos comunes, como la estructura HTML básica, y luego **extender éste en otros *templates* específicos para cada página de la aplicación.**
- **¿Para qué sirve todo esto? Para evitar REPETIR CÓDIGO.**
 - Se genera un archivo **base.html** con lo “general” de todas las páginas.
 - Se generan **n** archivos para las distintas páginas, que heredarán (*extends*) las características de base.html, y se sobrescribirá lo que haga falta.

Conociendo la estructura: *template inheritance*



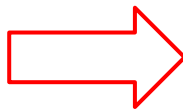
```
<!DOCTYPE html>
<html>
<head>
  <title>{% block title %}Mi Sitio Web{% endblock %}</title>
</head>
<body>
  <header>
    <h1>Mi Sitio Web</h1>
  </header>

  <nav>
    <ul>
      <li><a href="/">Inicio</a></li>
      <li><a href="/acerca_de/">Acerca de</a></li>
      <li><a href="/contacto/">Contacto</a></li>
    </ul>
  </nav>

  <main>
    {% block content %}
    {% endblock %}
  </main>

  <footer>
    <p>Derechos de autor © 2024 Mi Sitio Web</p>
  </footer>
</body>
</html>
```

BASE.HTML



Los bloques marcados **en rojo** son los “heredables” (*sobreescribibles*) por los HTMLs que tomen como **padre** a **base.html**.

Ejemplo en el siguiente *slide*.

base.html

Conociendo la estructura: *template inheritance*



about.html
hereda
(extends) toda
la estructura de
base.html.

Se **sobreescribe** el
título de la página.

```
{% extends 'base.html' %}  
  
{% block title %}Acerca de Nosotros{% endblock %}  
  
{% block content %}  
    <h2>Acerca de Nosotros</h2>  
    <p>Somos una empresa dedicada a...</p>  
{% endblock %}
```

about.html

Se **sobreescribe** el
contenido de la
página.

PRIMEROS PASOS: PARTE I



Primeros pasos: *virtual environments*



- Imaginemos que estamos trabajando con múltiples proyectos de Python:
 - ❖ Un e-commerce (Django).
 - ❖ Una aplicación de escritorio (Tkinter).
 - ❖ Un blog (Flask).
 - ❖ Web Scraping (Scrapy).

Está claro que no todos los proyectos trabajan con las mismas tecnologías, versión de Python, versión del gestor de dependencias, paquetes, etc. Es más, si cambiamos de proyecto, lo más común es tener que instalar “*todo*” *de nuevo*, perdiendo tiempo y esfuerzo. Es aquí donde los *virtual environments* cobran relevancia.

Primeros pasos: *virtual environments*



- Un entorno virtual (*virtual env*) es una manera de aislar la configuración y código del proyecto, separándolo de otros en el mismo dispositivo.

Gráficamente, se vería de la siguiente manera:

Django CRUD Python 3.9 + PIP + packages	Tkinter Project Python 3.8 + PIP + packages
Scrapy Python 3.8 + PIP + packages	Flask CRUD Python 3.10 + PIP + packages

Primeros pasos: *virtual environments*



- Comandos básicos para la generación de un entorno virtual (dentro de la carpeta del proyecto):

- **Paso 1 -**

- ***pip install virtualenv***: instalar módulo de PIP de virtualenv.

Utilizar ***virtualenv --version*** (doble guión) para verificar la correcta instalación del mismo.

Primeros pasos: *virtual environments*



- Comandos básicos para la generación de un entorno virtual (dentro de la carpeta del proyecto):
 - **Paso 2 -**
 - ***virtualenv venv***: crea el entorno virtual en la carpeta donde se lanza el comando. **NO** estará operativo hasta que se active (paso 3).

Primeros pasos: *virtual environments*



- Comandos básicos para la generación de un entorno virtual (dentro de la carpeta del proyecto):

- **Paso 3 -**

- **`.\venv\Scripts\activate`: activa el entorno virtual.**

Verificar con `python -version` (doble guión) que el entorno fue generado con éxito.

Primeros pasos: *virtual environments*



```
C:\Users\fazt\Desktop\django project>.\venv\Scripts\activate  
(venv) C:\Users\fazt\Desktop\django project>
```



Al ejecutar el script *activate*, automáticamente el entorno virtual queda establecido y se podrá visualizar (venv) antes de la ruta donde se encuentra el proyecto.

```
(venv) C:\Users\fazt\Desktop\django project>python --version  
Python 3.10.6
```



Esta versión de Python es exclusiva del proyecto “django project”.

Primeros pasos: instalación Django



Estando dentro de una terminal (Visual Studio Code, PowerShell, o similares):

Paso 1-A: pip **install** django

Instala el módulo de Django, en la última versión [LOV](#) disponible.

Paso 1-B (OPCIONAL): pip **install** django==4.2.10

Instala el módulo de Django en la versión 4.2.10.

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

(mysite-env) E:\Django Project>python -m pip install django
Collecting django
  Using cached Django-3.2.6-py3-none-any.whl (7.9 MB)
Collecting pytz
  Using cached pytz-2021.1-py2.py3-none-any.whl (510 kB)
Collecting sqlparse>=0.2.2
  Using cached sqlparse-0.4.1-py3-none-any.whl (42 kB)
Collecting asgiref<4,>=3.3.2
  Using cached asgiref-3.4.1-py3-none-any.whl (25 kB)
Installing collected packages: sqlparse, pytz, asgiref, django
Successfully installed asgiref-3.4.1 django-3.2.6 pytz-2021.1 sqlparse-0.4.1
```

Primeros pasos: instalación Django



Estando dentro de una terminal (Visual Studio Code, PowerShell, o similares):

Paso 2 (de requerirse): pip **install -r requirements.txt**

Instala todas las dependencias incluidas en el archivo *requirements.txt*.

```
C:\Users\gutkrish\Documents\Python\python general>pip install -r requirements.txt
Collecting pandas
  Downloading pandas-1.5.0-cp310-cp310-win_amd64.whl (10.4 MB)
    ----- 10.4/10.4 MB 8.9 MB/s eta 0:00:00
Requirement already satisfied: pillow in c:\program files\python310\lib\site-packages (from -r r
) (9.2.0)
Collecting numpy
  Downloading numpy-1.23.4-cp310-cp310-win_amd64.whl (14.6 MB)
    ----- 14.6/14.6 MB 18.2 MB/s eta 0:00:00
Collecting python-dateutil>=2.8.1
  Downloading python_dateutil-2.8.2-py2.py3-none-any.whl (247 kB)
    ----- 247.7/247.7 KB 14.8 MB/s eta 0:00:00
Collecting pytz>=2020.1
  Downloading pytz-2022.4-py2.py3-none-any.whl (500 kB)
    ----- 500.8/500.8 KB 10.4 MB/s eta 0:00:00
Collecting six>=1.5
  Downloading six-1.16.0-py2.py3-none-any.whl (11 kB)
Installing collected packages: pytz, six, numpy, python-dateutil, pandas
Successfully installed numpy-1.23.4 pandas-1.5.0 python-dateutil-2.8.2 pytz-2022.4 six-1.16.0
```


Primeros pasos: instalación Django



Estando dentro de una terminal (Visual Studio Code, PowerShell, o similares):

Paso 3: `python manage.py runserver 3000` ➡
Ejecuta el servidor local de Django, en el puerto 3000 (configurable).

Verificar que el puerto se encuentre **disponible** en la PC.

Paso 4: ingresar a `http://localhost:3000`
URL/punto de acceso inicial de la aplicación.

```
C:\Windows\System32\cmd.exe - python manage.py runserver
C:\Python\RESTapi\django-rest>django-admin startproject mysite

C:\Python\RESTapi\django-rest>dir
Volume in drive C has no label.
Volume Serial Number is 0479-A0C6

Directory of C:\Python\RESTapi\django-rest

09/10/2020  02:11 PM    <DIR>          .
09/10/2020  02:11 PM    <DIR>          ..
09/10/2020  02:11 PM    <DIR>          mysite
               0 File(s)                0 bytes
               3 Dir(s)  69,145,935,872 bytes free

C:\Python\RESTapi\django-rest>cd mysite

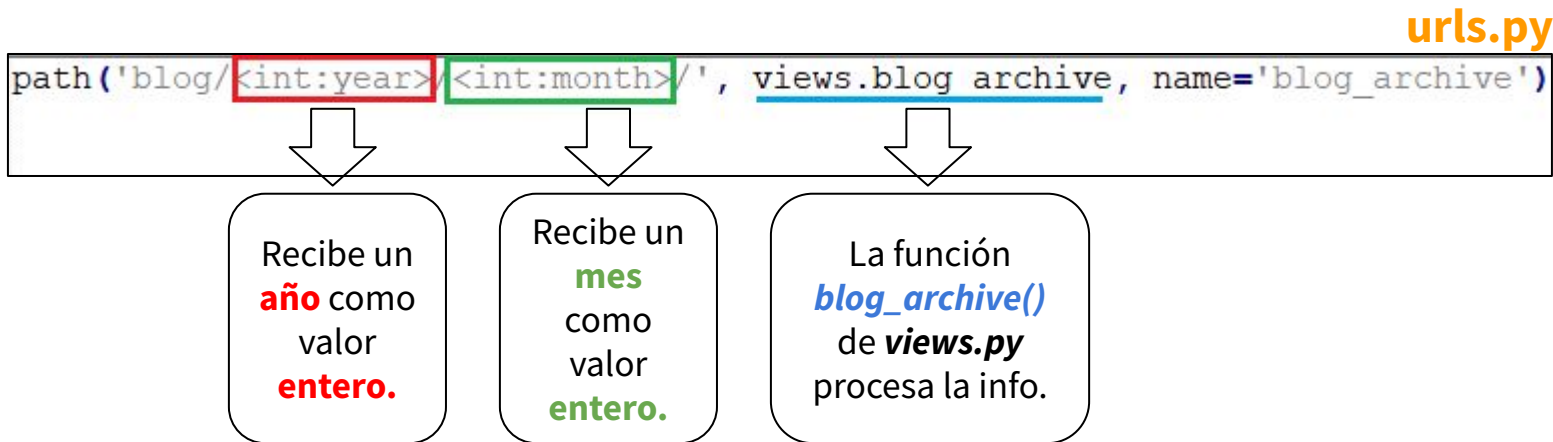
C:\Python\RESTapi\django-rest\mysite>python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin,
auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
September 10, 2020 - 14:12:50
Django version 3.1.1, using settings 'mysite.settings'
starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Primeros pasos: *params*

- Los *params* (parámetros) son datos que se pasan a una vista a través de la URL. Generalmente se definen en el archivo *urls.py*.



Primeros pasos: *params*



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

urlpatterns = [
    path('blog/<int:year>/<int:month>/', views.blog_archive, name='blog_archive'),
]
```

urls.py

```
from django.shortcuts import render

def blog_archive(request, year, month):
    context = {
        'year': year,
        'month': month,
    }
    return render(request, 'blog_archive.html', context)
```

views.py

```
<h1>Archivos del blog para {{ month }}/{{ year }}</h1>
```

template

El *url.py* está configurado para **recibir un año y mes**. Estos valores se pasan a la función *blog_archive(request, **year**, **month**)* que los guarda en un objeto/diccionario *<clave, valor>*, para que finalmente **se trasladen a un template que muestra su info**.

Primeros pasos: *render*



- **render()** se utiliza para renderizar un *template* junto con un contexto y devolver el resultado como una respuesta [HTTP](#). **Es una forma conveniente de generar respuestas HTML a partir de templates en las vistas.**
- Podemos traducir render() como “enviar”, pero... ¿qué enviamos?
 - (¿Quién me lo pide?) Un objeto *request*, que representa la **solicitud del cliente** (navegador).
 - (¿Qué voy a mostrar?) Un *template_name*, que indica el **nombre del template/HTML** a mostrar.
 - (¿Qué datos adicionales se requieren?) Opcionalmente, un *context*, que contiene un **diccionario <clave, valor> con los datos que se enviarán al template.**

Primeros pasos: *render*



```
def detalle_producto(request, producto_id):  
    # Supongamos que 'Producto' es un modelo que representa los productos en tu tienda.  
    producto = Producto.objects.get(id=producto_id)  
    contexto = {  
        'producto': producto,  
    }  
    return render(request, 'detalle_producto.html', contexto)
```

Se muestra el *template* **detalle_producto.html**, pasándole un diccionario con un producto en particular.

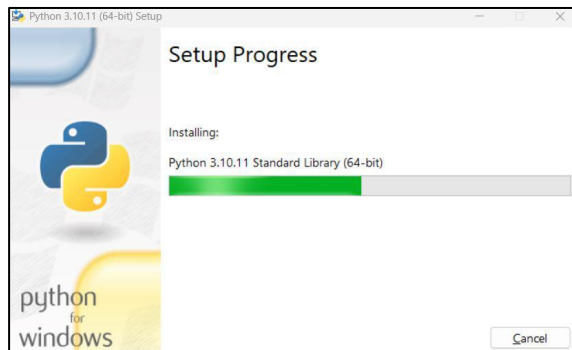
```
<!DOCTYPE html>  
<html lang="es">  
<head>  
    <meta charset="UTF-8">  
    <title>Detalle del Producto</title>  
</head>  
<body>  
    <h1>Detalle del Producto</h1>  
    <p>Nombre: {{ producto.nombre }}</p>  
    <p>Precio: {{ producto.precio }}</p>  
    <p>Descripción: {{ producto.descripcion }}</p>  
</body>  
</html>
```



Este *template* muestra el **nombre**, **precio** y **descripción** del producto, que se pasó anteriormente a través de la función *detalle_producto()*.

Primeros pasos: **Hola Mundo**

- Descarga de Python (última versión): <https://www.python.org/downloads/>



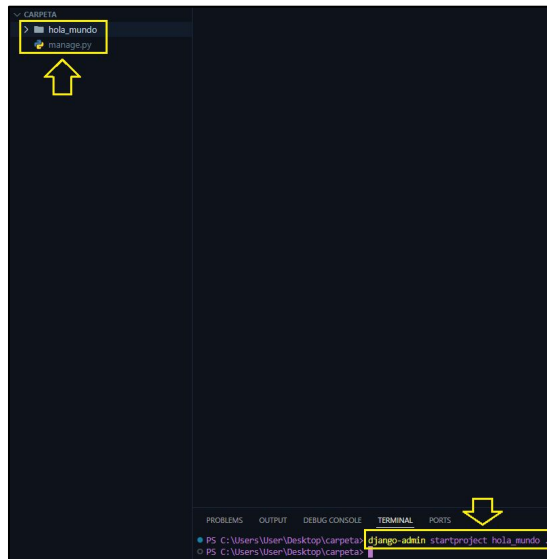
- Abrir una terminal o CMD e instalar Django: **pip install django**

```
C:\Windows\System32\cmd.exe

(mysite-env) E:\Django Project>python -m pip install Django
Collecting Django
  Using cached Django-3.2.6-py3-none-any.whl (7.9 MB)
Collecting pytz
  Using cached pytz-2021.1-py2.py3-none-any.whl (510 kB)
Collecting sqlparse>=0.2.2
  Using cached sqlparse-0.4.1-py3-none-any.whl (42 kB)
Collecting asgiref<4,>=3.3.2
  Using cached asgiref-3.4.1-py3-none-any.whl (25 kB)
Installing collected packages: sqlparse, pytz, asgiref, Django
Successfully installed Django-3.2.6 asgiref-3.4.1 pytz-2021.1 sqlparse-0.4.1
```

Primeros pasos: **Hola Mundo**

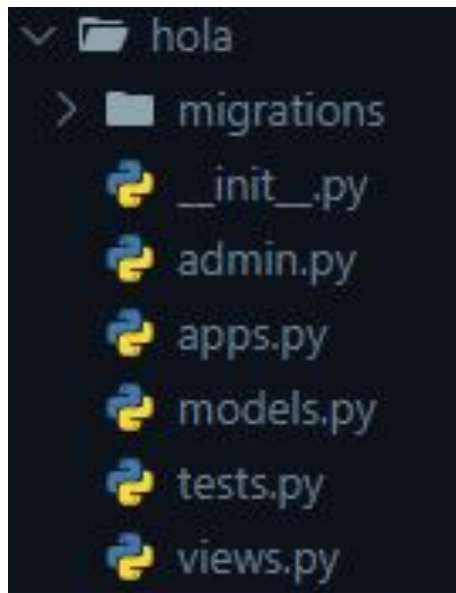
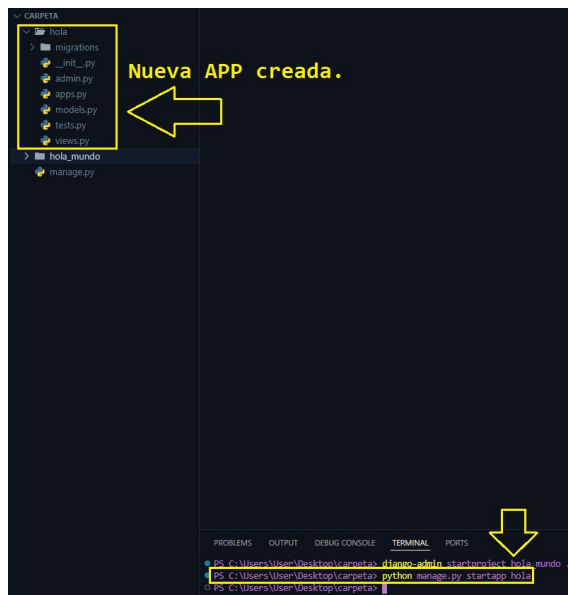
- Crear o posicionarse sobre la carpeta donde se creará el proyecto e instalar Django, con **VS Code**, usando ***django-admin startproject hola_mundo*** .



Primeros pasos: **Hola Mundo**



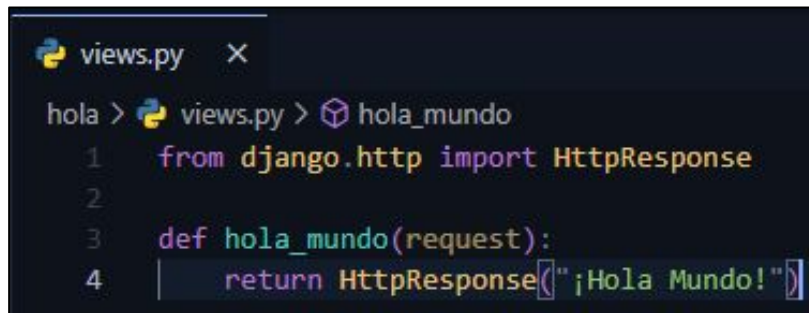
- Crear una nueva aplicación llamada “hola” usando ***python manage.py startapp hola***



Primeros pasos: Hola Mundo

- Abrir el módulo `views.py` y definir la siguiente función (NO olvidar importar `from`
`django.http import HttpResponse`)

```
def hola_mundo(request):  
    return HttpResponse(";Hola Mundo!")
```



```
views.py X  
hola > views.py > hola_mundo  
1 from django.http import HttpResponse  
2  
3 def hola_mundo(request):  
4     return HttpResponse(";Hola Mundo!")
```

Primeros pasos: Hola Mundo

- Editar el archivo `urls.py`, agregando el `path` que corresponda (NO olvidar importar

```
from hola.views import hola_mundo)
```

```
urlpatterns = [  
    path('hola/', hola_mundo),  
]
```



```
views.py  urls.py  
hola_mundo > urls.py > ...  
1  from django.contrib import admin  
2  from django.urls import path  
3  from hola.views import hola_mundo  
4  
5  urlpatterns = [  
6      path('admin/', admin.site.urls),  
7      path('hola/', hola_mundo)  
8  ]
```

Primeros pasos: Hola Mundo

- ¡Eso es todo! Guardamos los archivos y, desde la terminal, ejecutamos el servidor con **python manage.py runserver 3000**
- **Para ver el resultado, ingresar a <http://127.0.0.1:3000/hola/>**

```
PS C:\Users\User\Desktop\carpeta> python manage.py runserver 3000
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
• April 21, 2024 - 18:08:00
Django version 5.0.4, using settings 'hola_mundo.settings'
Starting development server at http://127.0.0.1:3000/ ➡ Servidor ejecutado con éxito.
Quit the server with CTRL-BREAK.
```



PRIMEROS PASOS: PARTE II



Primeros pasos: ciclo *FOR*



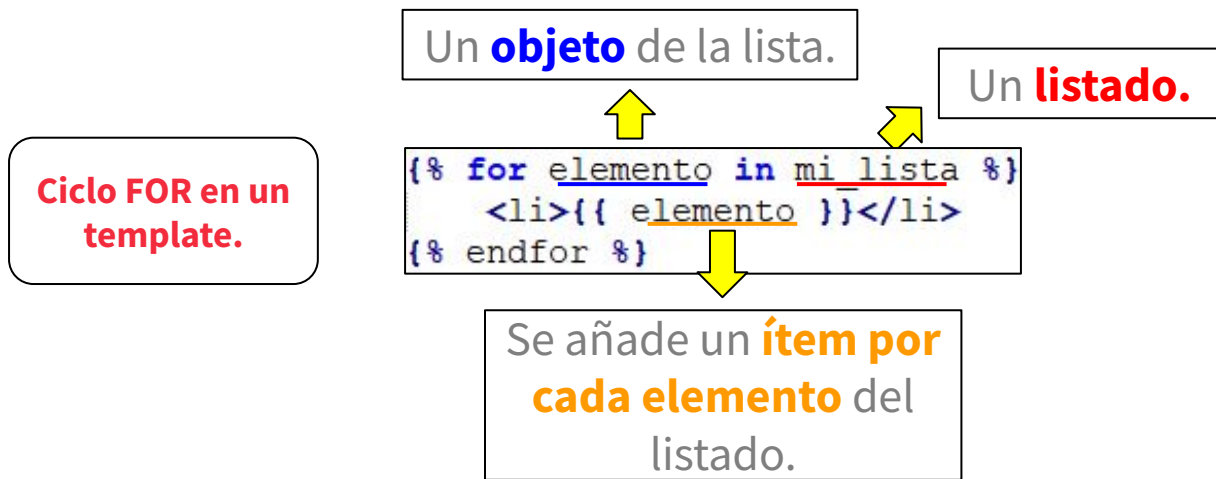
- El ciclo **FOR** es útil para iterar sobre conjuntos de datos en los *templates* HTML y renderizar contenido dinámico. **Existen también el ciclo WHILE, pero no es recomendado (¿por qué?).**
- La sintaxis es muy similar al **recorrido por elemento** o **foreach**.

```
{% for elemento in mi_lista %}  
    <li>{{ elemento }}</li>  
{% endfor %}
```

Primeros pasos: ciclo *FOR*

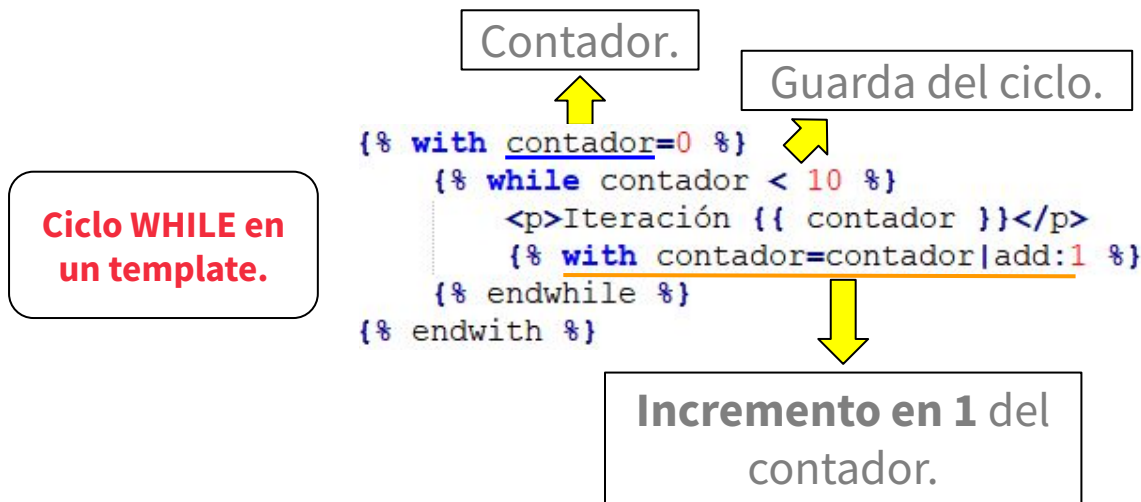


- El ciclo **FOR** es útil para iterar sobre conjuntos de datos en los *templates* HTML y renderizar contenido dinámico. **Existen también el ciclo WHILE, pero no es recomendado (¿por qué?).**
- La sintaxis es muy similar al **recorrido por elemento** o **foreach**.



Primeros pasos: ciclo *FOR*

- El ciclo **FOR** es útil para iterar sobre conjuntos de datos en los *templates* HTML y renderizar contenido dinámico. **Existen también el ciclo WHILE, pero no es recomendado (¿por qué?).**



Primeros pasos: *if-else*

- La estructura condicional **if-else** permite tomar una o varias decisiones, en función de una guarda.

```
from django.shortcuts import render

def mi_vista(request):
    mi_lista = [1, 2, 3] # Podría ser cualquier lista
    return render(request, 'mi_template.html', {'mi_lista': mi_lista})
```

```
<!DOCTYPE html>
<html>
<head>
    <title>Mi Template</title>
</head>
<body>
    {% if mi_lista|length > 0 %}
        <p>La lista contiene elementos:</p>
        <ul>
            {% for elemento in mi_lista %}
                <li>{{ elemento }}</li>
            {% endfor %}
        </ul>
    {% else %}
        <p>La lista está vacía.</p>
    {% endif %}
</body>
</html>
```


Primeros pasos: *if-else*

- La estructura condicional **if-else** permite evaluar una o más condiciones y, a su vez, tomar una o más decisiones:

views.py

```
from django.shortcuts import render

def mi_vista(request):
    mi_lista = [1, 2, 3] # Podría ser cualquier lista
    return render(request, 'mi_template.html', {'mi_lista': mi_lista})
```

template

```
<!DOCTYPE html>
<html>
<head>
  <title>Mi Template</title>
</head>
<body>
  {% if mi_lista|length > 0 %}
    <p>La lista contiene elementos:</p>
    <ul>
      {% for elemento in mi_lista %}
        <li>{{ elemento }}</li>
      {% endfor %}
    </ul>
  {% else %}
    <p>La lista está vacía.</p>
  {% endif %}
</body>
</html>
```

**Limitadores
del bloque
condicional.**

**Guarda del
condicional.**

**Rama
AFIRMATIVA.**

**Rama NEGATIVA
(OPCIONAL).**

Primeros pasos: *if-else*

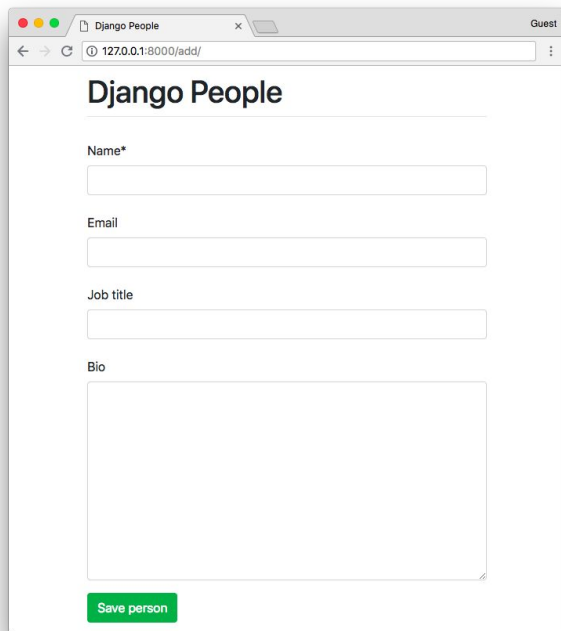
- También se pueden usar los operadores **AND**, **OR** y **NOT**:

Se declara una variable **numero**, con valor **7**.

Template

```
{% with numero=7 %}  
  {% if numero != 0 and numero % 2 == 0 %}  
    <p>El número es par.</p>  
  {% elif numero != 0 and numero % 2 != 0 %}  
    <p>El número es impar.</p>  
  {% else %}  
    <p>El número es igual a cero.</p>  
  {% endif %}  
{% endwith %}
```

Primeros pasos: *forms* **2 STEP**



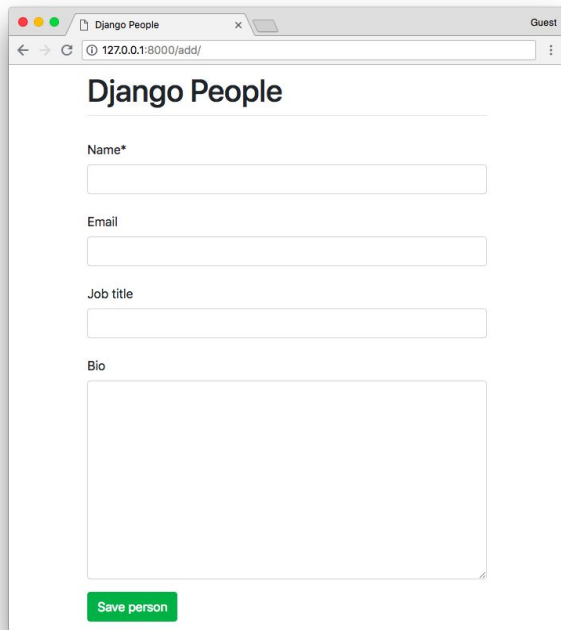
The screenshot shows a web browser window with the title 'Django People'. The address bar shows '127.0.0.1:8000/add/'. The form has the following fields:

- Name* (text input)
- Email (text input)
- Job title (text input)
- Bio (text area)

At the bottom of the form is a green button labeled 'Save person'.

- Los **formularios** son una forma de manejar la **entrada de datos del usuario** en una aplicación web. Permiten recopilar información de los usuarios, como nombres, direcciones de correo electrónico, contraseñas y más.
- Podemos crear formularios basándonos en un **modelo de Django** (simple, con **forms.Form**; basado en bases de datos, con **forms.ModelForm**) o usando alguno externo como **Bootstrap Forms**.
- **La principal diferencia entre ambos es la interactividad y el impacto visual.**

Primeros pasos: *forms*



Django People

Name*

Email

Job title

Bio

Save person

- Regularmente, se emplean 2 métodos: **GET** y **POST**.
- **GET** se usa generalmente para **mostrar datos**; **POST**, para **enviar datos** (¿por qué? 🤔).

`127.0.0.1:8000/create_task?title=a&description=b`




La info. que se envía a través de **GET**, se representa en la URL de la forma **<parametro=valor>**

Primeros pasos: *forms*

- Veamos un ejemplo con **forms.Form (Django)**:
 - En primer lugar, definimos el formulario:

3 campos: nombre,
correo y mensaje



```
from django import forms

class ContactoForm(forms.Form):
    nombre = forms.CharField(label='Nombre', max_length=100)
    correo = forms.EmailField(label='Correo electrónico', max_length=100)
    mensaje = forms.CharField(label='Mensaje', widget=forms.Textarea)
```

ContactoForm.py

Primeros pasos: *forms*

- Veamos un ejemplo con **forms.Form (Django)**:
 - En segundo lugar, creamos la vista del formulario:

```
from django.shortcuts import render
from .forms import ContactoForm

def contacto(request):
    if request.method == 'POST':
        formulario = ContactoForm(request.POST)
        if formulario.is_valid():
            # Procesar los datos del formulario
            nombre = formulario.cleaned_data['nombre']
            correo = formulario.cleaned_data['correo']
            mensaje = formulario.cleaned_data['mensaje']
            # Aquí podrías guardar los datos en la base de datos, enviar un correo, etc.
            return render(request, 'gracias.html')
        else:
            formulario = ContactoForm()
    return render(request, 'contacto.html', {'formulario': formulario})
```

views.py

Primeros pasos: *forms*

- Veamos un ejemplo con **forms.Form (Django)**:
 - En segundo lugar, creamos la vista del formulario:

```
from django.shortcuts import render
from .forms import ContactoForm

def contacto(request):
    if request.method == 'POST':
        formulario = ContactoForm(request.POST)
        if formulario.is_valid():
            # Procesar los datos del formulario
            nombre = formulario.cleaned_data['nombre']
            correo = formulario.cleaned_data['correo']
            mensaje = formulario.cleaned_data['mensaje']
            # Aquí podrías guardar los datos en la base de datos, enviar un correo, etc.
            return render(request, 'gracias.html')
        else:
            formulario = ContactoForm()
    return render(request, 'contacto.html', {'formulario': formulario})
```

Si el formulario es de tipo **POST**,
extraemos la info. (rama afirmativa del IF).

views.py

Primeros pasos: *forms*

- Veamos un ejemplo con **forms.Form (Django)**:
 - En segundo lugar, creamos la vista del formulario:

```
from django.shortcuts import render
from .forms import ContactoForm

def contacto(request):
    if request.method == 'POST':
        formulario = ContactoForm(request.POST)
        if formulario.is_valid():
            # Procesar los datos del formulario
            nombre = formulario.cleaned_data['nombre']
            correo = formulario.cleaned_data['correo']
            mensaje = formulario.cleaned_data['mensaje']
            # Aquí podrías guardar los datos en la base de datos, enviar un correo, etc.
            return render(request, 'gracias.html')
        else:
            formulario = ContactoForm()
    return render(request, 'contacto.html', {'formulario': formulario})
```

Si el formulario es de tipo **POST**,
extraemos la info. (rama afirmativa del IF).

Info a extraer.

views.py

Primeros pasos: *forms*

- Veamos un ejemplo con **forms.Form (Django)**:
 - En segundo lugar, creamos la vista del formulario:

Acá se
podrían
guardar los
datos en la
BD.

```
from django.shortcuts import render
from .forms import ContactoForm

def contacto(request):
    if request.method == 'POST':
        formulario = ContactoForm(request.POST)
        if formulario.is_valid():
            # Procesar los datos del formulario
            nombre = formulario.cleaned_data['nombre']
            correo = formulario.cleaned_data['correo']
            mensaje = formulario.cleaned_data['mensaje']
            # Aquí podrías guardar los datos en la base de datos, enviar un correo, etc.
            return render(request, 'gracias.html')
        else:
            formulario = ContactoForm()
            return render(request, 'contacto.html', {'formulario': formulario})
```

Si el formulario es de tipo **POST**,
extraemos la info. (rama afirmativa del IF).

Info a extraer.

views.py

Primeros pasos: *forms* 2 STEP

- Veamos un ejemplo con **forms.Form (Django)**:
 - En segundo lugar, creamos la vista del formulario:

```
from django.shortcuts import render
from .forms import ContactoForm

def contacto(request):
    if request.method == 'POST':
        formulario = ContactoForm(request.POST)
        if formulario.is_valid():
            # Procesar los datos del formulario
            nombre = formulario.cleaned_data['nombre']
            correo = formulario.cleaned_data['correo']
            mensaje = formulario.cleaned_data['mensaje']
            # Aquí podrías guardar los datos en la base de datos, enviar un correo, etc.
            return render(request, 'gracias.html')
        else:
            formulario = ContactoForm()
            return render(request, 'contacto.html', {'formulario': formulario})
```

Si el formulario es de tipo **POST**,
extraemos la info. (rama afirmativa del IF).

Info a extraer.

Acá se
podrían
guardar los
datos en la
BD.

Se muestra un
template de
salida
(gracias) al
usuario.

views.py

Primeros pasos: *forms* 2 STEP

- Veamos un ejemplo con **forms.Form (Django)**:
 - En segundo lugar, creamos la vista del formulario:

```
from django.shortcuts import render
from .forms import ContactoForm

def contacto(request):
    if request.method == 'POST':
        formulario = ContactoForm(request.POST)
        if formulario.is_valid():
            # Procesar los datos del formulario
            nombre = formulario.cleaned_data['nombre']
            correo = formulario.cleaned_data['correo']
            mensaje = formulario.cleaned_data['mensaje']
            # Aquí podrías guardar los datos en la base de datos, enviar un correo, etc.
            return render(request, 'gracias.html')
        else:
            formulario = ContactoForm()
            return render(request, 'contacto.html', {'formulario': formulario})
```

Si el formulario es de tipo **POST**,
extraemos la info. (rama afirmativa del IF).

Info a extraer.

Acá se
podrían
guardar los
datos en la
BD.

En esta rama,
el método es
GET. Creamos
un nuevo
formulario
vacío.

Se muestra un
template de
salida
(gracias) al
usuario.

views.py

Primeros pasos: *forms* 2 STEP

- Veamos un ejemplo con **forms.Form (Django)**:
 - En segundo lugar, creamos la vista del formulario:

```
from django.shortcuts import render
from .forms import ContactoForm

def contacto(request):
    if request.method == 'POST':
        formulario = ContactoForm(request.POST)
        if formulario.is_valid():
            # Procesar los datos del formulario
            nombre = formulario.cleaned_data['nombre']
            correo = formulario.cleaned_data['correo']
            mensaje = formulario.cleaned_data['mensaje']
            # Aquí podrías guardar los datos en la base de datos, enviar un correo, etc.
            return render(request, 'gracias.html')
        else:
            formulario = ContactoForm()
            return render(request, 'contacto.html', {'formulario': formulario})
```

Si el formulario es de tipo **POST**,
extraemos la info. (rama afirmativa del IF).

Info a extraer.

Acá se
podrían
guardar los
datos en la
BD.

En esta rama,
el método es
GET. Creamos
un nuevo
formulario
vacío.

Se muestra un
template de
salida
(gracias) al
usuario.

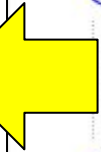
Se muestra el **formulario vacío**.

views.py

Primeros pasos: *forms*

- Veamos un ejemplo con **forms.Form (Django)**:
 - En tercer lugar, creamos el template del formulario:

Se genera un formulario en la web.



```
<!DOCTYPE html>
<html>
<head>
  <title>Contacto</title>
</head>
<body>
  <h1>Formulario de Contacto</h1>
  <form method="post">
    {% csrf_token %}
    {{ formulario.as_p }}
    <button type="submit">Enviar</button>
  </form>
</body>
</html>
```

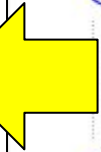
contacto.html

Primeros pasos: *forms*

- Veamos un ejemplo con **forms.Form (Django)**:
 - En tercer lugar, creamos el template del formulario:

Se genera un formulario en la web.

formulario.**as_p** renderiza el formulario utilizando **1 párrafo para cada campo.**



```
<!DOCTYPE html>
<html>
<head>
  <title>Contacto</title>
</head>
<body>
  <h1>Formulario de Contacto</h1>
  <form method="post">
    {% csrf_token %}
    {{ formulario.as_p }}
    <button type="submit">Enviar</button>
  </form>
</body>
</html>
```

contacto.html

Primeros pasos: *forms*

- Veamos un ejemplo con **forms.Form (Django)**:
 - En tercer lugar, creamos el template del formulario:

Se genera un formulario en la web.

formulario.**as_p** renderiza el formulario utilizando **1 párrafo para cada campo.**

```
<!DOCTYPE html>
<html>
<head>
  <title>Contacto</title>
</head>
<body>
  <h1>Formulario de Contacto</h1>
  <form method="post">
    {% csrf_token %}
    {{ formulario.as_p }}
    <button type="submit">Enviar</button>
  </form>
</body>
</html>
```

CSRF es una etiqueta/token que protege contra ataques de falsificación de requests entre sitios.

contacto.html

Primeros pasos: *forms*

- Veamos un ejemplo con **forms.Form (Django)**:
 - Para finalizar, creamos el template de gracias:

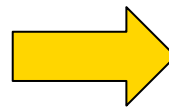
```
<!DOCTYPE html>
<html>
<head>
  <title>Gracias</title>
</head>
<body>
  <h1>¡Gracias por tu mensaje!</h1>
  <p>Nos pondremos en contacto pronto.</p>
</body>
</html>
```

gracias.html

Primeros pasos: *URL names*

- Sabemos que **el acceso a las distintas secciones (templates/páginas) de la aplicación se configura mediante un archivo `urls.py`** (enrutador).
- ¿Qué sucede si cambio la ruta de acceso? 🤔

```
urlpatterns = [  
    path('', home_view, name='home'),  
    path('contact/', contact_view),  
    path('about/', about_view),  
    path('social_view/', social_view),  
    path('product/', product_detail_view),  
    path('admin/', admin.site.urls),  
]
```



¿Qué pasa si el 2^{do} *path* (contact) pasa a llamarse **contact-info**?

Primeros pasos: *URL names*

- Sabemos que **el acceso a las distintas secciones (templates/páginas) de la aplicación se configura mediante un archivo `urls.py`** (enrutador).
- ¿Qué sucede si cambio la ruta de acceso? 🤖

Page not found (404)

Request Method: GET

Request URL: `http://localhost:8000/create_project/`

Primeros pasos: *URL names*

- Para evitar este error, a cada ruta se le puede asignar un nombre (URL name).
- Esto permite que, cada vez que se necesite acceder a una ruta en específico, se referencie por su nombre y NO por su *path*, con lo cual si la primera cambia, tendremos de igual manera un “acceso directo” a ella.

```
urlpatterns = [  
    path('', views.index, name="index"),  
    path('about/', views.about, name="about"),  
    path('hello/<str:username>', views.hello, name="hello"),  
    path('projects/', views.projects, name="projects"),  
    path('tasks/', views.tasks, name="tasks"),  
    path('create_task/', views.create_task, name="create_task"),  
    path('create_new_project/', views.create_project, name="create_project"),  
]
```

Nombres de referencia

Primeros pasos: *URL names*

```
<li>
  <a href="/about">About</a>
</li>
<li>
  <a href="/projects">Projects</a>
</li>
<li>
  <a href="/tasks">Tasks</a>
</li>
<li>
  <a href="/create_task">Create task</a>
</li>
```

ANTES

```
<nav>
  <ul>
    <li>
      <a href="{% url 'index' %}">Home</a>
    </li>
    <li>
      <a href="{% url 'about' %}">About</a>
    </li>
    <li>
      <a href="{% url 'projects' %}">Projects</a>
    </li>
    <li>
      <a href="{% url 'tasks' %}">Tasks</a>
    </li>
    <li>
      <a href="{% url 'create_task' %}">Create task</a>
    </li>
```

DESPUÉS

Primeros pasos: *static files*

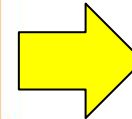
- Supongamos que se debe construir una aplicación que liste las materias de una determinada carrera.
- Un posible enfoque para resolver el problema es:

```
1) El usuario selecciona una carrera de un listado (Tecnicatura Univ. en Informática).  
2) El sistema valida que la carrera esté vigente.  
3) El sistema se conecta a la base de datos  
   y busca las materias vigentes de la carrera seleccionada.  
4) El sistema muestra una página con el listado de materias.  
5) FIN.
```

Primeros pasos: *static files*

- Supongamos que se debe construir una aplicación que liste las materias de una determinada carrera.
- Un posible enfoque para resolver el problema es:

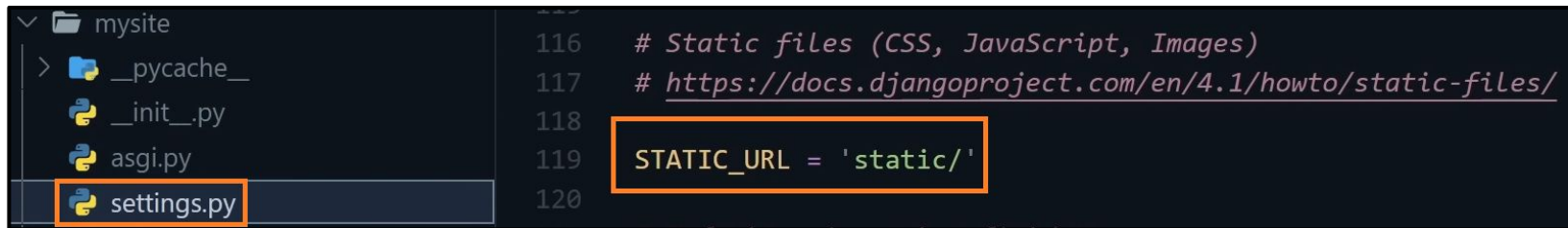
```
1) El usuario selecciona una carrera de un listado (Tecnicatura Univ. en Informática).  
2) El sistema valida que la carrera esté vigente.  
3) El sistema se conecta a la base de datos  
   y busca las materias vigentes de la carrera seleccionada.  
4) El sistema muestra una página con el listado de materias.  
5) FIN.
```



Este contenido se genera **dinámicamente (puede cambiar)**, dependiendo de la carrera y materias.

Primeros pasos: *static files*

- Pueden existir contenidos que siempre “son los mismos” (NO cambian) en la aplicación: algunas imágenes, hojas de estilo (CSS), audios, entre otros.
- **A estos archivos, que el servidor no procesa, se los conoce como *static files*** (archivos estáticos).

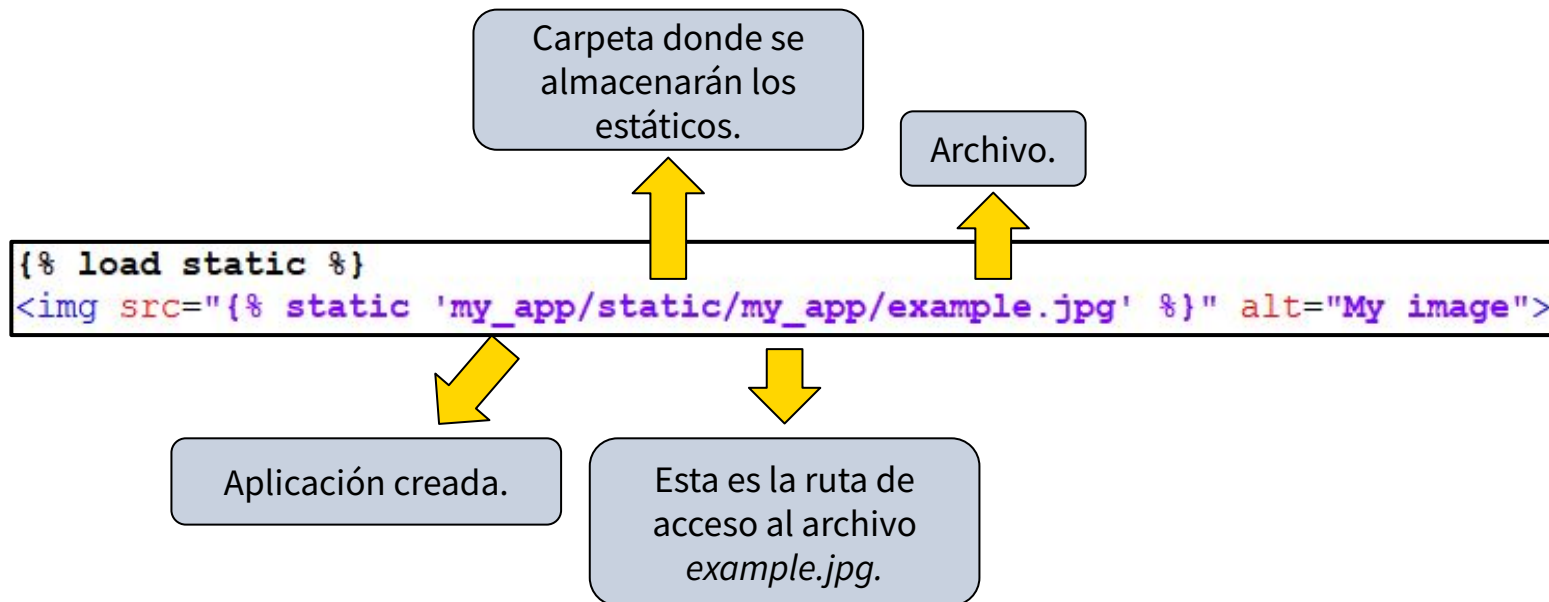


```
116 # Static files (CSS, JavaScript, Images)
117 # https://docs.djangoproject.com/en/4.1/howto/static-files/
118
119 STATIC_URL = 'static/'
120
```

Para habilitar los archivos estáticos, es obligatorio que la línea `STATIC_URL` de `settings.py` del proyecto esté habilitada (NO comentada).

Primeros pasos: *static files* 2 STEP

- Para hacer referencia a un archivo estático, desde un template, se utilizará la siguiente sintaxis base:



Primeros pasos: *static files*

- Se debe disponer de una carpeta llamada **static**, dentro de la aplicación en cuestión (crearla si no está presente):

Carpeta donde estarán los *static files*.



```
7 <meta http-equiv="X-UA-Compatible" content="IE=edge" >
8 <meta name="viewport" content="width=device-width,
9   initial-scale=1.0">
10 <title>Django Projects</title>
    <link rel="stylesheet" href="{% static '/css/main.css' %}"
```

Ruta de acceso al archivo estático.

ADICIONALES



Adicionales: *bootstrap library*



<http://tiny.cc/boostap-1>



<http://tiny.cc/bootstrap-2>



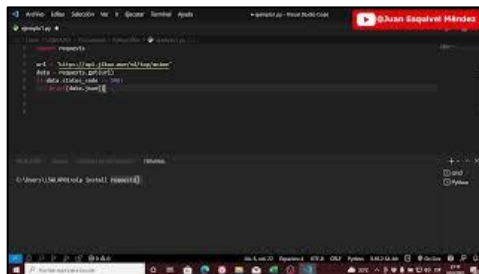
<http://tiny.cc/bootstrap-3>

Tutoriales sobre el uso de Bootstrap CSS Library (**el último está en inglés**).

Adicionales: ¿Qué son las APIs?



<http://tiny.cc/api-1>



<http://tiny.cc/api-2>



<http://tiny.cc/api-3>

Tutoriales sobre qué son las APIs (de izq. a der.): explicación simple (video 1), *request library* (video 2) y consumo/uso de la API de *Rick & Morty* (video 3).

FIN PARTE I: TEORÍA