

Introducción

Fuente: pildorasinformaticas

Django es un framework de desarrollo web de código abierto, escrito en Python, que respeta el patrón de diseño conocido como MVC (Modelo–Vista–Controlador). Fue desarrollado en origen para gestionar varias páginas orientadas a noticias de la World Company de Lawrence, Kansas, y fue liberada al público bajo una licencia BSD en julio de 2005; el framework fue nombrado en alusión al guitarrista de jazz gitano Django Reinhardt.

En junio de 2008 fue anunciado que la recién formada Django Software Foundation se haría cargo de Django en el futuro.

La meta fundamental de Django es facilitar la creación de sitios web complejos. Django pone énfasis en la reusabilidad, la conectividad y extensibilidad de componentes, el desarrollo rápido y el principio No te repitas (**DRY**, del inglés Don't Repeat Yourself). Python es usado en todas las partes del framework, incluso en configuraciones, archivos, y en los modelos de datos.

Instalación

Abrir cmd como administrador e insertar comando:

```
>>pip install django
```

Para saber que versiones tenemos instaladas:

```
>>pip freeze
```

Django divide el proyecto en varias apps que constan de diferentes partes, las apps realizan una tarea concreta. La carpeta que se crea al crear el proyecto es la **main app**. Esta división está enfocada a mejorar la modularización y reutilización de código entre proyectos.

Otra app en el proyecto agregaría una nueva característica a la aplicación web. Desde la main app se apuntará a las demás apps.

Se puede trabajar con el terminal dentro de VS Code, seleccionando **Python**.

Crear Proyecto

Crear carpeta, arrastrarla a **VS Code** y abrir un terminal, asegurarse de seleccionar **Python**. Si no aparece el terminal crear un archivo vacío con extensión .py y ejecutarlo. Se podrán insertar los comandos en el terminal que se despliega.

Escribiendo **django-admin** sabremos qué comandos tenemos disponibles. El comando para crear un proyecto es el siguiente:

```
>>django-admin startproject *nombreDelProyecto*
```

Archivos creados:

manage.py: Una utilidad de la línea de comandos que le permite interactuar con este proyecto Django de diferentes formas.

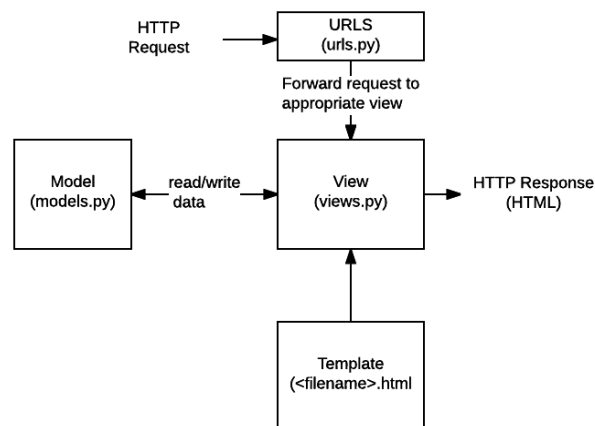
__init__.py: Un archivo vacío que le indica a Python que este directorio debería ser considerado como un paquete Python.

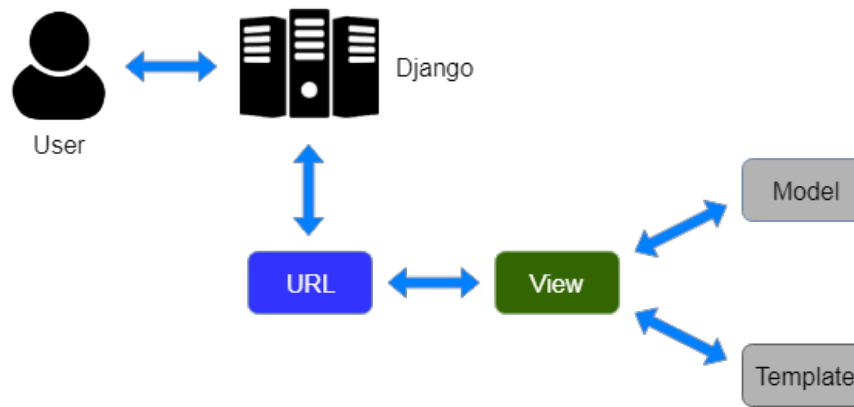
settings.py: Ajustes/configuración para este proyecto Django. Django settings le indicará todo sobre cómo funciona la configuración.

urls.py: Las declaraciones URL para este proyecto Django; una «tabla de contenidos» de su sitio basado en Django.

wsgi.py: Un punto de entrada para que los servidores web compatibles con WSGI puedan servir su proyecto.

Funcionamiento del framework





Creación de Vistas

Django trabaja con la clase **Request** para hacer peticiones y para enviar la respuesta utiliza **HttpResponse**.

Una función de vista, o "**view**" para abreviar, es simplemente una función de **Python** que toma una web **request** y devuelve una web **response**. Esta respuesta puede ser el contenido HTML de una página web, o una redirección, o un error 404, o un documento XML, o una imagen, etc. Ejemplo: Ten en cuenta que necesitas asociar una vista a una URL para verla como una página web.

Cada vez que se hace un request de una página, Django crea un objeto **HttpRequest** que contiene metadatos sobre el request. Después, Django carga la vista apropiada pasando el objeto **HttpRequest** como primer argumento a la función de la vista. Cada vista es responsable de devolver un **HttpResponse**. Ambos objetos están definidos en **django.http**.

Ejemplo práctico: Creación de Vistas

- 1) Crear archivo donde se almacenan las vistas que vayamos creando, se llama **views.py** por convención dentro de la carpeta de la main app. Posteriormente este paso **no lo haremos en la main app**, pero si en las apps.

En el archivo **views.py** importar **django.http** con:

```
from django.http import HttpResponse
```

- 2) Cada función dentro del archivo **views.py** es una vista. Se puede pasar HTML como respuesta pero **no se debe hacer así**, para eso se utilizarán las plantillas.

```
from django.http import HttpResponse

def saludo(request):
    return HttpResponse("Hola a todos!")

def saludo_html(request):
    documento="""<html><body><h1>Hola a todos!</h1></body></html>"""
```

```

    return HttpResponse(documento)

def despedida(request):
    return HttpResponse("Hasta luego!")

```

- 3) Hay que enlazar una URL para que devuelva la vista cuando accedemos a las vistas en el navegador. Se agrega dicha información en **urls.py**.

Cuando un usuario realiza una solicitud de una página de tu aplicación web, el controlador Django se encarga de buscar la vista correspondiente a través del archivo url.py, y luego devuelve la respuesta HTML o un error 404 no encontrado, si no se encuentra. En url.py, lo más importante es la lista "**urlpatterns**". Es donde se define el mapeo entre los URLs y las vistas.

```

from django.contrib import admin
from django.urls import path
from Proyecto.views import saludo,saludo_html,despedida

urlpatterns=[
    path("admin/",admin.site.urls),
    path("saludo/",saludo),
    path("saludohtml/",saludo_html),
    path("despedida/",despedida)
]

```

Primero hay que importar la función saludo del archivo correspondiente, y posteriormente enlazar un path para acceder a dicha vista dentro de la lista urlpatterns. No es necesario que el nombre del path coincida con el de la vista.

Posteriormente, ejecutar **python manage.py runserver** para correr el servidor local de pruebas y testear las vistas. Para cerrarlo, cliquear en la terminal y presionar CTRL+C.

```

localhost:8000/saludo/
localhost:8000/saludohtml/
localhost:8000/despedida/

```

Ejemplo Práctico: Trabajar con fechas (contenido dinámico)

(En **views.py**)

```

import datetime
def get_fecha(request):
    fecha_actual=datetime.datetime.now()
    documento="""<html><body><h1>Fecha: %s</h1></body></html>"""%fecha_actual
    return HttpResponse(documento)

```

(En **urls.py**) agregar al contenido del ejercicio anterior:

```

from django.contrib import admin
from django.urls import path
from Proyecto.views import saludo,saludo_html,despedida,get_fecha

urlpatterns=[
    path("admin/",admin.site.urls),
    path("saludo/",saludo),
    path("saludohtml/",saludo_html),
    path("despedida",despedida),
    path("fecha/",get_fecha)
]

```

Pasar parámetros por URL

Ejemplo: Calcular edad en un año determinado

(En **views.py**)

```

def calcular_edad(request,edad,agno):
    periodo=agno-2020
    edad_futura=edad+periodo
    documento="<html><body><h2>En el año %s tendrás %s años"%(agno,edad_futura)
    return HttpResponse(documento)

```

(En **urls.py**) agregar al contenido del ejercicio anterior:

```

from django.contrib import admin
from django.urls import path
from Proyecto.views import saludo,saludo_html,despedida,get_fecha,calcular_edad

urlpatterns=[
    path("admin/",admin.site.urls),
    path("saludo/",saludo),
    path("saludohtml/",saludo_html),
    path("despedida",despedida),
    path("fecha/",get_fecha),
    path("edades/<int:edad>/<int:agno>",calcular_edad)
]

```

Es necesario **castear** el año que se pasa por parámetro en la URL a entero, por defecto es string. Testear la vista corriendo el servidor y accediendo a: **localhost:8000/edades/60/2029**

Plantillas

Son cadenas de texto que tendrán HTML (casi siempre). Sirven para separar la lógica (datos) de la parte visual (presentación) de un documento web.

Las plantillas se guardan en un archivo diferente y se cargan desde la vista.

- 1) Crear objeto de tipo template en **views.py** e importar las clases **Template** y **Context**
- 2) Crear **contexto**, que son datos adicionales para el template (variables, funciones, etc, útiles cuando se usa contenido dinámico)

3) Renderizar el contenido con render(contexto)

Ejemplo práctico: Creación de plantillas

En la carpeta raíz del proyecto crear una carpeta llamada **templates**.

Proyecto

```
|--templates
|      |--*aquí irán las plantillas*
|--Proyecto
      |--__init__.py
      |--settings.py
      |--urls.py
      |--views.py
      |--wsgi.py
```

(En **plantilla.html** dentro de la carpeta templates)

```
<html>
  <body>
    <h1>Hola desde la plantilla!</h1>
  </body>
</html>
```

(En **views.py** modificando la view saludo creada anteriormente)

```
from django.http import HttpResponse
import datetime
from django.template import Template, Context

def saludo(request):
    arch=open("C:/Users/usuario/Desktop/Proyecto/Proyecto/templates/plantilla.html")
    plt=Template(arch.read())
    arch.close()
    ctx=Context()
    documento=plt.render(ctx)
    return HttpResponse(documento)
```

Ejemplo Práctico: Pasar variables del view a la plantilla.

Se utiliza en el ejemplo un diccionario para pasar la variable nombre a la plantilla. Se puede pasar cualquier tipo de dato, incluidas las listas.

(En **views.py** modificando la view saludo creada anteriormente)

```
def saludo(request):
    nombre="Juan"
    apellido= "Gonzalez"
    fecha = datetime.datetime.now()
    arch=open("C:/Users/usuario/Desktop/Proyecto/Proyecto/templates/plantilla.html")
    plt=Template(arch.read())
    arch.close()
    ctx=Context({"nombre_persona":nombre,"apellido_persona":apellido,"now":fecha})
    documento=plt.render(ctx)
    return HttpResponse(documento)
```

(En **plantilla.html** modificando la plantilla creada anteriormente)

```
<html>
    <body>
        <h1>Hola desde la plantilla!</h1>
        <p>Nombre: {{nombre_persona}} {{apellido_persona}}</p>
        <p>Fecha: {{now}}</p>
    </body>
</html>
```

También se puede pasar el valor directamente en el Contexto sin crear una variable.

```
ctx=Context({"nombre_persona":"Juan","apellido_persona":"Gonzalez","now":fecha})
```

Accediendo a propiedades de objetos

Al ser fecha un objeto datetime, se puede acceder a su propiedad day para mostrar solo el día actual

(En **plantilla.html** modificando la plantilla creada anteriormente)

```
<html>
    <body>
        <h1>Hola desde la plantilla!</h1>
        <p>Nombre: {{nombre_persona}} {{apellido_persona}}</p>
        <p>Fecha: {{now.day}}</p>
    </body>
</html>
```

Condicionales en las plantillas

(En **plantilla.html** modificando la plantilla creada anteriormente)

```
<html>
  <body>
    <h1>Hola desde la plantilla!</h1>
    <p>Nombre: {{nombre_persona}} {{apellido_persona}}</p>
    <p>Fecha: {{now.day}}</p>
    {% if nombre_persona == "Juan" %}
      <p>Te llamas Juan</p>
    {% else %}
      <p>No te llamas Juan</p>
    {% endif %}
  </body>
</html>
```

Llamar a métodos desde la plantilla

En este caso, no se deben usar los paréntesis, sólo se pone el nombre del método a llamar.

(En **plantilla.html** modificando la plantilla creada anteriormente)

```
<html>
  <body>
    <h1>Hola desde la plantilla!</h1>
    <p>Nombre: {{nombre_persona.upper}} {{apellido_persona.upper}}</p>
    <p>Fecha: {{now}}</p>
  </body>
</html>
```

Cuando Django se encuentra con una instrucción como **nombre.upper** lo primero que comprueba es que se trate de un diccionario, sino, de un atributo, después de un método y por último un índice de lista.

Listas e iteración en la plantilla

(En **views.py** modificando la view saludo creada anteriormente)

```
def saludo(request):
    temas=["Plantillas","Modelos","Formularios","Vistas"]
    arch=open("C:/Users/usuario/Desktop/Proyecto/Proyecto/templates/plantilla.html")
    plt=Template(arch.read())
    arch.close()
    ctx=Context({"temas_curso":temas})
    documento=plt.render(ctx)
    return HttpResponse(documento)
```

Si existen temas y los temas tienen información los muestra en la plantilla.

(En **plantilla.html** modificando la plantilla creada anteriormente)


```

<html>
  <body>
    <ul>
      {% if temas_curso %}
        {% for tema in temas_curso %}
          <li>{{tema}}</li>
        {% endfor %}
      {% else %}
        <p>No hay temas que mostrar</p>
      {% endif %}
    </ul>
  </body>
</html>

```

Filtros

El sistema de plantillas de Django tiene etiquetas y filtros incorporados, que son funciones dentro de la plantilla para representar contenido de una manera específica. Se pueden especificar varios filtros con canalizaciones y los filtros pueden tener argumentos

(En **plantilla.html** modificando la plantilla creada anteriormente)

```

<html>
  <body>
    <ul>
      {% if temas_curso %}
        {% for tema in temas_curso %}
          <li>{{tema|first|lower}}</li>
        {% endfor %}
      {% else %}
        <p>No hay temas que mostrar</p>
      {% endif %}
    </ul>
  </body>
</html>

```

Al aplicar estos dos filtros se mostrará sólo la primera letra de cada tema convertida en minúsculas.

Comentarios en las plantillas

(En **plantilla.html** modificando la plantilla creada anteriormente)

```

<html>
  <body>
    {# Comentario de una línea}
    <p>Nombre: {{nombre_persona.upper}} {{apellido_persona.upper}}</p>
    {% comment %}
      Comentario de
      varias líneas
    {% endcomment %}
  </body>
</html>

```

Cargadores de plantillas (loader)

Se le informa al framework que todas las plantillas están en un directorio en particular. Con esto, se indica sólo el nombre de la plantilla a utilizar, sin usar `open()`, `read()` o `close()`.

- 1) Importar loader en `views.py`

```
from django.template import loader
```

- 2) Para especificar la carpeta donde se encuentran las plantillas ir a **settings.py** y buscar la lista **TEMPLATES**. En concreto a la propiedad **DIRS**, que al estar vacía, hará que el framework busque las plantillas en una carpeta por defecto. Para eso importar **os** y especificar lo siguiente en dicha propiedad **DIRS**. Esto hará que siempre haga referencia a la carpeta del proyecto, sin importar si se mueve de ubicación.

```
#Esto no es todo el archivo, simplemente es una parte

import os

#Aquí habrán otras líneas de código

TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

Método Shortcut render()

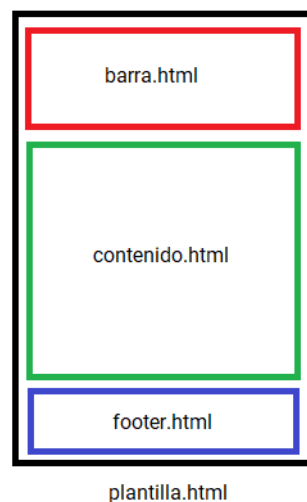
Teniendo el path a las plantillas se puede simplificar aún más el código en las vistas importando el siguiente recurso.

```
from django.http import HttpResponse
import datetime
from django.template import Template, Context
from django.template.loader import get_template
from django.shortcuts import render

def saludo(request):
    nombre = "Juan"
    return render(request, "plantilla.html", {"nombre_persona": nombre})
```

Plantillas incrustadas

Son plantillas dentro de otras, útiles para reutilizar código que se repite entre páginas. Lo primero que hay que hacer es determinar dónde se quiere incrustar las plantillas.



Ejemplo:

Proyecto

```
|--templates
|   |--plantilla.html
|   |--barra.html
|--Proyecto
    |--__init__.py
    |--settings.py
    |--urls.py
    |--views.py
    |--wsgi.py
```

(En **plantilla.html** modificando la plantilla creada anteriormente)

```
<html>
  <body>
    {% include 'barra.html' %}
    <ul>
      {% if temas_curso %}
        {% for tema in temas_curso %}
          <li>{{tema|first|lower}}</li>
        {% endfor %}
      {% else %}
        <p>No hay temas que mostrar</p>
      {% endif %}
    </ul>
  </body>
</html>
```

Es frecuente organizar las plantillas en sub-carpetas, como muestra el siguiente ejemplo. Hay que incluir el path completo para que funcione.

Proyecto

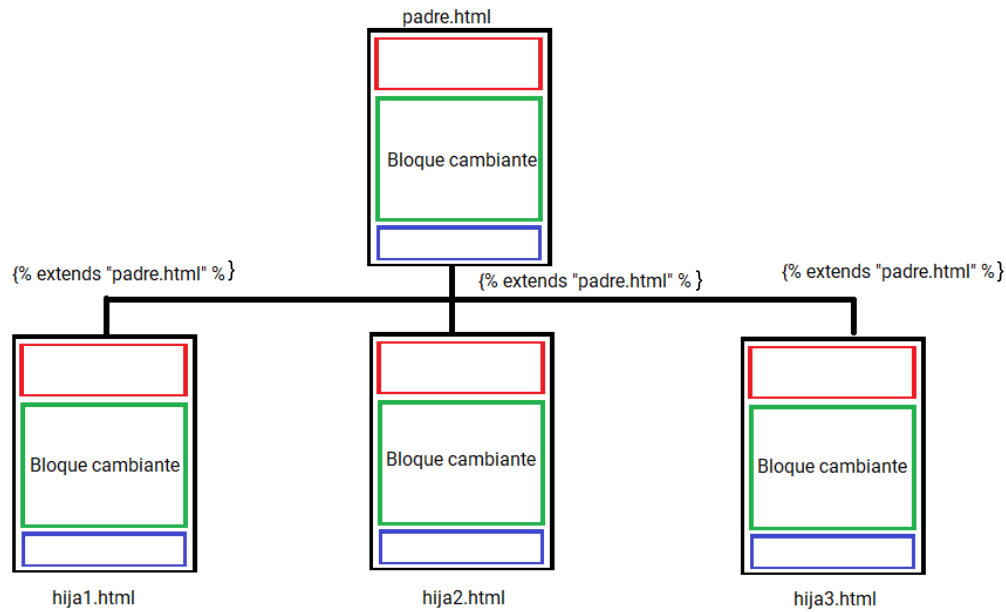
```
--templates
|      |--plantilla.html
|      |--superior
|      |--barra.html
|--Proyecto
|      |--__init__.py
|      |--settings.py
|      |--urls.py
|      |--views.py
|      |--wsgi.py
```

(En **plantilla.html** modificando la plantilla creada anteriormente)

```
<html>
  <body>
    {% include 'superior/barra.html' %}
    <ul>
      {% if temas_curso %}
        {% for tema in temas_curso %}
          <li>{{tema|first|lower}}</li>
        {% endfor %}
      {% else %}
        <p>No hay temas que mostrar</p>
      {% endif %}
    </ul>
  </body>
</html>
```

Herencia de plantillas

Surge en los casos que se repite el formato de las páginas, pero cambia su contenido. Consiste en crear una plantilla en base a la cual se van a basar las demás plantillas del sitio. Al cambiar contenido en la plantilla padre, cambia automáticamente en todas las que hereden.



Ejemplo:

Con la etiqueta dentro del `<title>` se le indica a Django que dicho contenido va a cambiar en cada plantilla.

Proyecto

```
--templates
|   |--plantilla.html
|   |--base.html
|   |--curso.html
|   |--superior
|   |--barra.html
--Proyecto
|   |--__init__.py
|   |--settings.py
|   |--urls.py
|   |--views.py
|   |--wsgi.py
```

(En **base.html**)

```
<html>
  <head>
    <title>{% block title %} {% endblock %} </title>
  </head>
  <body>
    {% block content %}
    {% endblock %}
  </body>
</html>
```

(En **curso.html**)

```
{% extends "base.html" %}
{% block title %} Curso {% endblock %}
{% block content %}
  <p>Fecha: {{now}}</p>
{% endblock %}
```

(En **views.py**)

```
from django.http import HttpResponse
import datetime
from django.template import Template, Context
from django.template.loader import get_template
from django.shortcuts import render

def curso(request):
    fecha = datetime.datetime.now()
    return render(request, "curso.html", {"now": fecha})
```

(En **urls.py**)

```
from django.contrib import admin
from django.urls import path
from Proyecto.views import curso
urlpatterns=[
    path("curso/", curso)
]
```

Django conectado a una base de datos

Usar el motor PostgreSQL

Descargar **PostgreSQL** desde su web oficial, con el instalador presente en postgresql.org/download/windows/

Quick Links

- Downloads
- Binary
- Source
- Software Catalogue
- File Browser

Windows installers

Interactive installer by EnterpriseDB

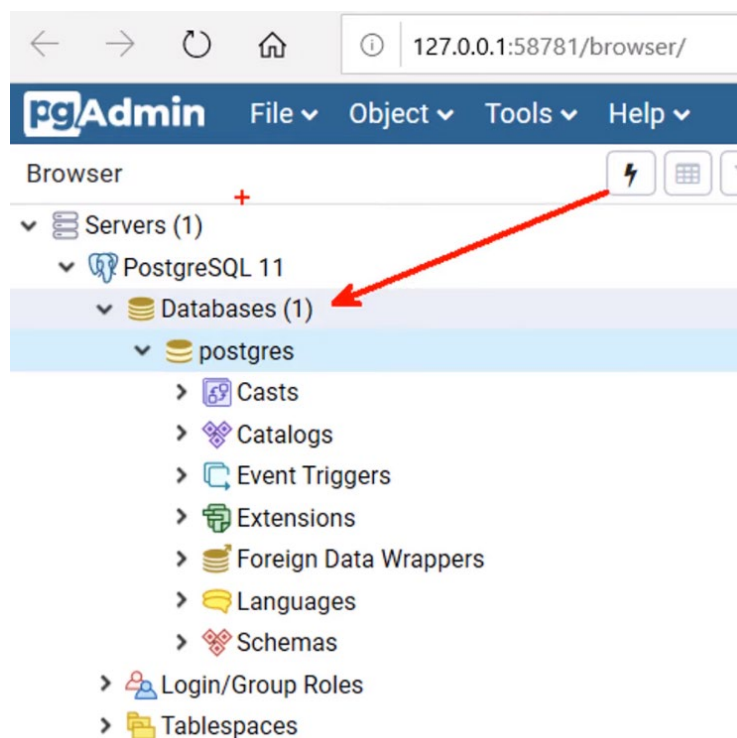
Download the **installer** certified by EnterpriseDB for all supported PostgreSQL versions.

This installer includes the PostgreSQL server, pgAdmin; a graphical tool for managing and developing your databases, and StackBuilder; a package manager that can be used to download and install additional PostgreSQL tools and drivers. Stackbuilder includes management, integration, migration, replication, geospatial, connectors and other tools.

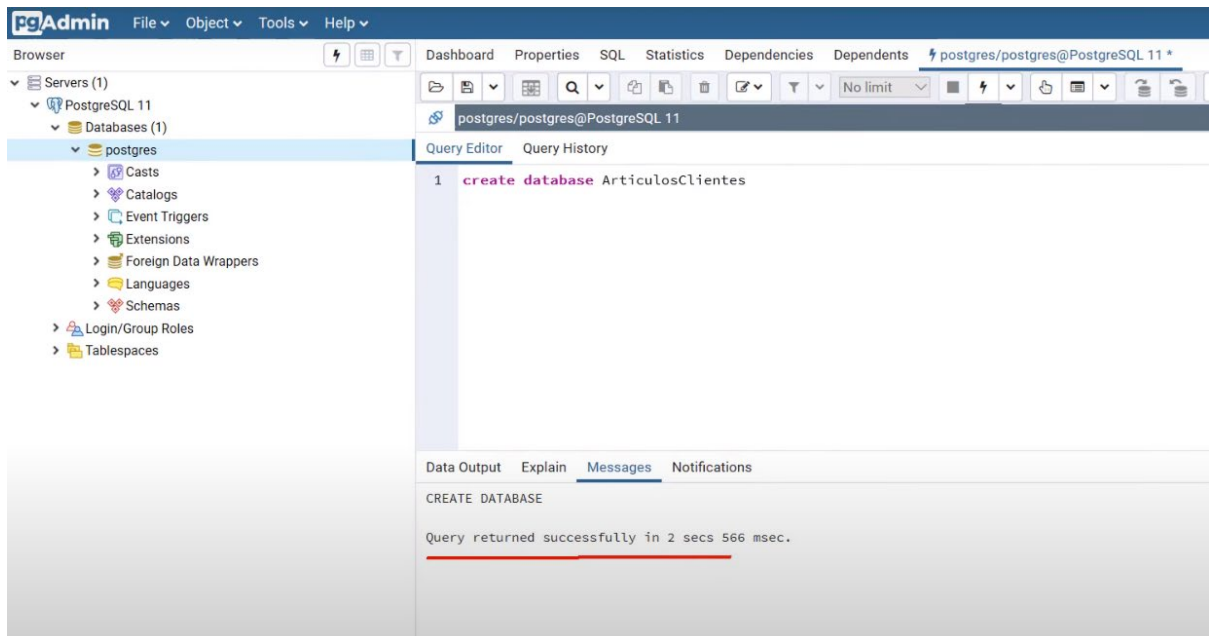
Dejar el puerto en blanco durante la instalación y definir una contraseña de acceso.

Buscar la aplicación **pgAdmin** con el explorador y ejecutarla.

Acceder al **PGAdmin** mediante el navegador.



Botón derecho sobre la base de datos -> **Create Database**. Posteriormente hacer un refresh.



Instalar la librería **psycopg2** para conectar el proyecto con la base de datos de **PostgreSQL**, con un terminal en el directorio del proyecto:

```
pip install psycopg2
```

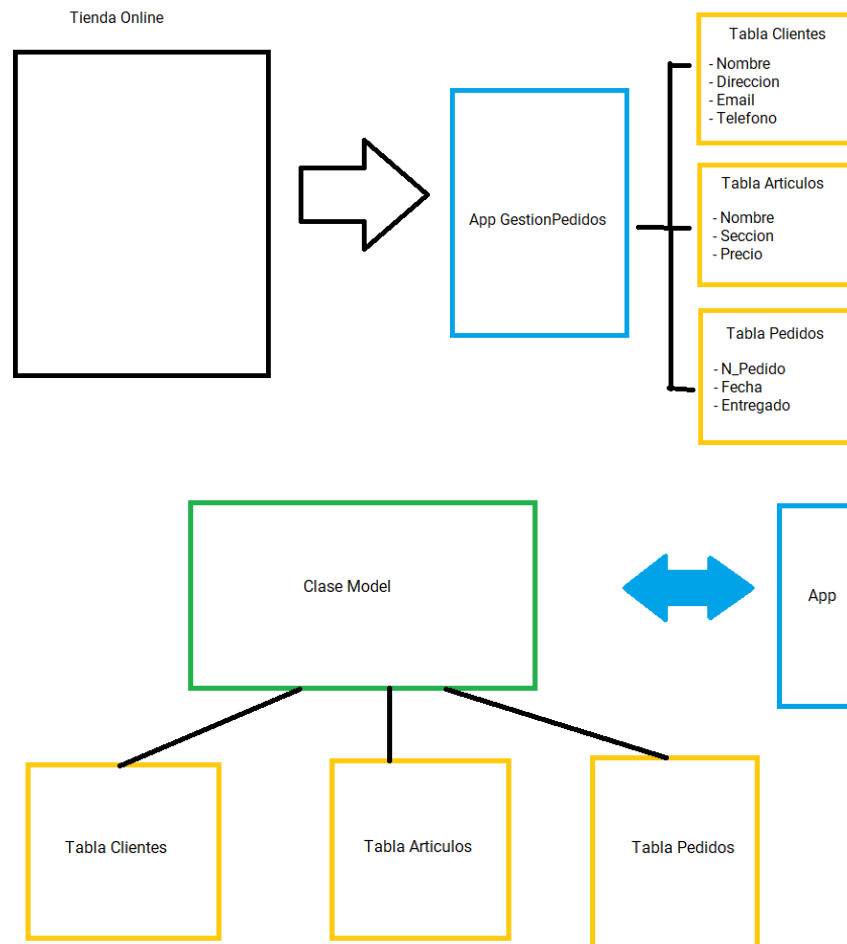
Agregar los datos de la base de datos en **settings.py** justo despues de la propiedad **WSGI_APPLICATION**

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'nombrebasededatos',
        'USER': 'postgres',
        'PASSWORD': 'contraseña_user',
        'HOST': '127.0.0.1',
        'PORT': '5432',
    }
}
```

Realizar migraciones

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

Ejemplo Práctico: Django conectado a una base de datos PostgreSQL.



Con la clase Model se harán todas las consultas a la base de datos. El modelo debe estar si o si dentro de una aplicación para poder trabajar.

Para crear la app ejecutamos el siguiente comando:

```
>>python manage.py startapp gestionPedidos
```

Dentro de la app encontraremos los siguientes archivos:

- **__init__.py** – Gracias a este archivo Python identifica la carpeta como un package válido.
- **admin.py** – Modifica el panel de administración.
- **models.py** – Aquí serán guardados los modelos de las aplicaciones.
- **tests.py** – Aquí estarán los unit tests.
- **views.py** – Aquí estarán las vistas.

Para crear la base de datos se trabajará con el archivo **models.py**. Dentro de dicho archivo se creará una clase por cada tabla que necesitemos en la base de datos. Esto es útil porque

posteriormente se ejecutará el código necesario para el motor de bases de datos que elijamos, por lo tanto nos abstrae de la sintaxis específica de cada motor.

(En **models.py** dentro de la carpeta de la app gestionPedidos)

```
from django.db import models

class Clientes(models.Model):
    nombre=models.CharField(max_length=30)
    direccion=models.CharField(max_length=50)
    email=models.EmailField()
    telefono=models.CharField(max_length=7)

class Articulos(models.Model):
    nombre=models.CharField(max_length=30)
    seccion=models.CharField(max_length=20)
    precio=models.IntegerField()

class Pedidos(models.Model):
    numero=models.IntegerField()
    fecha=models.DateField()
    entregado=models.BooleanField()
```

Posteriormente hay que registrar la aplicación creada en el proyecto de Django, porque en primera instancia no es reconocida por el mismo. Dicho registro se hará en **settings.py**.

(En **settings.py**)

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

Con el comando **python manage.py check *nombre de la app*** chequeamos si fue correctamente incluida al proyecto la nueva aplicación.

```
>>python manage.py check gestionPedidos
```

Para crear el archivo de la base de datos es necesario introducir el siguiente comando:

```
>>python manage.py makemigrations
```

Si todo va bien, veremos el siguiente mensaje.

```
Migrations for 'gestionPedidos':
  gestionPedidos\migrations\0001_initial.py
```

```
-Create model Articulos  
-Create model Clientes  
-Create model Pedidos
```

Los siguientes comandos ejecutarán las sentencias SQL para estructurar la base de datos. El dato final es el número de migración que figura al haber ejecutado el comando `makemigrations`.

```
>>python manage.py sqlmigrate gestionPedidos 0001  
>>python manage.py migrate
```

Cada vez que se hace un cambio en el modelo hay que volver a migrar (makemigrations y migrate)

Insertar registros

Con esta primera forma se harán operaciones con registros hasta que sepamos usar los formularios.

Lo primero que se debe hacer es abrir el **shell**. El shell se cierra con **exit()**

```
>>python manage.py shell
```

De ahí, se debe importar el modelo creado anteriormente, escribiendo en el terminal:

```
>>from gestionPedidos.models import Articulos
```

Y el registro se ingresa de forma par llave-valor, de esta forma Django creará la consulta, pero después hay que ejecutarla.

```
>>art=Articulos(nombre='mesa',seccion='decoracion',precio=90)  
>>art.save()
```

Hay otra forma de hacerlo con una sola instrucción:

```
>>art3=Articulos.objects.create(nombre='taladro',seccion='ferreteria',precio=65)
```

Actualizar registros

Habiendo anteriormente creado el artículo que se guardó en la variable **art3**, solo modificando una propiedad de dicha variable y aplicando **save()** se modificará el registro.

```
>>art3.precio=95  
>>art3.save()
```

Borrar registros

```
>>art5=Articulos.objects.get(id=6)
>>art5.delete()
```

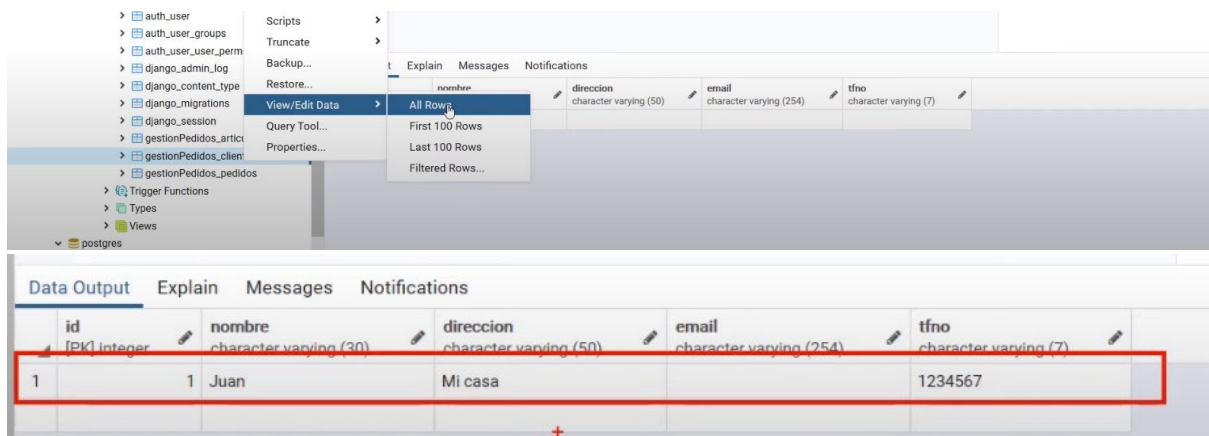
Mostrar la query guardada en las variables

Almacenar la query en una variable y posteriormente mostrarla:

```
>>Lista=Articulos.objects.all()
>>Lista.query.__str__()

#Resultado
>>'SELECT "gestionPedidos_articulos"."id", "gestionPedidos_articulos"."nombre"...'
```

Introducir registro



Select con Where

```
>>python manage.py shell
>>from gestionPedidos.models import Articulos
>>Articulos.objects.filter(seccion='deportes')
```

Posteriormente hay que transformar la respuesta de la anterior línea ejecutada a **String**, se realiza con un método `__str__()` dentro de la class Articulos en **models.py**

```
class Articulos(models.Model):
```

```

nombre=models.CharField(max_length=30)
seccion=models.CharField(max_length=20)
precio=models.IntegerField()
def __str__(self):
    return 'El nombre es %s la sección %s y el precio
%s'%(self.nombre,self.seccion,self.precio)

```

Where con and

```
>>Articulos.objects.filter(nombre='mesa',seccion='decoracion')
```

Where con mayor/menor

```
>>Articulos.objects.filter(seccion='deportes',precio__gte=100)
```

gte: Greater Than/Equal | **lte:** Lesser Than/Equal

gt: Greater Than | **lt:** Lesser Than

precio__range(10,150) #Precio entre 10 y 150

Order By (ASC)

```
>>Articulos.objects.filter(precio__gte=50).order_by('precio')
```

Order By (DESC)

```
>>Articulos.objects.filter(precio__gte=50).order_by('-precio')
```

Documentación: <https://docs.djangoproject.com/en/3.1/ref/models/queriesets/>

Para salir del **shell** escribir **exit()**

Panel de Administración

Sirve para manejar el contenido del sitio de una forma más sencilla. Mantiene actualizado el sitio dado que permite agregar/quitar usuarios, contenido, etc.

Está habilitado por defecto en todos los proyectos de Django. El archivo **admin.py** sirve para modificar todo lo relativo al panel de administración. La idea de usar el panel radica en no tener que escribir las consultas constantemente para cambiar los modelos del proyecto.



En el archivo **urls.py** ya viene importado el panel de administrador.

(en `urls.py`)

```
from django.contrib import admin
from django.urls import path

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

Se debe crear un superusuario con perfil de administrador.

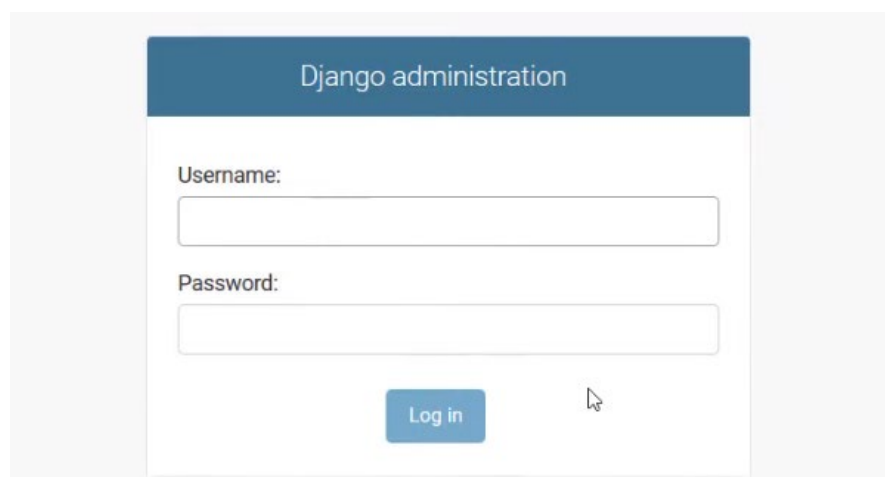
```
>>python manage.py createsuperuser
```

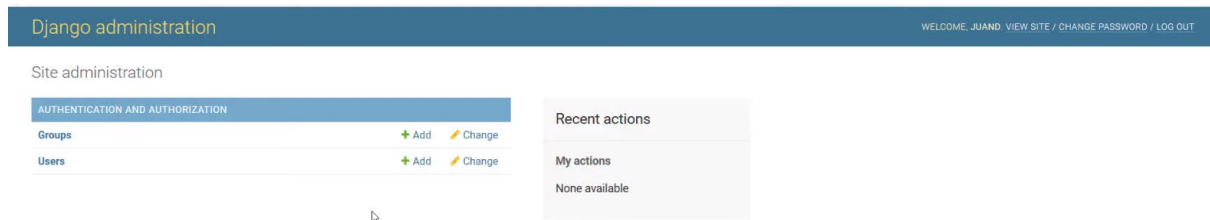
Django nos pedirá crear un nombre de usuario, un email y contraseña para el usuario. Cuando se ingresa la contraseña **no hay un feedback visual** de que insertamos caracteres, pero los está incluyendo.

```
>>Username (leave blank to use 'usuarioejemplo'): superusuario
>>Email address: ejemplo@ejemplo.com
>>Password:
>>Password (again):
```

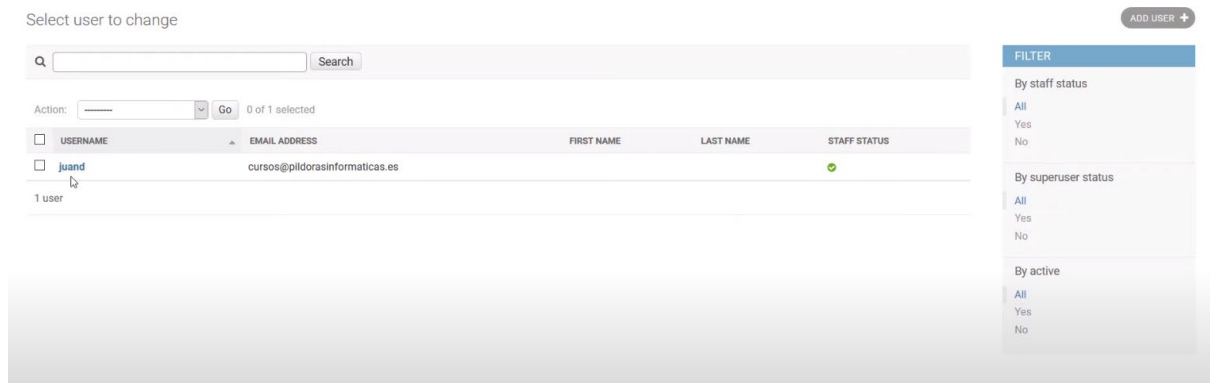
No hay que olvidarse de ejecutar el servidor con **python manage.py runserver** para ver los cambios impactados en el proyecto.

Posteriormente, se debe correr el servidor. Se accede al panel de administrador mediante la siguiente dirección: **127.0.0.1:8000/admin/**

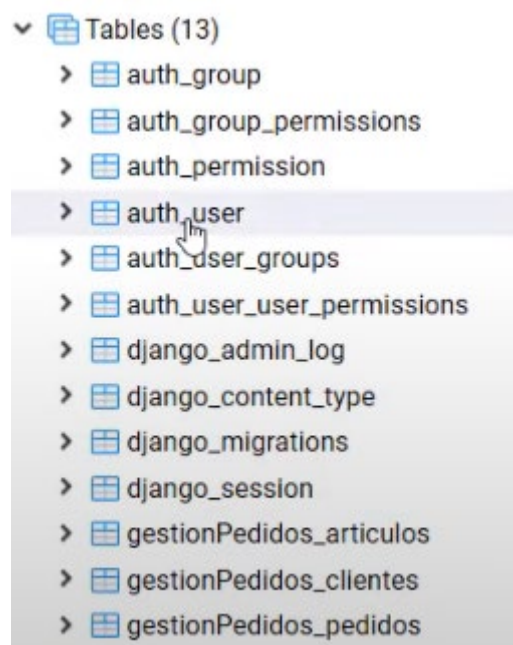
The image shows a web browser window displaying the Django administration login interface. At the top, there is a dark blue header bar with the text "Django administration" in white. Below the header, the page has a light gray background. In the center, there is a white rectangular box containing the login form. The form has two labels: "Username:" and "Password:", each followed by a text input field. Below the input fields, there is a blue button with the text "Log in" in white. A mouse cursor is visible over the "Log in" button.



Por defecto se pueden crear y modificar usuarios y grupos



En la base de datos de PostgreSQL podemos ver la tabla **auth_users**, Se ven los usuarios creados con la contraseña encriptada.



Data Output	Explain	Messages	Notifications
id	password	last_login	is_superuser
1	pbkdf2_sha256\$150000\$nvDj...	2019-11-13 12:35:08.680382+01	true
2	pbkdf2_sha256\$150000\$3l7p...	[null]	false

En **admin.py** se codifica lo necesario para modificar los modelos. Hay que importar los modelos. Django le agrega una **s** al final todas las tablas en el panel de administrador. (En **admin.py**)

```
from django.contrib import admin
from gestionPedidos.models import Clientes, Articulos, Pedidos
```

Site administration

AUTHENTICATION AND AUTHORIZATION

Groups	+ Add	Change
Users	+ Add	Change

GESTIONPEDIDOS

Articulos	+ Add	Change
Clientes	+ Add	Change
Pedidos	+ Add	Change

Recent actions

My actions

+ Clientes object (2)

Clientes

✗ Clientes object (1)

Clientes

Los campos obligatorios aparecen con la fuente en **bold**.

Add clientes

+

Nombre:

Direccion:

Email:

Tfno:

Save and add another

Save and continue editing

SAVE

Si está incluido el método `__str__()` los registros se van a ver de la siguiente manera:

Select artículos to change

Action: 0 of 8 selected

<input type="checkbox"/>	ARTICULOS
<input type="checkbox"/>	El nombre es tren eléctrico la sección es juguetería y el precio es 135
<input type="checkbox"/>	El nombre es muñeca la sección es juguetería y el precio es 15
<input type="checkbox"/>	El nombre es raqueta la sección es deportes y el precio es 105
<input type="checkbox"/>	El nombre es balón la sección es deportes y el precio es 25
<input type="checkbox"/>	El nombre es destornillador la sección es ferretería y el precio es 35
<input type="checkbox"/>	El nombre es pantalón la sección es confección y el precio es 45
<input type="checkbox"/>	El nombre es lámpara la sección es decoración y el precio es 70
<input type="checkbox"/>	El nombre es mesa la sección es decoración y el precio es 90

8 articulos

Para hacer que un campo sea opcional hay que modificar el modelo y posteriormente hacer **makemigrations** y **migrate**. No hay que olvidarse tampoco de **correr el servidor**.

(En models.py)

```
email=models.EmailField(blank=True,null=True)
```

Nombre:

Direccion:

Email:

Tfno:

Cambios de nombres de campos en el panel de administración

Django pone automáticamente en el panel los campos de los modelos con otro formato, lo cual no quiere decir que los cambie en el modelo, simplemente los formatea en la vista.

Quitará los guiones bajos y siempre la primera letra la pondrá en mayúscula.

(En models.py)

```
direccion=models.CharField(max_length=50,verbose_name="La dirección")
```

Nombre:	<input type="text"/>
La dirección:	<input type="text"/>
Email:	<input type="text"/>
Tfno:	<input type="text"/>

Resultado final (en **models.py**)

```
class Clientes(models.Model):
    nombre=models.CharField(max_length=30)
    direccion=models.CharField(max_length=50,verbose_name="La dirección")
    email=models.EmailField(blank=True,null=True)
    telefono=models.CharField(max_length=7)

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

Select clientes to change

Action: Go 0 of 2 selected

<input type="checkbox"/>	CLIENTES
<input type="checkbox"/>	Elena
<input type="checkbox"/>	María

2 clientess

Si queremos mostrar sólo algunos campos de los modelos desde el panel de administración tenemos que trabajar con clases que hereden de ModelAdmin. Hay que detener y volver a arrancar el servidor.

(En **admin.py**)

```
from django.contrib import admin
from gestionPedidos.models import Clientes,Articulos,Pedidos

class ClientesAdmin(admin.ModelAdmin):
    list_display=("nombre","direccion","telefono",)

admin.site.register(Clientes,ClientesAdmin)
```

Select clientes to change

Action: Go 0 of 2 selected

<input type="checkbox"/>	NOMBRE	LA DIRECCIÓN	TFNO
<input type="checkbox"/>	Elena	P vergara	654654
<input type="checkbox"/>	María	Gran Vía	123456

2 clientess

Campo de búsqueda en el Panel

Se utilizará **search_fields()**, a dicho método hay que indicarle por qué campos necesitamos hacer las búsquedas.

(En **admin.py**)

```
from django.contrib import admin
from gestionPedidos.models import Clientes, Articulos, Pedidos

class ClientesAdmin(admin.ModelAdmin):
    list_display=("nombre", "direccion", "telefono",)
    search_fields=("nombre", "telefono",)

admin.site.register(Clientes, ClientesAdmin)
```

Select clientes to change



The screenshot shows the Django Admin interface for the 'Clientes' model. At the top, there is a search bar with a magnifying glass icon and a 'Search' button. Below the search bar, there is an 'Action:' dropdown menu and a 'Go' button. The main part of the interface is a table with three columns: 'NOMBRE', 'LA DIRECCIÓN', and 'TELNO'. The table contains two rows of data: 'Elena' with address 'P vergara' and phone number '654654', and 'Maria' with address 'Gran Via' and phone number '123456'. At the bottom of the table, it says '2 clientess'.

<input type="checkbox"/>	NOMBRE	LA DIRECCIÓN	TELNO
<input type="checkbox"/>	Elena	P vergara	654654
<input type="checkbox"/>	Maria	Gran Via	123456

2 clientess

Filtros

Crear una nueva clase que herede de **ModelAdmin**. Con **list_filter** especificamos el campo a filtrar. Los campos deben ser escritos de la misma forma que están en la base de datos.

(En **admin.py**)

```
from django.contrib import admin
from gestionPedidos.models import Clientes, Articulos, Pedidos

class ClientesAdmin(admin.ModelAdmin):
    list_display=("nombre", "direccion", "telefono",)
    search_fields=("nombre", "telefono",)

class ArticulosAdmin(admin.ModelAdmin):
    list_filter=("seccion",)

admin.site.register(Clientes, ClientesAdmin)
admin.site.register(Articulos, ArticulosAdmin)
```

Select artículos to change

Action: Go 0 of 8 selected

ARTICULOS
<input type="checkbox"/> El nombre es tren eléctrico la sección es juguetería y el precio es 250
<input type="checkbox"/> El nombre es muñeca la sección es juguetería y el precio es 15
<input type="checkbox"/> El nombre es raqueta la sección es deportes y el precio es 105
<input type="checkbox"/> El nombre es balón la sección es deportes y el precio es 25
<input type="checkbox"/> El nombre es destornillador la sección es ferretería y el precio es 35
<input type="checkbox"/> El nombre es pantalón la sección es confección y el precio es 45
<input type="checkbox"/> El nombre es lámpara la sección es decoración y el precio es 70
<input type="checkbox"/> El nombre es mesa la sección es decoración y el precio es 90

8 artículos

ADD ARTICULOS +

FILTER

By section

All +

confección

decoración

deportes

ferretería

juguetería

Filtrar por fecha

Como el panel está en Inglés, las fechas deben ser agregadas en ese formato:

Add pedidos

Numero:

Fecha: Today |

Note: You are 1 hour ahead of server time.

☐ Entregado

(En admin.py)

```
from django.contrib import admin
from gestionPedidos.models import Clientes,Articulos,Pedidos

class ClientesAdmin(admin.ModelAdmin):
    list_display=("nombre","direccion","telefono",)
    search_fields=("nombre","telefono",)

class ArticulosAdmin(admin.ModelAdmin):
    list_filter=("seccion",)

class PedidosAdmin(admin.ModelAdmin):
    list_display=("numero","fecha",)
    list_filter=("fecha",)

admin.site.register(Clientes,ClientesAdmin)
```

```
admin.site.register(Articulos,ArticulosAdmin)
admin.site.register(Pedidos,PedidosAdmin)
```

Select pedidos to change

Action: Go 0 of 4 selected

<input type="checkbox"/>	NUMERO	FECHA
<input type="checkbox"/>	1008	Sept. 15, 2019
<input type="checkbox"/>	1007	Nov. 19, 2019
<input type="checkbox"/>	1006	Nov. 15, 2019
<input type="checkbox"/>	1005	Dec. 30, 2019

4 pedidos

ADD PEDIDOS +

FILTER

By fecha

Any date

Today

Past 7 days

This month

This year

Agregar Breadcrumbs

(En admin.py)

```
from django.contrib import admin
from gestionPedidos.models import Clientes,Articulos,Pedidos

class ClientesAdmin(admin.ModelAdmin):
    list_display=("nombre","direccion","telefono",)
    search_fields=("nombre","telefono",)
class ArticulosAdmin(admin.ModelAdmin):
    list_filter=("seccion",)

class PedidosAdmin(admin.ModelAdmin):
    list_display=("numero","fecha",)
    list_filter=("fecha",)
    date_hierarchy="fecha"

admin.site.register(Clientes,ClientesAdmin)
admin.site.register(Articulos,ArticulosAdmin)
admin.site.register(Pedidos,PedidosAdmin)
```

Select pedidos to change

All dates September 2019 November 2019 December 2019

Action: Go 0 of 4 selected

<input type="checkbox"/>	NUMERO	FECHA
<input type="checkbox"/>	1008	Sept. 15, 2019
<input type="checkbox"/>	1007	Nov. 19, 2019
<input type="checkbox"/>	1006	Nov. 15, 2019
<input type="checkbox"/>	1005	Dec. 30, 2019

4 pedidos

ADD PEDIDOS +

FILTER

By fecha

Any date

Today

Past 7 days

This month

This year

Cambiar idioma del Panel

Ir a **settings.py** y ubicar la propiedad **LANGUAGE_CODE**. Cambiar su valor por 'es-eu'

(en **settings.py**)

LANGUAGE_CODE = 'es-es'

Usuarios

Al primer usuario del panel **Django** lo asigna como **Superusuario**. El campo **Es Staff** permite que el usuario entre al Panel.

Escoja usuario a modificar

Acción: seleccionados 0 de 1

<input type="checkbox"/>	NOMBRE DE USUARIO	DIRECCIÓN DE CORREO ELECTRÓNICO	NOMBRE	APELLIDOS	ES STAFF
<input type="checkbox"/>	juand	cursos@pildorasinformaticas.es			<input checked="" type="checkbox"/>

1 usuario

AÑADIR USUARIO +

FILTRO

Por es staff

Todo

Sí

No

Por es superusuario

Todo

Sí

Añadir usuario

Primero introduzca un nombre de usuario y una contraseña. Luego podrá editar el resto de opciones del usuario.

Nombre de usuario:

sonia

Requerido. 150 caracteres como máximo. Únicamente letras, dígitos y @/./+/-/_

Contraseña:

Su contraseña no puede asemejarse tanto a su otra información personal.
Su contraseña debe contener al menos 8 caracteres.
Su contraseña no puede ser una clave utilizada comunmente.
Su contraseña no puede ser completamente numérica.

Contraseña (confirmación):

Para verificar, introduzca la misma contraseña anterior.

Apellidos:

Dirección de correo electrónico:

Permisos

☒ **Activo**
Indica si el usuario debe ser tratado como activo. Desmarque esta opción en lugar de borrar la cuenta.

☐ **Es staff**
Indica si el usuario puede entrar en este sitio de administración.

☐ **Es superusuario**
Indica que este usuario tiene todos los permisos sin asignárselos explícitamente.

Grupos:

grupos Disponibles

administradores standard

grupos elegidos

Selecciona todos Eliminar todos

Los pertenecientes a un grupo tienen una serie de permisos asociados, aunque también se pueden definir más permisos.

Permisos de usuario:

permisos de usuario Disponibles

admin | entrada de registro | Can add log entry
admin | entrada de registro | Can change log entry
admin | entrada de registro | Can delete log entry
admin | entrada de registro | Can view log entry
auth | grupo | Can add group
auth | grupo | Can change group
auth | grupo | Can delete group
auth | grupo | Can view group
auth | permiso | Can add permission
auth | permiso | Can change permission
auth | permiso | Can delete permission
auth | permiso | Can view permission
auth | usuario | Can add user
auth | usuario | Can change user
auth | usuario | Can delete user

permisos de usuario elegidos

Selecciona todos Eliminar todos

Creación de Grupo

Inicio › Autenticación y autorización › Grupos

Escoja grupo a modificar

Acción:

 seleccionados 0 de 1

☐ GRUPO

☐ administradores standard

1 grupo

Añadir grupo

Nombre:

Permisos:

permisos Disponibles ?

contenttypes | tipo de contenido | Can add content type

contenttypes | tipo de contenido | Can change content type

contenttypes | tipo de contenido | Can delete content type

contenttypes | tipo de contenido | Can view content type

gestionPedidos | articulos | Can add articulos

gestionPedidos | articulos | Can change articulos

gestionPedidos | articulos | Can delete articulos

gestionPedidos | articulos | Can view articulos

gestionPedidos | clientes | Can add clientes

gestionPedidos | clientes | Can change clientes

gestionPedidos | clientes | Can delete clientes

gestionPedidos | clientes | Can view clientes

gestionPedidos | pedidos | Can add pedidos

gestionPedidos | pedidos | Can change pedidos

gestionPedidos | pedidos | Can delete pedidos

Selecciona todos ?

permisos elegidos ?

Mantenga presionado "Control", o "Command" en un Mac, para seleccionar más de una opción.

Formularios

Crear formulario **html** en carpeta Templates

(Dentro de la carpeta templates, archivo **busqueda_productos.html**)

```
<html>
  <head>
    <title>Búsqueda de productos</title>
  </head>
  <body>
    <form action="/buscar/" method="GET">
      <input type="text" name="prd">
      <input type="submit" value="Buscar">
    </form>
  </body>
</html>
```

(Dentro de la carpeta templates, archivo **resultados_busqueda.html**)

```
<html>
  <head>
    <title>Resultado de la búsqueda</title>
  </head>
  <body>
    <p>Estás buscando: <strong>{{query}}</strong></p>
    {% if articulos %}
      <p>Artículos encontrados: {{articulos|length}} articulos</p>
      <ul>
        {% for articulo in articulos %}
          <li>{{articulo.nombre}} &nbsp; {{articulo.seccion}} &nbsp;
            {{articulo.precio}}</li>
        {% endfor %}
      </ul>
    {% else %}
      <p>No está el artículo buscado</p>
    {% endif %}
  </body>
</html>
```

icontains funciona como el **like** en **SQL**: `SELECT * FROM Articulos LIKE nombre=" "`
Todo aquel registro que tenga la palabra en algún lugar que se ingresó en el formulario se mostrará.

(En **views.py**)

```
from django.shortcuts import render
from django.http import HttpResponse
from gestionPedidos.models import Articulos

def busqueda_productos(request):
    return render(request, "busqueda_productos.html")

def buscar(request):
    if request.GET["prd"]:
        producto=request.GET["prd"]
        articulos=articulos.objects.filter(nombre__icontains=producto)
        return render(request, "resultados_busqueda.html", {"articulos":articulos, "query":producto})
    else:
        mensaje="No has introducido ningún dato"
        return HttpResponse(mensaje)
```

(En **urls.py**)

```
from django.contrib import admin
from django.urls import path
from gestionPedidos import views

urlpatterns=[
    path('admin/', admin.site.urls),
    path('busqueda_productos/', views.busqueda_productos),
    path('buscar/', views.buscar),
]
```

Con la información introducida en el formulario, que llegó al servidor con el **GET** del objeto **request**, podemos hacer una consulta a la base de datos.

Data Output	Explain	Messages	Notifications
	id [PK] integer	nombre character varying (30)	seccion character varying (20) precio integer
1	1	mesa	decoración 90
2	2	lámpara	decoración 70
3	3	pantalón	confección 45
4	4	destornillador	ferretería 35
5	5	balón	deportes 25
6	6	raqueta	deportes 105
7	7	muñeca	juguetería 15
8	8	tren eléctrico	juguetería 250

Limitar caracteres de búsqueda en consulta a Base de Datos

(En **views.py**)

```
from django.shortcuts import render
from django.http import HttpResponse
from gestionPedidos.models import Articulos

def busqueda_productos(request):
    return render(request, "busqueda_productos.html")

def buscar(request):
    if request.GET["prd"]:
        producto=request.GET["prd"]
        if len(producto)>20:
            mensaje="Texto de búsqueda demasiado largo"
        else:
            articulos=Articulos.objects.filter(nombre__icontains=producto)
        return render(request, "resultados_busqueda.html", {"articulos": articulos, "query": producto})
    else:
        mensaje="No has introducido ningún dato"
        return HttpResponse(mensaje)
```

Fix class has no objects member

Este error no impide el correcto funcionamiento del proyecto, pero si queremos desactivarlo necesitamos instalar pylint-django y configurarlo

```
>>pip install pylint-django
```

Abrir el Command Palette con Ctrl+Shift+P. Escribir en el buscador Preferences: Configure Language Specific Settings. Seleccionar Python.

Pegar el siguiente código dentro de las primeras llaves.

```
"python.linting.pylintArgs": [ "--load-plugins=pylint_django", ]
```

Forms API

Hay que crear un archivo llamado **forms.py** que tendrá una clase por formulario existente en el proyecto. Se crea en el mismo directorio que el archivo **views.py**

(En **forms.py**)

```
from django import forms
class FormularioContacto(forms.Form):
    asunto=forms.CharField()
    email=forms.EmailField()
    mensaje=forms.CharField()
```

Se puede probar con el shell que hay en la instancia del formulario una vez creado. El método `.as_*`() siendo * p, ul, etc, formateara el formulario.

```
>>>python manage.py shell
>>>from gestionPedidos.forms import FormularioContacto
>>>miFormulario=FormularioContacto()
>>>print(miFormulario)
>><tr><th><label for="id_asunto">Asunto:</label></th><td><input type="text" name="asunto"
required id="id_asunto"></td></tr>
<tr><th><label for="id_email">Email:</label></th><td><input type="email" name="email"
required id="id_email"></td></tr>
<tr><th><label for="id_mensaje">Mensaje:</label></th><td><input type="text"
name="mensaje" required id="id_mensaje"></td></tr>
```

```
>>>print(miFormulario.as_p())
>><p><label for="id_asunto">Asunto:</label><input type="text" name="asunto" required
id="id_asunto"></p>
<p><label for="id_email">Email:</label><input type="email" name="email" required
id="id_email"></p>
<p><label for="id_mensaje">Mensaje:</label><input type="text" name="mensaje" required
id="id_mensaje"></p>
```

```
>>>print(miFormulario.as_ul())
>><li><label for="id_asunto">Asunto:</label><input type="text" name="asunto" required
id="id_asunto"></li>
<li><label for="id_email">Email:</label><input type="email" name="email" required
id="id_email"></li>
<li><label for="id_mensaje">Mensaje:</label><input type="text" name="mensaje" required
id="id_mensaje"></li>
```

`is_valid()` es un método de **API Forms** que si devuelve **True** (cuando los datos ingresados en él son válidos) podemos usar la propiedad **cleaned_data** que devuelve un diccionario con la información insertada en el formulario.

En **views.py** importar la clase creada en **forms.py**

```
from gestionPedidos.forms import FormularioContacto
```

Al hacer click el usuario en enviar, la información se almacena en **miFormulario** a través del método **POST**. Si el formulario pasa la validación, se crea una variable que almacenará toda la información insertada en el formulario con **.cleaned_data**.

Después se renderizan en un template los datos de dicho formulario. Cuando el método es **POST** significa que el usuario envía información al servidor.

Al no crear la etiqueta **<form>** cuando se crea el formulario con **API Forms**, es necesario incluir las etiquetas necesarias en el template.

(En **views.py**)

```
def contacto(request):
    if request.method=="POST":
        miFormulario=FormularioContacto(request.POST)
        if miFormulario.is_valid():
            infForm=miFormulario.cleaned_data
            #Realizar operación con los datos aquí e incluir gracias.html
            return render(request,"gracias.html")
        else:
            miFormulario=FormularioContacto()
    return render(request,"formulario_contacto.html",{"form":miFormulario})
```

Agregar a **urls.py**

```
from django.contrib import admin
from django.urls import path
from gestionPedidos import views

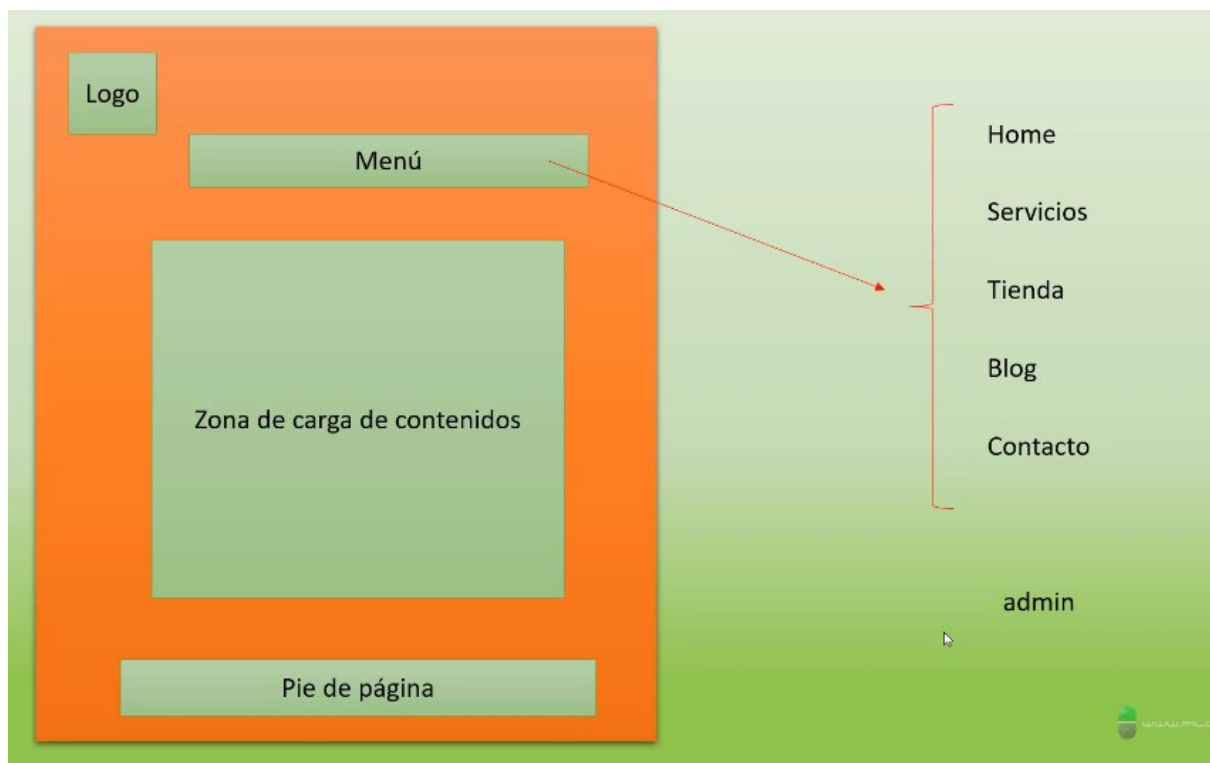
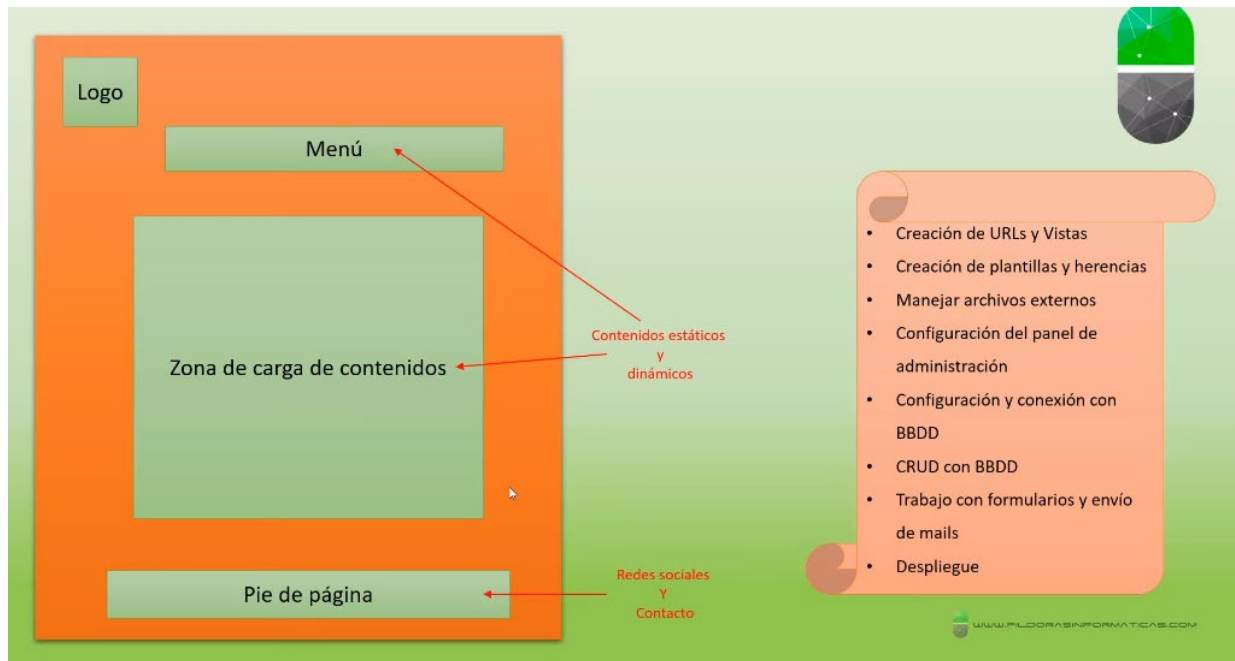
urlpatterns=[
    path('admin/',admin.site.urls),
    path('busqueda_productos/',views.busqueda_productos),
    path('buscar/',views.buscar),
    path('contacto/',views.contacto),
]
```

(Dentro de la carpeta templates, en **formulario_contacto.html**)

```
<html>
  <head>
    <title>Formulario de Contacto</title>
  </head>
  <body>
    <h1>Contáctanos</h1>
    {% if form.errors %}
      <p style="color:red;">Por favor revisa este campo</p>
    {% endif %}
    <form action="" method="POST">{% csrf_token %}
      <table>
        {{form.as_table}}
      </table>
      <input type="submit" value="Enviar">
    </form>
  </body>
</html>
```

Ver documentación sobre **CSRF** en: <https://docs.djangoproject.com/en/3.1/ref/csrf/>

Ejemplo de Proyecto



Crear proyecto y app

```
>>django-admin startproject ProyectoWeb
>>cd ProyectoWeb
>>python manage.py startapp ProyectoWebApp
```

Se puede probar en este momento si está bien armada la aplicación

```
>>python manage.py runserver
```

Podemos comenzar por las vistas, hay que tener tantas vistas como páginas tenga el sitio.

(En **views.py**)

```
from django.http import HttpResponseRedirect

def home(request):
    return HttpResponseRedirect("Home")

def servicios(request):
    return HttpResponseRedirect("Servicios")

def blog(request):
    return HttpResponseRedirect("Blog")

def tienda(request):
    return HttpResponseRedirect("Tienda")

def contacto(request):
    return HttpResponseRedirect("Contacto")
```

Para tener mejor organizadas las urls es conveniente crear un archivo de urls que va a ser válido para solo una aplicación. Dentro de **ProyectoWebApp** crear un archivo **urls.py**.

(En **urls.py** dentro de la carpeta de la app)

```
from django.contrib import admin
from django.urls import path
from ProyectoWebApp import views

urlpatterns=[
    path('admin/',admin.site.urls),
    path('',views.home, name="Home"),
    path('servicios',views.servicios,name="Servicios"),
    path('tienda',views.tienda,name="Tienda"),
    path('contacto',views.contacto,name="Contacto"),
    path('blog',views.blog,name="Blog"),
]
```

Desde el **urls.py** del proyecto se enlaza al **urls.py** de la app.

(En **urls.py** dentro de la carpeta del proyecto)

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

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

Reiniciar el servidor para hacer pruebas.

Crear una carpeta dentro de la carpeta del proyecto llamada **templates**, y dentro de ella una carpeta llamada como la app “**ProyectoWebApp**”

```
ProyectoWeb
|--templates
    |--ProyectoWebApp
```

Modificar **settings.py** dentro de la carpeta del proyecto agregando la app, **no olvidar la coma al final**.

(En **settings.py**)

```
...
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    ],
]

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

Modificar en **views.py** las vistas necesarias para el proyecto.

(En **views.py** en la carpeta del proyecto)

```
from django.shortcuts import render,HttpResponse
def home(request):
    return render(request,"home.html")

def servicios(request):
    return render(request,"servicios.html")

def blog(request):
    return render(request,"blog.html")

def tienda(request):
    return render(request,"tienda.html")

def contacto(request):
    return render(request,"contacto.html")
```

Archivos Estáticos

Aplicar formato con **Bootstrap** y cargar archivos estáticos.

Ir a la carpeta **ProyectoWebApp** y crear una carpeta para contener a los archivos estáticos. Será llamada **static**. Dentro de **static** se crea otra carpeta con el nombre de la app "**ProyectoWebApp**".

Descargar zip del video de **Curso Django - Píldoras Informáticas - Clase 29**. Copiar el contenido de ese zip dentro de la carpeta creada anteriormente.

```
ProyectoWebApp
|--static-----|
      ProyectoWebApp
            |--css
            |--img
            |--vendor
            |--home.html
```

El archivo **home.html** sustituirá al home que habíamos creado como template en un principio.

Dentro de él estarán los enlaces a las imágenes, a los archivos requeridos para Bootstrap y demás.

Para cargar los contenidos de la carpeta **static** (al principio del head):

```
<head>
    {% load static %}
    <link href="{% static 'ProyectoWebApp/vendor/bootstrap/css/bootstrap.min.css' %}"
rel="stylesheet">
    <link href="{% static 'ProyectoWebApp/css/gestion.css' %}" rel="stylesheet">
</head>
```

Ejemplo de carga de imágenes dentro de la carpeta static (en el <body>)

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

Ejemplo de carga de scripts dentro de la carpeta static (en el <body>)

```
<body>
    <script src="{% static 'ProyectoWebApp/vendor/jquery/jquery.min.js' %}"></script>
</body>
```

El header y el footer deberían ser **comunes** a todas las páginas, por lo tanto se creará la **herencia de plantillas**.

Hacemos una copia de **home.html** y la renombramos **base.html**. En la plantilla **base.html** se quitan los contenidos que cambiarán entre las páginas que son los siguientes:

```
<!-- Heading -->
<section class="page-section clearfix"></section>
<!-- Message -->
<section class="page-section cta"></section>
```

Y se agregan en su lugar los bloques que indican que dicha parte del HTML cambiará en las plantillas que hereden de **base.html**.

```
<!-- Contenido cambiante -->
{% block content %}
{% endblock %}
```

Hay que modificar todas las urls de **base.html** para que no haya problemas al acceder a las demás páginas. Hay que hacer coincidir el nombre después de cada url con lo que pusimos en **urls.py**. Replicar para todos los enlaces.

```
<body>
    <ul class="navbar-nav mx-auto">
        <li class="nav-item active px-lg-4">
            <a class="nav-link text-uppercase text-expanded" href="{%url
'Home'%}">Inicio</a>
        </li>
    </ul>
</body>
```

En **home.html** hay que borrar todo lo que es de **base.html**. Básicamente se borra todo menos el **section heading** y el **section message**.

El **block content** se abre cuando empiezan los contenidos propios de **home.html** y se cierra cuando terminan.

```
{% extends "ProyectoWebApp/base.html" %}
{% load static %}
{% block content %}
<!-- Heading -->
<section class="page-section clearfix"></section>
<!-- Message -->
<section class="page-section cta"></section>
{% endblock %}
```

Si no se ven los cambios hechos en css **borrar caché** y **reiniciar servidor**.

(En **servicios.html**)

```
{% extends "ProyectoWebApp/base.html" %}
{% load static %}
{% block content %}
<!-- Heading -->
<section class="page-section clearfix"></section>
<!-- Message -->
<section class="page-section cta"></section>
{% endblock %}
```

Para resaltar la opción en el menú dependiendo de en qué página se está:

(en **base.html**)

```
<body>
  <ul class="navbar-nav mx-auto">
    <li class="nav-item px-lg-4">
      <a class="nav-link text-uppercase text-expanded" href="{%url
'Home'%}">Inicio</a>
    </li>
    <li class="nav-item {% if 'servicios' in request.path %}active{% endif %} px-lg-4">
      <a class="nav-link text-uppercase text-expanded" href="{%url
'Servicios'%}">Servicios</a>
    </li>
    <li class="nav-item {% if 'tienda' in request.path %}active{% endif %} px-lg-4">
      <a class="nav-link text-uppercase text-expanded" href="{%url
'Tienda'%}">Tienda</a>
    </li>
    <li class="nav-item {% if 'contacto' in request.path %}active{% endif %} px-lg-4">
      <a class="nav-link text-uppercase text-expanded" href="{%url
'Contacto'%}">Contacto</a>
    </li>
    <li class="nav-item {% if 'blog' in request.path %}active{% endif %} px-lg-4">
      <a class="nav-link text-uppercase text-expanded" href="{%url
'Blog'%}">Blog</a>
    </li>
  </ul>
</body>
```

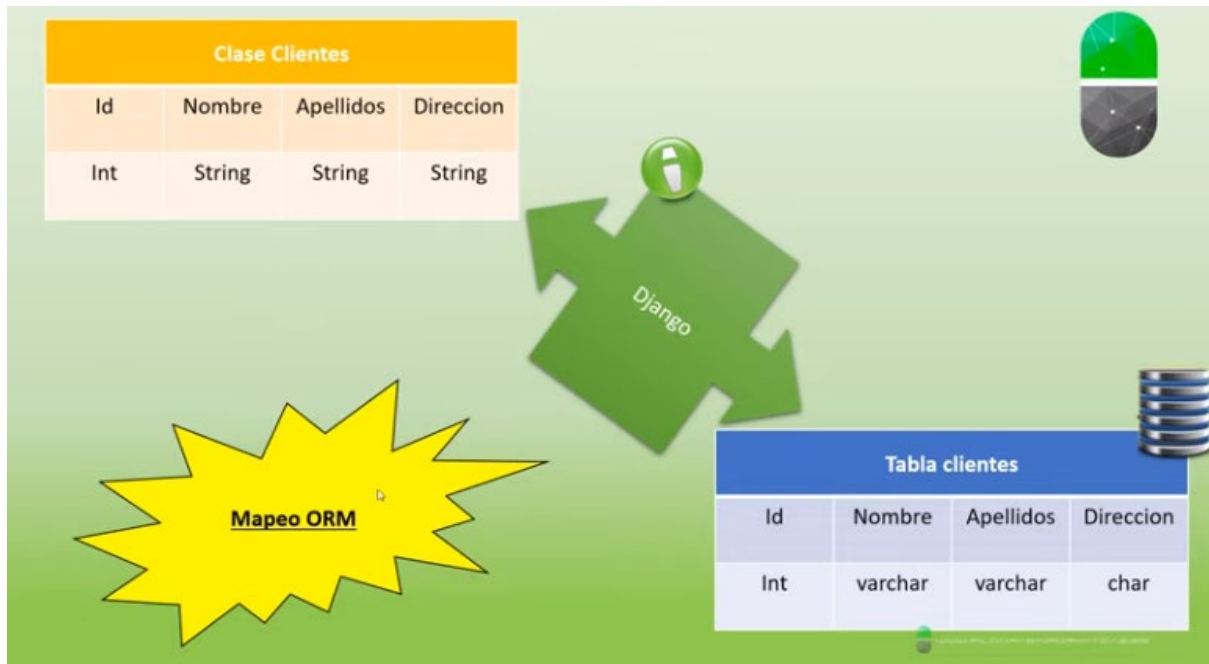
Crear la app **servicios** y registrarla en **settings.py**.

```
>>python manage.py startapp servicios
```

(En **settings.py**)

```
...
INSTALLED_APPS=[
    'django.contrib.admin',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'ProyectoWebApp',
    'servicios',
]
```

Mapeo ORM: Mapeo Objeto-Relacional: Se crea un objeto en código que representa a la tabla de la Base de Datos con sus propiedades.



Ir a **servicios/models.py**

```
class Servicio(models.Model):
    titulo=models.CharField(max_length=50)
    contenido=models.CharField(max_length=50)
    imagen=models.ImageField()
    created=models.DateTimeField(auto_now_add=True)
    updated=models.DateTimeField(auto_now_add=True)
    class Meta:
        verbose_name='servicio'
        verbose_name_plural='servicios'
    def __str__(self):
        return self.titulo
```

auto_now_add agrega la fecha **now** cuando se modifica o crea un registro.
La clase interna **Meta** modificará el nombre en singular y plural de la tabla.

Para manipular imágenes instalar **Pillow**.

```
>>pip install Pillow
```

Django por defecto **no muestra imágenes** en modo **Debug**.

Cuando se quiera subir el proyecto al servidor hay que setear el **Debug** en **False**.

(En **settings.py**)

```
DEBUG = True
```

Crear dentro de la carpeta del proyecto una carpeta **media**. En esa carpeta se subirán las fotos cargadas en **servicios**, dentro del panel de administrador.

ProyectoWeb

```
--media
--ProyectoWeb
--ProyectoWebApp
--servicios
```

(En **settings.py**)

```
import os
STATIC_URL = '/static/'
MEDIA_URL = '/media/'
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
```

Al modificar el modelo agregar para la imagen **upload_to** para crear una subcarpeta “**servicios**” dentro de **media** y que las imágenes que se suban dentro de esa app lo hagan allí.

(En **servicios/models.py**)

```
from django.db import models

class Servicio(models.Model):
    titulo=models.CharField(max_length=50)
    contenido=models.CharField(max_length=50)
    imagen=models.ImageField(upload_to="servicios")
    created=models.DateTimeField(auto_now_add=True)
    updated=models.DateTimeField(auto_now_add=True)
    class Meta:
        verbose_name='servicio'
        verbose_name_plural='servicios'
    def __str__(self):
        return self.titulo
```

Modificar el archivo **urls.py** dentro de **ProyectoWebApp** importando las configuraciones creadas en **settings.py**.

```
from django.conf import settings
from django.conf.urls.static import static
from django.urls import path
from ProyectoWebApp import views

urlpatterns=[
    path('',views.home, name="Home"),
    path('servicios',views.servicios,name="Servicios"),
    path('tienda',views.tienda,name="Tienda"),
    path('contacto',views.contacto,name="Contacto"),
    path('blog',views.blog,name="Blog"),
]
```

```
urlpatterns+=static(settings.MEDIA_URL,document_root=settings.MEDIA_ROOT)
```

Ejecutar **makemigrations** y **migrate**.

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

(En **servicios/admin.py**)

```
from django.contrib import admin
from .models import Servicio

class ServicioAdmin(admin.ModelAdmin):
    readonly_fields=('created','updated')

admin.site.register(Servicio,ServicioAdmin)
```



The screenshot shows the 'Add servicio' form in the Django admin interface. It includes fields for 'Titulo:', 'Contenido:', 'Imagen:' (with an 'Examinar...' button and a message 'No se ha seleccionado ningún archivo.'), 'Created:', and 'Updated:'. The 'Created:' and 'Updated:' fields show a '-' symbol, indicating they are read-only.

Mostrar servicios creados en las templates

(En **views.py**)

```
from django.shortcuts import render,HttpResponse
from servicios.models import Servicio
def home(request):
    return render(request,"home.html")

def servicios(request):
    servicios=Servicio.objects.all()
    return render(request,"servicios.html",{"servicios":servicios})

def blog(request):
    return render(request,"blog.html")

def tienda(request):
    return render(request,"tienda.html")

def contacto(request):
    return render(request,"contacto.html")
```

(En **servicios.html**)

```
{% extends "ProyectoWebApp/base.html" %}
{% load static %}
{% block content %}
{% for servicio in servicios %}
    <section class="page-section clearfix">
        <div class="container">
            <div class="intro">
                
                <div class="intro-text left-0 text-center bg-faded p-5 rounded">
                    <h2 class="section-heading mb-4">
                        <span class="section-heading-lower">{{servicio.titulo}}</span>
                        <span class="section-heading-upper">{{servicio.contenido}}</span>
                    </h2>
                </div>
            </div>
        </div>
    </section>
{% endfor %}
{% endblock %}
```

App Blog

Crear app con **startapp**

```
>>python manage.py startapp blog
```

Todos los post creados por un usuario deben ser dados de baja cuando el mismo da de baja su cuenta en el sitio. Eso se hace con el atributo **on_delete=models.CASCADE**.

Una categoría puede estar en **varios post** y un post puede estar en **varias categorías**, esa relación se define con **models.ManyToManyField(Categoria)**.

(En **blog/models.py**)


```

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

class Categoria(models.Model):
    nombre=models.CharField(max_length=50)
    created=models.DateTimeField(auto_now_add=True)
    updated=models.DateTimeField(auto_now_add=True)
    class Meta:
        verbose_name="categoria"
        verbose_name_plural="categorias"

    def __str__(self):
        return self.nombre

class Post(models.Model):
    titulo=models.CharField(max_length=50)
    contenido=models.CharField(max_length=50)
    imagen=models.ImageField(upload_to="blog",null=True,blank=True)
    autor=models.ForeignKey(User,on_delete=models.CASCADE)
    categorias=models.ManyToManyField(Categoria)
    created=models.DateTimeField(auto_now_add=True)
    updated=models.DateTimeField(auto_now_add=True)
    class Meta:
        verbose_name="post"
        verbose_name_plural="posts"

    def __str__(self):
        return self.titulo

```

Registrar app en **settings.py**

```

...
INSTALLED_APPS=[
    'django.contrib.admin',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'ProyectoWebApp',
    'servicios',
    'blog',
]
...

```

Crear **migraciones**.

```

>>python manage.py makemigrations
>>python manage.py migrate

```

(En **blog/admin.py**)

```
from django.contrib import admin
from .models import Categoria, Post

class CategoriaAdmin(admin.ModelAdmin):
    readonly_fields=("created", "updated")

class PostAdmin(admin.ModelAdmin):
    readonly_fields=("created", "updated")

admin.site.register(Categoria, CategoriaAdmin)
admin.site.register(Post, PostAdmin)
```

Crear **primero** las **categorías** y **después** los **posts**.

(En **views.py** dentro de la carpeta **ProyectoWebApp**)

```
from django.shortcuts import render, HttpResponse
from servicios.models import Servicio
from blog.models import Post

def home(request):
    return render(request, "home.html")

def servicios(request):
    servicios=Servicio.objects.all()
    return render(request, "servicios.html", {"servicios":servicios})

def blog(request):
    posts=Post.objects.all()
    return render(request, "blog.html", {"posts":posts})

def tienda(request):
    return render(request, "tienda.html")

def contacto(request):
    return render(request, "contacto.html")
```

(En **blog.html**)

```

{% extends "ProyectoWebApp/base.html" %}
{% load static %}
{% block content %}
{% for post in posts %}
    <section class="page-section clearfix">
        <div class="container">
            <div class="intro">
                
                <div class="intro-text left-0 text-center bg-faded p-5 rounded">
                    <h2 class="section-heading mb-4">
                        <span class="section-heading-lower">{{post.titulo}}</span>
                        <span class="section-heading-upper">{{post.contenido}}</span>
                    </h2>
                    <div style="text-align: left; font-size: 15px;">
                        Autor: {{post.autor}}
                    </div>
                </div>
            </div>
        </div>
    </section>
{% endfor %}
<section>
    <div style="width: 75%; margin: auto; text-align: center; color: white;">
        Categorías:
        {% for post in posts %}
            {% for categoria in post.categorias.all %}
                {{categoria.nombre}}
            {% endfor %}
        {% endfor %}
    </div>
</section>
{% endblock %}

```

Deployment

Leer artículo:

<https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Deployment>

Tipos de infraestructura cloud: IaaS, PaaS y SaaS

Fuente: tecon.es

La **nube** es un tipo de tecnología que permite tener todos nuestros archivos y ficheros en Internet, sin tener que preocuparnos por si hay capacidad suficiente para almacenar información o programas en nuestro ordenador o servidor local. ¿Qué tipos de nube existen y cómo funcionan? Te explicamos las diferencias entre los tipos de infraestructura cloud: IaaS, PaaS y SaaS.

En primer lugar, hay que destacar que las tres modalidades de hosting cloud tienen varios aspectos en común:

1. Proporcionan acceso a unos recursos que se encuentran en la nube – datacenters con gran cantidad de servidores – a los que se accede mediante Internet.
2. Pagas por el espacio o rendimiento que usas.
3. Pueden ser utilizadas por las empresas para crear aplicaciones o soluciones con poca inversión. Cuando se han probado, es muy fácil ampliar recursos.
4. La administración del hardware la lleva a cabo el proveedor del servicio cloud, ahorrando costes de mantenimiento a las empresas.

Tipos de infraestructura en la nube

IaaS, Infrastructure as a Service

La Infraestructura como Servicio ofrece recursos informáticos equivalentes a hardware virtualizado. La IaaS comprende aspectos como el espacio en servidores, conexiones de red, ancho de banda, direcciones IP... El IaaS permite construir una infraestructura propia mediante distintos componentes virtualizados.

Entre las ventajas de la IaaS, destaca su alta escalabilidad. Estas infraestructuras permiten ampliar su capacidad en función de las necesidades concretas de tu negocio. Además, ya no son necesarias grandes inversiones en hardware ni afrontar sus costes de amortización.

PaaS, Platform as a Service

Este tipo de arquitectura facilita un entorno adecuado para construir aplicaciones y servicios en Internet, incluyendo networking, almacenamiento, soporte de software y servicios de gestión.

Entre las ventajas del PaaS, destacan: su flexibilidad, pues el cliente tiene el control pleno sobre las herramientas instaladas en su plataforma; la adaptabilidad, ya que puedes cambiar las características en función de tus necesidades; la movilidad, puesto que tienes acceso a la plataforma desde cualquier dispositivo sólo con conectarte a Internet; y la velocidad en el desarrollo, debido a que con esta infraestructura el entorno estará preparado para tal efecto – muchos incluyen CMS.

SaaS, Software as a Service

Por último, este concepto hace referencia a un tipo de servicio cloud por el cual los clientes pagan por el acceso a una aplicación en Internet. Es el caso de soluciones de negocio como CRM, ERP y otras soluciones de productividad. Una de sus grandes ventajas son el acceso multidispositivo – ya que sólo necesitas disponer de conexión a Internet – y el ahorro en costes. Con soluciones SaaS tienes a tu disposición soluciones profesionales de gran calidad, evitando que queden obsoletas, ya que alquilas su uso junto con otros profesionales y empresas (sólo asumes el coste que requiera su configuración y coste por número de usuarios al mes, no pagas por licencia). Grandes proveedores de esas tecnologías son los que se encargan de la creación de la aplicación, su almacenamiento, administración, seguridad y mantenimiento. Con este modelo, ¡ahorrarás tiempo, dinero y quebraderos de cabeza!

Nube pública, híbrida y privada

Además de las diferencias en arquitectura, nos encontramos con otra clasificación: nube pública, híbrida y privada.

- Se considera nube **pública** cuando todos los servicios se ofrecen desde servidores externos a la organización que los contrata. Estos servicios son administrados por terceros y los trabajos de los usuarios de esa “nube” pueden estar mezclados en los distintos servidores o sistemas de almacenamiento. Los usuarios finales no conocen qué procesos y con quién están compartiendo espacio y recursos. La carga operacional y la seguridad de los datos (backup, firewall, accesibilidad, etc.) recae íntegramente sobre el proveedor de servicios cloud. Por esta razón, el riesgo en la disponibilidad de servicio de estas tecnologías es mucho menor que cuando se trabaja con una infraestructura propia y local.
- Llamamos nube **privada** a aquella que es de uso exclusivo de una entidad. Se trata de un centro de datos completamente virtualizado con autoservicio y automatización. Estos servicios se virtualizan y se agrupan en pools para garantizar que se utilice la capacidad máxima de la infraestructura física que los contiene. Ofrecen una mejor eficiencia de los recursos, aunque la gestión propia es más compleja que en la utilización de otras nubes.
- Por último, la nube **híbrida** combina los modelos de nube pública y privada, tanto local como en la nube. La organización es propietaria de unas partes y comparte otras, aunque de una manera controlada. Las nubes híbridas ofrecen la ventaja del escalado bajo demanda pero añaden la complejidad de determinar cómo distribuir las aplicaciones a través de los diferentes entornos. Una nube híbrida tiene la ventaja de poder realizar una inversión inicial más moderada en infraestructura local ya que se combina con IaaS bajo demanda. Utilizando las APIs de las distintas nubes públicas existentes se tiene la posibilidad de escalar sin invertir en más servidores físicos. Suele tener buena aceptación en las organizaciones que ya cuentan con servidores propios y quieren aprovechar las ventajas del pago por uso y otras características de la nube.