**Framework de programación Django2**

Para esta formación, principalmente nos apoyaremos en la lectura del libro:..

**La guía definitiva de Django** escrito por Adrian Holovaty y Jacob Kaplan. Traducido por Saúl García

Y por los cursos:...

* **Curso Maestro de Django 2 Desarrollo Web Backend con Python**
* **Curso Django por Píldoras informáticas (youtube)**
* **Curso Djando 2 por developers.pe (Youtube)**

**La instalación de Python y de Django está muy detallado en mi documento “Configuración de mi GNU Linux para desarrollo”**

Iniciamos esta nueva y fascinante aventura después de la espectacular y muy agradable picada de condigo en Python. Hoy, viernes 3 de Abril y en plena cuarentena, vamos a iniciar con este poderoso Framework para estar muy pronto con nuevos proyectos, otros desafíos y porqué no... Un buen empleo.

Los genios creadores de MOSAIC, el primer navegador de Internet gráfico, Un grupo de emprendedores del NCSA(Centro Nacional de Aplicaciones para Super computadoras) crearon un protocolo llamado CGI lo que cambió para siempre nuestro mundo y la manera y cantidad de tiempo que consumimos Internet.

CGI nos permitió pensar en la primera generación de páginas web generadas de manera dinámica desde un servidor bajo demanda, y olvidarnos de archivos HTML estáticos guardados como simples archivos de disco.

Sin embargo los script CGI contienen gran cantidad de código repetitivo, lo cual hace que sean poco re-utilizables y tediosos de mantener con el tiempo. Entonces aparece la estrella de esta película en escena... **El lenguaje de programación PHP.** Con PHP quedaron atrás varios de los problemas que tenía CGI. Y además de eso, PHP era y sigue siendo MUY fácil de aprender y de implementar. Fue creado específicamente para la Web, de manera que no necesita obligatoriamente de frameworks o plataformas complejas. Basta con su interprete y un navegador Web para comenzar a crear magia que, en sus inicios iba embebida en código HTML.

Estuvo tan bien pensado que para muchos entornos, plataformas y lenguajes, se fijaron en el diseño de PHP(entre ellos ASP, JSP y otros). Poco tiempo después el crecimiento en el número de usuarios de la Web fue evidentemente exponencial, y PHP tuvo que evolucionar hacia la orientación a Objetos para hacer de sus códigos, scripts re-utilizables, más entendibles y fáciles de mantener para hacer proyectos escalables. Esto, por medio de las Clases y conceptos como la herencia y el encapsulamiento. Se generó entonces una Explosión por el uso de la Web, lo que condujo a que los desarrolladores Web produjeran mas y mas cada día. Y esta exigencia llevo a la necesidad de implementar proyectos y/o plataformas que normalizaran las cuestiones comunes en los desarrollos informáticos. Y apareció entonces la necesidad de crear tanto para PHP, como para otros lenguajes, los famosos **Frameworks** de desarrollo.

**¿Que es un Framework Web?:...**

No se puede negar que los programadores somos como el común de la gente... Queremos las cosas lo más fácil posible, ahorro de esfuerzo, lo que conlleva a tener más tiempo para otras ocupaciones o para nuestros caprichos :)

Un marco de trabajo o Framework es esencialmente un programa que provee principalmente una infraestructura muy organizada que normaliza(o estandariza) la manera de implementar código,no solo para nosotros, pero también para los futuros lectores de nuestro código. Además provee bibliotecas, paquetes, Clases métodos y otras funcionalidades preestablecidas que son comunes en los proyectos Web, para no tener que reinventar la rueda y de manera que podamos centrar nuestra atención en la “**singularidad**” de nuestro proyecto.

En el caso de Python, y así como para PHP, hay varios frameworks Web. Para Python los que destacan son:...

**Pyramid**

**Bottle**

**Django**

**Flask**

de los anteriores, los más robustos son Django y Pyramid.

Y bueno, lo que nos ocupa en este estudio es el Framework Django....

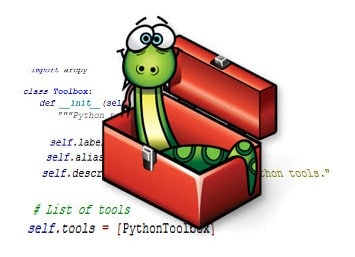
Los objetivos principales de Django, son los mismos que los de otros frameworks:... agilizar de la mejor manera el trabajo de los desarrolladores y unificar la estructura del código a nivel mundial, de manera que sin importar la nacionalidad y ubicación, nuestro proyecto sea entendible para cualquier otro programador Django.

Es una de las ventajas de utilizar un Framework... Podemos hacer foco en la singularidad de nuestro proyecto, así, las partes comunes a todo proyecto y las partes repetitivas son tarea de nuestro framework.

La convención es una de las tareas más importantes para utilizar estos marcos de trabajo, evitamos así... código desordenado, y caos in-entendible tanto para nosotros como para los futuros lectores de nuestros scripts

Pero Django nos permite, de manera sutil y organizada, separarnos del framework cuando lo consideremos necesario.

Aprenderemos entonces, todo lo necesario para producir el desarrollo y despliegue de nuestros futuros proyectos Web y como dice el libro... “Nos vamos a sentir orgullosos de ello” (guiño guiño :P )



**Requisitos recomendables para iniciar con DJango:**

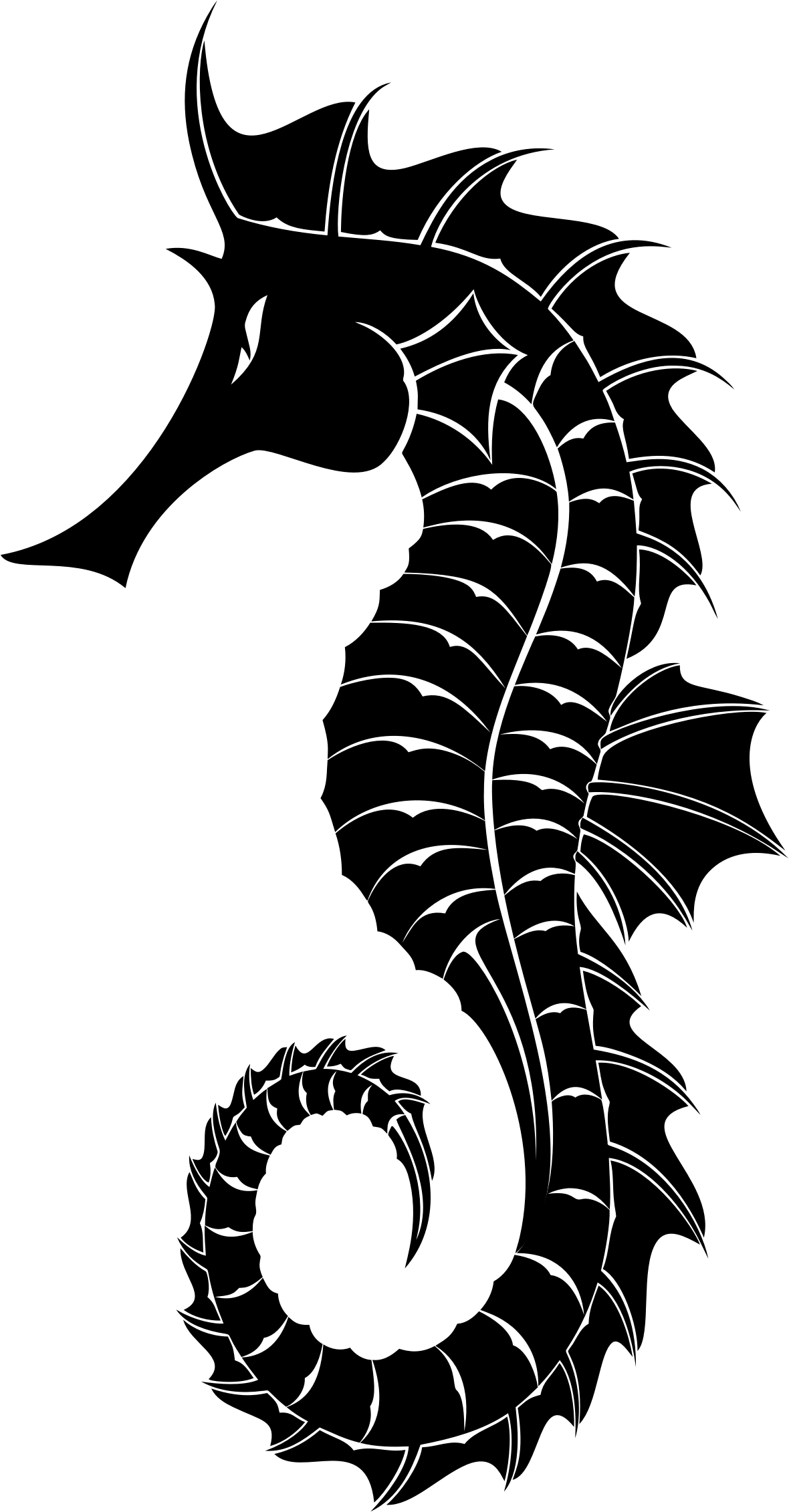
* **tener conciencia de lógica de programación y algoritmos**
* **Tener muy buen gusto musical (Kreator, Death, Megadeth, etc :P)**
* **Es muy recomendable conocer el lenguaje Python3**
* **se recomienda utilizar el sistema operativo GNU/Linux Deepin porque China es la ostia tio :) y la verdad es que este sistema es el que mejor me ha funcionado**
* **saber modificar plantillas HTML y archivos CSS o en su defecto saber implementar y manipular un framework como Bulma, Twiter Bootstrap, Foundation, etc**

**Instalación y configuración de las herramientas necesarias**

Para la puesta en marcha de Django en nuestra computadora, Tenemos los pasos necesarios para llegar hasta crear un proyecto básico de Django en nuestra guía de instalación de GNU/Linux deepin.

De manera que no vamos a entrar en detalles sobre dicha instalación y configuración. Y entraremos directamente en el aprendizaje del Framework como tal.

**El Framework Django:...**



Django se convirtió con el tiempo en un poderoso proyecto para desarrollar productos de nivel empresarial. Voy a tratar de ser lo más claro posible en los puntos fundamentales para el aprendizaje de este framework de manera que pueda ser una guía de mucha utilidad para otros noveles en Django.

Espero que con este libro se puedan divertir tanto como yo lo estoy haciendo y desde luego, espero que sea tan útil como lo es y lo seguirá siendo para mi.

Para comenzar a entender Django, tenemos que comprender de manera general su modelo, el cual nos presenta la manera de estructurar nuestro código... ,metodología Unificada. Esto hace que todos los programadores Django trabajen e implementen un código en completa armonía.

**Modelo de programación en Django:..**

Para normalizar nuestro código, los creadores de DJango basaron su diseño en el famoso **“Modelo-Vista-Controlador”,** en el cual el **Modelo** contiene los paquetes, archivos y códigos que se conectan y comunican directamente con los datos almacenados en uno o unos servidores. el **Controlador** se encarga de manipular los datos que provienen del usuario(Vista) o de la fuente de datos(Modelo) y realizar operaciones sobre ellos, como también se encarga de enviar solicitudes al Modelo. Y la **Vista** presenta el resultado de esas operaciones al usuario mediante el navegador web.

Podemos comprender que hay una evidente separación de código que nos presenta una estructura muy ordenada y mucho más fácil de mantener y de escalar.

**Model-View-Template o MTV:...**

**El modelo de estructurar el código en Django se llama Model-Template-View donde:..**

**Model**:... es la capa de acceso a la fuente de Datos

**Template**:.. esta capa se encarga de entregar los datos ya sea a una página Web o a otro tipo de documento

**View:...** es la capa que sirve de puente entre el Modelo y la Plantilla, creo que hace parte del controlador, porque Django maneja al controlador de manera interna, así, nosotros como programadores manipulamos por lo general al View.

Para comprender mejor y estar al tanto de las mejoras y modificaciones de fututras versiones de Python, el traductor a español de “**el libro de Django2**” recomienda descargar y leer la versión en ingles de este libro, la cual está disponible en http://www.djangobook.com/

Como también es posible conectarse con la comunidad más grande de usuarios de Django y hacer preguntas o aportar mejoras e ideas al proyecto, podemos suscribirnos(cuando aprendamos ingles) en

http://www.djangoproject.com/r/django-users

Bueno. Vamos a verificar la versión de Django que tenemos instalada en nuestro poderoso X-200 :) ...(mi portátil portátil Lenovo cargado con GNU/Linux Deepin)

* ejecutamos la Terminal

Podemos verificar la versión instalada de varias maneras:...

1. La primera es cuando el usuario ha instalado una versión en su máquina local directamente sobre el sistema operativo(No es nuestro caso)ejecutando la Shell de Python con el comando **python3** momento en el que se activa la shell de Python, acto seguido, se ejecuta la importación del módulo django con **import django** y después de dar ENTER se llama a la función correspondiente para tener la info. que deseamos con **django.VERSION**. Al presionar ENTER podrémos verificar la versión que tiene la máquina instalada
2. otra manera (y es la que nos ocupa) es cuando hemos instalado **virtualenv** en nuestra computadora y hemos creado un entorno virtual o virtual environment. Nos dirigimos mediante la **terminal** al directorio donde hemos creado el entorno virtual que en nuestro caso esta dentro del directorio Developer/Django2 . De manera que mediante la terminal quedamos posicionados en:...

**Demonscript/Django2**

acto seguido, ejecutamos...

**source bin/activate** y así se dispara nuestro virtual environment en el cual ya hemos instalado Django

ahora podemos ejecutar el comando **django-admin.py —version**

con la ejecución de ese comando podemos ver que nuestra versión de Django es la versión **2.2.12**

**{{ PARTE 1 }}** **(Nivel básico)**...

**Crear primer Proyecto Django, en el cual veremos la manera básica en que funciona Django, aprenderemos cómo se cinfigura inicialmente, cómo manipular sus archibos escenciales, qué son y cómo crear vistas, cómo crear Apps, hacer registros de cambios elementales y realizar migraciones o registros de cambios en nuestro proyecto, manipular el panes de administración, ingresar datos desde la interfaz Web sobre la BD y mostrar mediante el navegador Web todos los resultados conseuentes**

**Este proyecto dará vida a un Blog Personal:...**

Ahora creamos nuestro primer proyecto, recordemos que podemos hacerlo mediante la terminal del sistema, mediante la terminal del IDE Eclipse o de la manera sencilla... mediante la interfaz gráfica de Eclipse IDE:

**Crear proyecto mediante Comandos de la Terminal:..**

Mediante cualquiera de las dos terminales(en este caso mediante la terminal del IDE Eclipse):

Posicionados en el lugar donde ya hemos creado nuestro entorno virtual y donde ya lo activamos(source bin/activate). Lanzamos el comando:

**django-admin.py startproject Django\_project1**

De esta manera hemos creado el primer proyecto Django al cual hemos llamado **Django\_project1**

Podemos explorar su contenido y desde la terminal ejecutamos un servidor de pruevas para verificar su funcionamiento....

Ingresamos al directorio que contiene nuestro nuevo proyecto...

**cd Django\_project1**

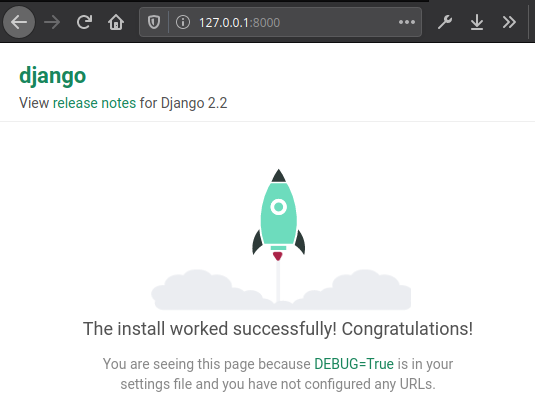
y ahora ejecutamos el comando que activa el servidor web de pruebas...

**python3 manage.py runserver**

En este momento la terminal de comandos nos muestra una serie de procesos llevados a cabo para lanzar el servidor Web.

Al final de toda la información nos muestra entre otros datos, la dirección en la cual está corriendo nuestro proyecto. en nuestro caso fue **http://127.0.0.1:8000/** dirección que podemos copiar y visitarla mediante el poderoso Firefox u otro navegador Web.

Al visitar el sitio por nuestro navegador Web, podemos ver algo muy parecido a la siguiente imagen:..



Comprobamos así, que nuestro proyecto está bien creado y corre sin problemas :)

**Crear proyectos mediante la interfaz gráfica de Eclipse IDE:...**

Esta es la mejor opción si queremos trabajar nuestros proyectos con **github**, ya que por teminal me generaba problemas porque ya existía un directorio(el de git) con el mismo nombre del proyecto a crear y el cual ya tenía archivos(los que se generan al crear el repo).

entonces lo que primero debemos hacer es crear el repositorio **Django\_project1** en local y en github mediante **GitKraken** y posteriormente creamos el projecto Django en este mismo directorio mediante la interfaz gráfica de eclipse....

teniendo abierto nuestro IDE, damos clic en **file,new,project**

seleccionamos la opción **PyDev Django Project** y damos clic en **Next**

En la nueva ventana damos nombre a nuestro nuevo proyecto(**Django\_project1**)

desactivamos la opción **Use Default** y en el botón **Browser** navegamos hasta **Developer/Django\_project1**, ingresamos en dicho directorio y damos clic en el botón **abrir** con lo que regresamos a la ventana anterior y seleccionamos el interprete que en este caso será python 3.5 y presionamos el botón **Next**

En la siguiente interfaz elegimos la versión más cercana a nuestra versión de Django, botón **Next**

En la siguiente interfaz no refenciamos otro proyecto, botón Next

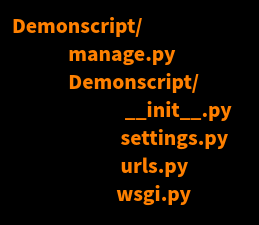
dejamos de momento el gestor de bases de datos por defecto, que es SqlLite3

Presionamos el botón **Finish**

Se crea entonces nuestro proyecto y podemos explorarlo desde la parte derecha de nuestro IDE.

Para ejecutar utilizamos la terminal. **python3 manage.py runserver** y visitamos nuestro poryecto de nuevo, como siempre mediante nuestro navegador Web

Al crear un nuevo proyecto de Django, se genera de manera automática el siguiente sistema de archivos:...



Donde:...

1. El directorio principal **Demonscript** es nuestro contenedor, que ademas de contener todo nuestro proyecto, tiene el archivo **manage.py**

El nombre de nuestro directorio contenedor lo podemos cambiar y esto NO afecta la funcionalidad del proyecto.

1. **manage.py** es el archivo que nos provee varias funcionalidades que iremos conociendo al paso de este curso. Para comenzar nos provee los códigos que levantan el servidor Web para nuestro Proyecto. Ya lo ejecutamos con el comando...

**python3 manage.py runserver**

manage.py nos permite interactuar de muchas otras formas con nuestro proyecto. Podemos ejecutar el comando **python3 manage.py help**  para ver todas las funcionalidades lo que nos provee. Lo mismo podemos hacer al ejecutar solamente **python3 manage.py**

Este archivo no debemos editarlo ni moverlo de su lugar ya que convenientemente ha sido puesto allí para gestionar nuestro proyecto

1. **Django\_project1/Django\_project1** Contiene en principio, todo lo que necesitamos para codificar con Python nuestro proyecto, como \_\_init\_\_.py , settings.py , urls.py y wsgi.py
2. **\_\_init\_\_.py** Es un archivo requerido para que Python trate el directorio interno **Django\_project1** como un paquete o grupo de módulos, a este archivo por lo general NO necesitamos modificarlo.
3. **settings.py** es donde vamos a ingresar o implementar configuraciones para nuestro proyecto Django. tiene ya unos valores y configuraciones predefinidos, poco a poco lo vamos a conocer y aprender a manipular
4. **urls.py** Es un archivo para declarar las url’s de nuestro proyecto. Podemos pensar que es como una TABLA de contenidos de nuestro site hecho con Django
5. **wsgi.py** el punto de entrada WSGI para el servidor Web, es el archivo encargado de “servir” nuestro proyecto, este archivo también nos servirá de mucha utilidad para el proceso de despliegue de nuestro proyecto a **producción**

Adicional a esto, en el directorio principal tenemos para este proyecto un archivo **db.sqlite3** que especifica la base de datos creada con el SGBD Sqlte3 para nuestro proyecto.

En la parte de abajo del archivo **settings.py** hay una variable como en formato Json llamada **INSTALLED\_APPS** Contiene el nombre de todas las aplicaciones Django que están activadas para nuestra instancia(nuestro proyecto se puede definir como una instancia de Django). Las aplicaciones pueden ser empacadas y distribuidas para ser usadas por otros proyectos.

de forma predeterminada INSTALLED\_APPS contiene todas las aplicaciones que vienen por defecto en nuestro proyecto Django, las cuales son:...

* **django.contrib.admin** La interfaz administrativa.
* **django.contrib.auth** El sistema de autentificación.
* **django.contrib.contenttypes** Un framework para tipos de contenidos.
* **django.contrib.sessions** Un framework. para manejar sesiones
* **django.contrib.messages** Un framework para manejar mensajes
* **django.contrib.staticfiles** Un framework para manejar archivos estáticos.

Las aplicaciones anteriores se incluyen por defecto y de manera conveniente para los casos más comunes en todo proyecto. Algunas de esas aplicaciones hacen uso de la Base de datos del proyecto, razón por la cual se deben crear tablas para ellas en la BD, antes de poder utilizarlas. Para ello vamos al directorio contenedor de nuestro proyecto Demonscript y ejecutamos el sig. comando para que nuestro proyecto funcione correctamente :...

**python3 manage.py migrate**

el anterior comando verifica la variable **INSTALLED\_APPS** y crea en la BD las tablas necesarias para cada una de las aplicaciones registradas en el archivo **settings.py**, por cada migración aplicada veremos un OK.

**El servidor para pruebas de Desarrollo:...**

Es el servidor que se despliega cuando ejecutamos el comando...

**python3 manage.py runserver**

éste es un servidor Web Liviano usado unicamente mientras estamos en el proceso de desarrollo y hasta que el proyecto está listo para producción(momento en el cual se utiliza un servidor Web robusto como Apache o Nginx)

a parte de ser ligero, nuestro servidor Web de Python vigila nuestro código en espera de cambios y se reinicia automáticamente mostrando los cambios sin que tengamos que reiniciar nada nosotros mismos.

recordemos que, al ejecutar el servidor web de nuestro proyecto, podemos copiar la url que muestra la terminal y pegar esa url en nuestro navegador Web favorito para visitar nuestro proyecto o sitio Web.

**Cambiar el host y el puerto para lanzar nuestro proyecto**:...

Por defecto nuestro site se lanza en el puerto 8000 Pero podemos cambiar el puerto indicándolo como parámetro en el comando que lanza el servidor, por ej...

**python3 manage.py runserver 8080**

si tenemos nuestra computadora en red y queremos compartir el proyecto, Lo que tenemos que hacer es obrener nuestra ip. que en GNU/Linux la obtenemos con el comando **ifconfig** En nuestro caso la dir ip es 192.168.0.21

de manera que ejecutamos el comando...

python3 manage.py runserver 192.168.0.21:8000

Creo que es así. sujeto a modificaciones

**parar o frenar el servidor Web de Django:...**

Muchas veces no funciona el simple control + c en la terminal para frenar el proceso. Pero podemos matar el proceso con el comando...

**sudo fuser -k -n tcp 8000**

**Nuestra Primera página Web creada con Django:...**

Bien, ya vimos cómo crear proyectos Web de Django y cómo lanzarlo en nuestro servidor de pruebas. Desde luego que el site aún no hace algo útil. Ahora vamos a crear nuestra primera página Web dinámica.

En Django, el contenido de la página será producido por la función vista y la URL se especificará en la URLconf.

**Primera vista creada con Django:...**

para comenzar, quiero cambiar el idioma de mi proyecto a español. Eso lo hacemos buscando en el archivo settings.py la variable LANGUAGE\_CODE y modificamos su valor o contenido a es-esp, de manera que quedará así:...

**LANGUAGE\_CODE = ‘es-esp’**

guardamos los cambios y lanzamos o recargamos el servidor

**Ahora vamos crear la primera vista en nuestro proyecto....**

1. si vamos al archivo **settings.py** y buscamos la variable **DEBUG** podemos ver que está en **True** y esto se debe a que estamos en un entorno de **DESARROLLO**. De ninguna manera se debe dejar con este valor en el momento en que el proyecto pasará a un entorno de producción, porque en ese estado, Django nos muestra en caso de algún error, una serie de información que es confidencial y que no debe estar pública en un entorno de **PRODUCCIÓN**
2. como pudimos ver en el archivo **settings.py**, Django provee un sistema de re-utilización de código que nos lo presenta en una manera de aplicaciones internas de nuestro proyecto. de momento ya tenemos varias aplicaciones disponibles...

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles'

Aplicaciones que están especificadas en la variable **INSTALLED\_APPS**

ya sabemos que cada una de esas aplicaciones implementan funcionalidades especificas.

Podemos agregar nuevas **apps** de terceros o creadas por nosotros mismos para mejorar, personalizar o escalar nuestro proyecto, así, esas apps las podremos utilizar en multiples proyectos. Podemos concluir que una app es una funcionalidad de un proyecto Web, y a su vez, un proyecto web se compone de muchas funcionalidades.

Vamos entonces a crear nuestra primera app, la cual será en núcleo de este pequeño proyecto y a partir de esta app crearemos nuestra primera vista...

1. Teninedo nuestro entorno activo, nos dirigimos al directorio contenedor de nuestro proyecto y lanzamos el comando...

**python3 manage.py startapp core**

Actualizamos nuestro proyecto en Eclipse IDE y podemos ver que el anterior comando nos crea un paquete/directorio con varios módulos de python, los cuales vamos a conocer poco a poco. de momento nos centramos en el archivo **views.py** que es muy importante porque en él se definen las vistas de nuestra app.

Las vistas hacen referencia a la lógica que se ejecuta cuando el cliente/usuario hace una petición en nuestro proyecto Web. Crearemos entonces una vista para procesar la petición a la raíz de nuestro proyecto Web, esto es, a la portada o página inicial de nuestra Web...

1. Al abrir el archivo views.py vemos que viene por defecto con el método **render,** importado del módulo **django.shortcuts**

De ese mismo módulo necesitamos importar un método llamado **HttpResponse.** como pertenece al mismo módulo, podemos añadirlo así:..

**from django.shortcuts import render, HttpResponse**

el método HttpResponse nos permite responder a una solicitud del tipo HTTP. Definimos nuestra vista para la portada y retornaremos un código HTML...

1. el archivo nos sugiere que implementemos nuestras vistas debajo del comentario #create your views here, y así lo vamos a hacer...

def index (request):

return HttpResponse(“<h1>Hello Demonscript</h1>”)

Donde:... request es la petición que recibimos como parámetro, y retornamos una respuesta HTML mediante el método HttpResponse

Ahora le indicamos a Django en qué URL debe mostrar dicha respuesta. Para ello, vamos al archivo urls.py de nuestro proyecto y vemos las indicaciones que tenemos a modo de comentario en function views. donde se nos indica que debemos importar el módulo **views.py** de nuestra nueva app (la que llamamos **core** ) y lo hacemos debajo de lo que ya está importado en el archivo, nuestro código añadido será...

**from core import views**

1. lo segundo que nos indica el comentario es añadir una URL a la variable **urlpatterns.** Agregamos entonces un nuevo path, y como es en la página inicial, el primer parámetro deberá estar entre comillas simples sin llevar NADA, el segundo parámetro es el llamado al método index que definimos en views.py y el tercero es un nombre o “alias” que le damos por defecto a ese proceso...

urlpatterns = [

**path('',views.index,name='index'),**

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

]

El objetivo con nuestro código agregado es que cuando se cargue mediante el navegador a la página inicial del sitio, el proyecto ejecute el método o vista index del módulo views.py

ejecutamos el servidor y abrimos la url en el navegador, que en nuestro caso es http://127.0.0.1:8000/

1. De manera similar podemos crear nuevas vistas para solicitudes desde diferentes sitios de nuestra web. Por ejemplo, vamos a crear una vista para responder a un llamado diferente al de la página inicial de nuestro site, por ejemplo http://127.0.0.1:8000/info...
2. **añadimos una nueva vista en views.py...**

**def info(request):**

**return HttpResponse(**

**‘’’**

**<title>Demonscript</title>**

**<center><h1>Hello, this is my site :)</h1>**

**<h2>Información del sitio</h2></center>**

**‘’’**

**)**

1. Ahora le indicamos a Django dónde debe mostrar dicha respuesta y para ello regresamos al archivo urls.py y agregamos el nuevo path...

urlpatterns = [

**path('info/',views.info,name='info'),**

path('',views.index,name='index'),

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

]

1. Ejecutamos de nuevo el servidor y en el navegador visitamos la URL:...

http://127.0.0.1:8000/info/

Podemos ver que hasta ahora, la dinámica es indicar desde urls.py el código que se debe mostrar desde views.py

Vamos a seguir trabajando en estos dos archivos, ahora pongamos un código compartido entre páginas:..

1. En views.py vamos a agregar el código compartido, dentro de una variable y antes de las vistas implementadas...

**commonhtml ='''**

**<title>Demonscript!!:.. </title>**

**<center>**

**<h1> Mi Sitio Personal</h1>**

**</center>**

**<ul>**

**<li><a href='/'>Home</a></li>**

**<li><a href='/info'>Info del sitio</a></li>**

**<li></li>**

**<li></li>**

**</ul>**

**'''**

1. y ahora en cada vista, implementamos su código singular agregado al código compartido, por ej. para el index...

**def index(request):**

**return HttpResponse( commonhtml +**

**'''**

**<center>**

**<h2>Bienvenido/a a mi sitio Web Django</h2>**

**<p>Hola, soy Demonscript y éste es mi primer site con<br>**

**el Poderoso Framework Django de Python :)</p>**

**</center>**

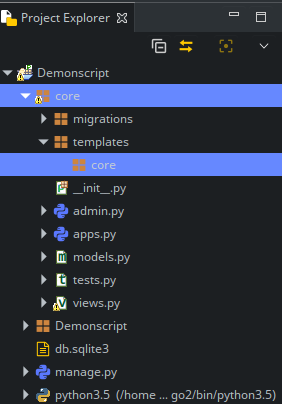
**'''**

**)**

1. Al visitar las páginas en el navegador, podremos ver el resultado y probar cada link. Supongo que este es el inicio mas rustico que le podemos dar a Django porque la manera correcta es mediante Plantillas.

**Intro. a las Plantillas o TEMPLATES:...**

1. Crear plantillas bajo la norma Django. Debemos para ello, crear un directorio que llamaremos **templates** en nuestro directorio **core** y dentro del directorio templates crearemos un nuevo directorio que llevará el nombre de nuestra app ... **core** de manera que, por cada nueva app que implementamos, debemos crear un directorio dentro de templates con el mismo nombre de dicha app, así...



1. y dentro del subdirectorio **core** que pertenece a **templates** vamos a implementar nuestras plantillas o archivos html. Creamos el primer archivo **index.html** y dentro de este vamos a pegar el código html que está en views.py y que corresponde a la vista index...

**<h2>Bienvenido/a a mi sitio Web Django</h2>**

**<p>Hola, soy Demonscript y éste es mi primer site con<br>**

**el Poderoso Framework Django de Python :)</p>**

1. modificamos el código de la vista index ...

**def index(request):**

**return render(request, "core/index.html")**

1. Pero esto aún no funciona, porque al visitar la página, el navegador muestra el mensaje..

**TemplateDoesNotExist at /...**

El mensaje anterior y su contenido, se debe a que no hemos agregado nuestra app **core** en la lista de **INSTALLED\_APPS** en el archivo **settings.py** de manera que procedemos a agregar nuestra app en la variable que contiene la lista, y quedaría así..

**INSTALLED\_APPS = [**

**'django.contrib.admin',**

**'django.contrib.auth',**

**'django.contrib.contenttypes',**

**'django.contrib.sessions',**

**'django.contrib.messages',**

**'django.contrib.staticfiles',**

**'core'**

]

1. ahora si podemos visitar nuestra página desde el navegador. Podemos ver que separamos el código html de el código python.

Procedemos a hacer el mismo procedimiento con la vista **info**

**HERENCIA DE PLANTILLAS:...**

Como ya sabemos, uno de los objetivos principales de la herencia en el código es eliminar el exceso de código repetitivo, y facilitarnos la edición futura para todos los archivos y referente a dicho código heredado.

Para nuestro ejercicio vamos a utilizar dos template tags o etiquetas de templte de Django(existen muchos, los cuales irémos manejando en el proceso de aprendizaje).

**NOTA:** Los Template tags de Django se usan para implementar lógica de programación en medio del código HTML.

los dos template tags que, por ahora vamos a utilizar son: **block content** y **extends**. los cuales los utilizamos así:...

1. creamos nuestro archivo html dentro de **templates/core** que en nuestro caso lo llamamos common.html
2. en dicho archivo que será nuestra palntilla heredada, implementamos el código común a los otros dos archivos html (index.html e info.html), el cual es el código que genera el menú de navegación.

Y para el código entrante de las demás páginas que heredan de esta plantilla, implementamos la etiqueta **block content**, la cual indica el lugar donde se mostrará el código de la página que hereda...

<html lang="es-esp">

<head>

<meta charset="UTF-8">

<title>Demonscript!!..</title>

</head>

<body>

<center>

<h1> Mi Sitio Personal</h1>

</center>

<ul>

<li><a href='/'>Home</a></li>

<li><a href='/info'>Info del sitio</a></li>

<li>pendiente</li>

<li>pendiente</li>

</ul>

<!-- Content:... -->

**{% block content %}**

**{% endblock %}**

</body>

</html>

1. ahora nos dirigimos a los archivos que heredan(index e info) ...

en index.html por ej. el código quedaría así:...

{% extends 'core/common.html' %}

{% block content %}

<center>

<h2>Bienvenido/a a mi sitio Web Django</h2>

<p>Hola, soy Demonscript y éste es mi primer site <br>

lanzado mediante el Poderoso Framework Django de Python</p>

</center>

{% endblock %}

1. ya vimos entonces una manera de separar nuestros códigos del fornt-end y de organizar nuestro contenido :) Realizamos el mismo procedimiento para el archivo info.html y creamos los archivos services.html y contact.html e implementamos en ellos el código para que hereden de common.html y su código respectivo que los hace singulares.

**block title:...**

es otro **template tag** de Django. vamos a añadir una nueva modificación para tener en este caso, títulos personalizados para cada una de las cuatro pginas.

Para ello:..

1. implementamos el tag en la etiqueta title de la plantilla common.html...

**<title>{% block title %}{% endblock %}|Demonscript!!..</title>**

1. y vamos ahora al archivo index.html que hereda de common.html y a parte del código que ya tiene, preferiblemente antes de ese código, implementamos...

**{% block title %}Inicio{% endblock %}**

1. al visitar la página mediante el navegador, podemos ver en el título de nuestra página(puntualmente en la pestaña del navegador):.. **Inicio|Demonscript!!..**

Podemos hacer el mismo procedimiento para el archivo info.html y personalizar el texto de su title, que en nuestro caso quedó con:.. **Info|Demonscript!!..** :)

**Generar enlaces dinámica-mente en nuestros html:...**

para ello hacemos uso del template tag **url** que nos permite hacer enlaces relativos tomados de los **nombres** que le damos a los enlaces en el archivo urls.py. Para ello vamos a sustituir el enlace de cada etiqueta **<a>** en la plantilla common.html por el tag url en el cual usamos el **name** que le dimos a nuestros link en cada **path** agregado, así:...

1. código modificado en common.html...

**<li><a href="{% url 'index' %}">Home</a></li>**

**<li><a href="{% url 'info' %}">Info del sitio</a></li>**

De esta manera, si cambiamos la ruta por ej, de info a about en el **path** de urls.py, la página funcionará porque estamos utilizando es el nombre que le damos, no la ruta propiamente...

**path('about/',views.info,name='info')**

De manera que, para nuestra comodidad futura, siempre utilizar template tag url y no utilizar elnaces escritos directamente en las plantilla html

**configurando el front-end con sus archivos estáticos(Archivos CSS, javascript, Bootstrap, jquery, etc):...**

Tomaremos como referencia el index y los demás archivos estáticos de “web-personal” del video-curso.

De manera que:...

1. Copiamos el directorio “web-personal” al directorio contenedor de nuestro proyecto “Demonscript”.
2. abrimos en Eclipse el archivo index.html de dicho directorio
3. hacemos una copia indexcopia.html
4. vamos a ir comparando nuestro archivo common.html con el indexcopia.html de “web-personal” y vamos reemplazando y homologando las partes donde deben ir los template tags. Por ej, en la etiqueta <title> del index hay que poner el template tag **block title**  para reemplazar el contenido estático.
5. en su menú de navegación implementamos el template tag **url** para cada link (en este caso son cuatro links)
6. el pie de página y la cabecera la editaremos después. Por ahora seleccionamos todo el código y lo pegamos en common.html reemplazando todo su código existente.
7. podemos cerrar indexcopia.html
8. podemos ver en el navegador algunos cambios. Ahora vamos a agregar los estilos Css, los códigos javascript, etc...
9. En nuestra app **core** creamos un directorio llamado **static** y dentro de este, creamos el respectivo directorio de nuestra app... el directorio **core**
10. Dentro de este subdirectorio **static/core** pegamos los directorios vendor, js, img y css que están en el directorio “web-personal”
11. regresamos al IDE y vamos a editar el archivo common.html de nuevo. Buscamos el comentario “**estilos y fuentes del template**” e inmediatamente debajo, escribimos la siguiente template tag...

{% load static %}

1. debajo de load static está el llamado a los archivos estáticos. Cambiamos el href por defecto de los archivos locales para implementar nuestro tag, así:...

**<link href="{% static 'core/vendor/bootstrap/css/bootstrap.min.css' %}" rel="stylesheet">**

**<link href="{% static 'core/vendor/font-awesome/css/font-awesome.min.css' %}" rel="stylesheet" type="text/css">**

**<link href="{% static 'core/css/clean-blog.min.css' %}" rel="stylesheet">**

Los otros link se dejan tal como vienen ya que son llamados a archivos servidos por Google, no son archivos locales

1. Más abajo hay un header con una imágen local de fondo, lo cambiamos de la sig, manera:...

**<header class="masthead" style="background-image: url('{% static 'core/img/home-bg.jpg' %}')">**

1. debajo del footer hay un llamado a archivos .js y bootstrap que también son locales. Deberán quedar así:...

**<script src="{% static 'core/vendor/jquery/jquery.min.js' %}"></script>**

**<script src="{% static 'core/vendor/bootstrap/js/bootstrap.bundle.min.js' %}"></script>**

**<script src="{% static 'core/js/clean-blog.min.js' %}"></script>**

1. Tenemos que volver a lanzar el servidor y ya podemos ver los cambios por medio del navegador Web

**Hagamos el mismo procedimiento para hacer una cabecera header dinámica:..**

se hace de una manera muy similar, utilizando un **tag template** para cada área que queremos dinamizar, queremos variar para cada página, la imágen de fondo, el título y el subtítulo de la cabecera...

1. vamos entonces mediante Eclipse a nuestro archivo common.html y nos hubicamos en la etiqueta **<header>**
2. del código...

<header class="masthead" style="background-image: url('{% static 'core/img/home-bg.jpg' %}')">

cortamos el contenido de la url, de manera que queda así...

**<header class="masthead" style="background-image: url('')">**

1. creamos entre esas comillas un template tag de tipo block y lo llamamos background...

<header class="masthead" style="background-image: url('**{% block background %}{% endblock %}**')">

1. ahora podemos dinamizar ese bloque desde cada archivo que hereda de common.html, por ej, hagamoslo desde index.html, en el que agregamos justo después del **block title** el siguiente bloque...

**{% block background %}{% load static %}{% static 'core/img/home-bg.jpg' %}{% endblock %}**

Notese que el código que está en azul es el código que cortamos de common.html ya que esa es la imágen que corresponde a la página inicial, y es esa imágen la que cambiaremos para cada página.

Algo **IMPORTANTE** es que, por alguna razón que desconozco, cuando se trata de generar imágenes mediante block y load static, el código no lo podemos ordenar(identar) porque al probar en el navegador NO carga la imagen deseada. Para solucionar esto, el código DEBE quedar en una sola línea

1. con esto, ya podemos cargar en el navegador el index.html y podemos ver su imagen respectiva, mientras que, si damos clic en el link **NOSOTROS,** se carga info.html pero sin imagen de fondo, igal si damos clic en cualquiera de los otros link.

Ahora hacemos el mismo procedimiento para los demás archivos, por ej, el archivo info quedaría con su block background así:...

**{% block background %}{% load static %}{% static 'core/img/about-bg.jpg' %}{% endblock %}**

Si vamos al navegador, actualizamos la página de portada y damos clic en el link nosotros, se carga el acrchivo info.html con una imagen de fondo distinta. Ahora realizamos el respectivo procedimineto a las otras dos páginas.

**Vamos a dinamizar el contenido de cada <h1> y de cada <spam> del header para cada página...**

1. en el archivo common.html y dentro del la etiqueta <div> que va a mostrar el texto en cada página, creamos un template tag llamado **headers**...

**<div class="site-heading">**

**{% block headers %}**

**{% endblock %}**

**</div>**

1. ahora vamos a cada archivo e implementamos ese block, por ej, en index.html, justo debajo del **block background** implementamos...

**{% block headers %}**

**<h1>Demonscript</h1>**

**<span class="subheading">Desarrollador Web</span>**

**{% endblock %}**

el código en azul es el código que implementamos para diferenciar a cada archivo. Así, el archivo info.html quedaría con su **block headers** así:..

**{% block headers %}**

**<h1>Acerca de</h1>**

**<span class="subheading">Biografía</span>**

**{% endblock %}**

modificamos de esta manera el block headers en los otros dos archivos service.html y contact.html

1. ahora damos dinamismo a la parte del contenido. Para ello abrimos cuanquiera de los archivos del diseño en el dir “web-personal”, abrimos por ej, content.html y tomamos su código perteneciente al <!— contenido principal —> y lo pegamos en el

<!— contenido principal —> de nuestro common.html

1. al comparar todos los archivos de “web-personal” podremos ver que el contenido de ese bloque de código cambia esepcificamente en el **<div>** más interno, de manera que ese será en nuestros archivos del dir **templates/core,**  el **block content**, así el <!—contenido principal —> en nuestro common.html quedará así

**<div class="container">**

**<div class="row">**

**{% block content %}{% endblock %}**

**</div>**

**</div>**

1. vamos a nuestro index.html de templates/core e implementamos su código singular dentro del **block content** y quedará así...

**{% block content %}**

**<div class="col-lg-8 col-md-10 mx-auto">**

**<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit. Saepe nostrum ullam eveniet pariatur voluptates odit, fuga atque ea nobis sit soluta odio, adipisci quas excepturi maxime quae totam ducimus consectetur?</p>**

**<br>**

**<p><b>Teléfono:</b> +09 876 543 210</p>**

**<p><b>Honorarios:</b> 60€/h (precio base)</p>**

**</div>**

**{% endblock %}**

En azul, el código singular del contenido principal para el index.html. Implementamos de la misma manera el contenido del block content para los demás archivos, cuyo código podemos copiar de los archivos de diseño que están en “web-personal”, proceso que es general excepto para info.html y services.html, porque a parte de etiquetas contenedoras de texto, generan imágenes por lo que debemos utilizar static, por ejemplo en. ...

**<img class="img-fluid avatar" src="{% static 'core/img/developer.jpg' %}" alt="">**

vamos a configurar un separador <hr> que tienen algunos archivos originales del sitio “web-personal” entre el contenido y el footer...

El archivo que no lleva este separador es en index. Con un condicional y una var **request.path** que captura la ruta del archivo, implementamos de manera dinamica el separador. Así:..

**{% if request.path != "/" %}**

**<hr>**

**{% endif %}**

De esa manera, todo archivo que no sea la raíz, tendrá el separador

1. Notese que en cada archivo individual que hereda de common.html en su content hay un class= “row” igual que en el common.html. Esto genera un error en los mágenes de services.html, al poner el navegador en tamaño pequeño el texto aparece pegado al borde, cosa poco estetica. Esto se debe a que bootstrap reconoce que hay una clase “row” dentrode otra clase “row”. Para solucionar esto, eliminamos el div class=”row” del common.html y lo ponemos en el content de cada archivo que lo requiera(info y contact)de manera que no queda un row dentro de otro row. Y con eso se soluciona el pequeño problema

**Administrar la sección de servicios(portafolio):..**

para esto, vamos a crear una app especificamente para manejar la sección se SERVICIOS desde un panel de administrador, haciendo que se conecte a una BD**.**

Recordemos que lo que gestiona el area de servicios es “proyectos” y los campos de un proyecto son...

* título
* descripción
* enlace
* una imágen

Para que Django pueda manipular imágenes, necesita de la instalación de un módulo externo, supongo que hay varios, pero en nuestro caso vamos a instalar el módulo **pilow** ...

estando en el directorio contenedor de nuestro proyecto

(Django2/Django\_project1) y con nuestro virtual environment activado, instalamos el módulo...

**pip3 install pillow**

ahora vamos a interactuar con el ORM(Mapeo Obeto-Relacional) de Django.

un **ORM** básicamente es una herramienta que permite convertir o formatear objetos de nuestra aplicación para que se puedan almacenar en una BD, de una manera abstracta al SGBD lo que nos ahorra el tener que escribir código SQL para los gestores de BD, permitiendo entre otras cosas, el poder cambiar de gestor de BD sin tener que preocuparnos de la sintaxis del nuevo SGBD (siempre que sea soportado por el framework)

siguiendo las pautas de Django en el MTV, podemos trabajar con ovbjetos mapeados en la BD así, al crear instancias de una clase especifica, dichas instancias y/o sus valores quedarán guardadas en la BD, así como sus posteriores modificaciones. En Django, las clases que manejan esos objetos persistentes son llamadas “MODELOS”.

Creamos entonces nuestra app services desde el directorio contenendor...

**python3 manage.py startapp services**

actualizamos nuestro proyecto y podemos ver que se ha creado nuestra nueva app **services** al mismo nivel de la app **core**

entre todo el contenido de la app **services** tenemos un archivo llamado **models.py** que es el script donde vamos a definir nuestros modelos que son las Clases enlazadas a la BD:..

de manera generica, un modelo se define implementando una **clase** con su resṕectivo **nombre** en singular y la cual hereda de la superclase **models.Model**

Dicha clase, va a representar una tabla de nuestra BD, tabla que representa sus registros mediante filas. Así, cada uno de los atributos de nuestra clase, será un campo de nuestro registro en la tabla. de manera que para registrar un nuevo proyecto en nuestro servicio o portafolio, el código correspondiente sería:...

**class New\_Project(models.Model):**

**title = models.CharField(max\_length = 80)**

**description = models.TextField()**

**image = models.ImageField()**

**created = models.DateTimeField(auto\_now\_add = True)**

**modified = models.DateTimeField(auto\_now = True)**

donde:

a cada atributo le indicamos su tipo, lo cual sirve para crear la tabla en la BD. el atributo **auto\_now \_add = true** indica al gestor BD que ese dato se ingresa únicamente la primera vez que se ingresa un nuevo proyecto y el atributo **auto\_now = true** se ingresa cada vez que haya una modificación de dicho registro.

cuando los datos van a ser de tipo charfield, debemos especificar su lon

ud o num. de caracteres que va a tener como máximo nuestro dato

ahora lo que tenemos que hacer es agregar al archivo **settings.py** de nuestro proyecto nuestra nueva app en la variable **INSTALLED\_APPS**

Ahora migramos nuestra app con los comandos **makemigrations** que sirve para indicarle a Django que hay cambios en algún nuevo modelo y así se crea un sistema de backup para retornar a un estado anterior de dicha app en caso de que se presente algún error y también ejecutamos el comando **migrate** el cual aplica la nueva migración a la BD. Ejecutamos entonces el primer comando en el directorio principal de nuestro proyecto...

**python3 manage.py makemigrations services**

Con el antrior comando se genera en nuestro poryecto django nuestro modelo **New\_Project**

ahora ejecutamos el segundo comando estando de nuevo sobre el dir. principal de nuestro proyecto para aplicar la nueva migración a la BD...

**python manage.py migrate services**

con lo anterior ya podemos ingresar nuevos registros de proyectos en la BD. Esto lo podemos hacer de dos maneras.

1. creando una vista en nuestro proyecto, para que procese un formulario que va a contener los campos correspondientes para registrar un nuevo proyecto. Este método es conveniente cuando trabajamos en entornos abiertos, donde se tenga que proporcionar formularios a los visitantes del sistema Web para que interactuen con la sección de proyectos
2. Utilizar el panel de administracion de Django el cual se encargará de todo. Este método cerrado es utilizado cuando solo algunos usuarios con exclusividad en el sistema y acceso mediante credenciales para que interactuen con la BD.

Por ahora utilizaremos el segundo método. El panel de Administración

**Otros tipos de Campos en Django y sus diferentes opciones**

como deciamos, vamos a utilizar el panel de andministrador de Django, la cual es una funcionalidad que viene por defecto. Y a la cual ingresamos, disparando el servidor e ingresando mediante el navegador a nuestro sitio/admin...

**http://127.0.0.1:8000/admin/**

podemos ver que se muestra un formulario de acceso con privilegios de administrador.

Si miramos el archivo urls.py del proyecto, podemos ver que la variable urlpatterns contiene los path que hemos agregado para nuestras páginas, y podemos ver que por defecto tiene un path que apunta hacia la página a la que acabamos de ingresar.

Como no tenemos un usuario creado para ingresar como administrador de nuestro proyecto, lo que vamos a hacer es crear un

de nuestro sistema Web...

paramos el servidor y ejecutamos el comando:...

python3 manage.py createsuperuser

ingresamos los datos para nuestro superusuario...

nombre, en este caso ingresé **Demonscript**

e-mal... **demonscript.io@gmail.com**

password... **developer**

si todo se realiza correctamente, debe mostrar el sig. mensaje...

**Superuser created successfully.**

si volvemos a lanzar el servidor e ingresamos a la página de administrador, ya tenemos las credenciales para ingresar.

podemos ver nuestra interfaz de inicio, la cual es first-mobile.

Podemos dar clic en **usuarios** donde, de momento, solo está el superusuario que acabamos de crear y al cual podemos acceder para ver su información

El resto de opciones las veremos más adelante. Ahora nos encargamos de activar el modelo de proyecto para poder crear nuevos proyectos en **services** y guardarlos en la BD..

Para que nuestro modelo aparezca en la interfaz del administrador, tenemos que registrar el modelo en el fichero **admin.py** de nuestra app **services.** De manera que abrimos el archivo y añadimos el siguiente código..

from services.models import New\_Project

Donde:..

**services.models** hace referencia al archivo **models.py** que pertenece a la app **services** y **New\_Project** es la clase que creamos en dicho archivo para ingresar nuevos proyectos

y ahora más abajo, puntualmente debajo de **# Register your models here.** Ingresamos el siguiente código

**admin.site.register(New\_Project)**

el anterior código registra un nuevo proyecto con el nombre New\_projects y habilita la visualización de modelos en la app services por medio de la página de adminstrador.

Guardamos los cambios y vamos a la pág. de inicio de Admin en el navegador, donde podemos ver que ya está habilidada la opción para ingresar nuevos proyectos.

Por ahora todo va muy bien, pero antes de ingresar nuevos proyectos mediante la interfaz de admin. Vamos a personalizar un proco su presentación, por ejemplo vamos a cambiar algunos apectos como el idioma y alguna configuración para poder subir imágenes sin problemas...

Para hacer pruebas, descargamos dos imágenes para guardarlas en nuestro proyecto (las nombré robot1 y robot2 ambas .jpg)

Nos disponemos a ingresar un nuevo proyecto mediante el navegador y la interfaz de admin..

podemos ver que al ingresar los datos correspondientes para **title**, **description** e **image** y presionar el botón **GRABAR**, aparentemente todo se ingresa correctamente, pero vemos que en la pág. principal de admin al dar clic en **services/New\_projects**, aparece como nombre de nuestro nuevo proyecto **New\_Project object (1)**, cuando debería aparecer **Desing Robot**

Otra falla a corregir es que al dar clic en Actualmente: **robots1.jpg** nos informa que la imagen no existe porque quizá fué eliminada cuando en realidad no es así.

El idioma se ve en espñol, cuando el resto del proyecto está config en español.

Todos esos detalles los vamos a resolver y mejorar:..

la primera modificación que vamos a realizar es cambiar en la Web, el nombre de la app a Español ya que aparece “Services”..

1. Lo primero que hacemos es abrir el archivo **apps.py** contenida en la app **services** Archivo en el cual vamos a agregar una variable auxiliar. Debajo de la var name ingresamos:..

**verbose\_name = ‘Servicios’**

1. Ahora abrimos el archivo **settings.py** de nuestro proyecto y en la variable **INSTALLED\_APPS** modificamos la var **‘services’** de manera que quedará así

**'services.apps.ServicesConfig'**

Donde:... **services** es el nombre de nuestra app, **apps** hace referencia al archivo apps.py de nuestra app services. Y **ServiceConfig** es el nombre de la Clase donde implementamos la var **verbose\_name**

Al ejecutar de nuevo la pág de amin en el navegador, poemos ver que ya está en español en nombre de “servicios”, porque lo acabamos de config con la var **verbose\_name**

Ahora vamos a traducir el nombre del modelo a español...

1. Nos dirigimos al archivo models.py de la app Services. Donde, dentro de la clase **New\_Project** creamos una subclase al final de su código, con el sig contenido...

**class Meta():**

**verbose\_name = "Mi Proyecto"**

**verbose\_name\_plural = "Mis Proyectos"**

**ordering = ["-created"]**

Como ya vimos, verbose\_name es el nombre que desamos mostrar en español, verbose\_name\_plural es el nombre en plural y ordering es una lista donde indicamos en que orden deseamos mostrar los proyectos ingresados. En este caso, los vamos a mostrar por orden de el más reciente al más antiguo. “created” si quisieramos mostrarlos del más antiguo al más reciente.

1. Ahora vamos a hacer que se muestre el nombre del proyecto que nosotros ingresamos originalmente, para esto, ingresamos una re-definición del método \_\_str\_\_ dentro de la clase **New\_Project** y justo después y al mismo nivel de la subclase **class Meta,** ingresamos una redefinición del método \_\_str\_\_ ...

**def \_\_str\_\_(self):**

**return self.title**

Con eso ya obtenemos en la web el nombre que está almacenado en la BD, por medio de la var **title**

1. También podemos cambiar a español los nombres en el formulario de ingreso de nuevos proyectos, añadiendo **verbose\_name** a cada var de nuestra clase **New\_Project**:...

**class New\_Project(models.Model):**

**title = models.CharField(max\_length = 80, verbose\_name = "Nombre de Proyecto")**

**description = models.TextField(verbose\_name = "Descripción")**

**image = models.ImageField(verbose\_name = "Imagen")**

**created = models.DateTimeField(auto\_now\_add = True,verbose\_name = "F. Creación")**

**modified = models.DateTimeField(auto\_now = True,verbose\_name = "F. Modificación")**

Ya podemos ver en el navegador el formulario con sus campos en español. Por defecto no se muestran los campos **created** y **modified.** por obvias razones Django bloquea la edición de esos campos después de haber creado un nuevo proyecto. Pero podemos mostrarlos en modo de “Solo lectura” para dar más información en pantalla...

Para esto, vamos a modificar la config. base del administrador. Abrimos el archivo admin.py de la app servivces e implementamos una clase para extender las funcionalidades del panel de administración.

**# Register your models here.**

**#La sig clase convencoinalmente se nombra ProjectAdmin**

**class Admin\_Projects(admin.ModelAdmin):**

**readonly\_fields = ('created','modified')**

hacemos que herede de la superclase **admin.ModelAdmin.**

**readonly\_fields** es una tupla que existe en Django y que hemos re-definido implementando en ella los campos de la BD que deseamos mostrar en solo lectura

1. Y en el código que registra el modelo, añadimos nuestra configuración extendida ...

**admin.site.register(New\_Project,Admin\_Projects)**

Solo nos queda hacer que se muestre la imágen que aún no se carga, como si no la hubieramos subido, previo a eso, configurar Django para que guarde las imágenes en un directorio adecuado y especificado por nosotros...

1. para ello, inicialmente creamos un directorio que yo llamé “multimedia”, en la raíz de nuestro proyecto y en el cual se guardarán todos los archivos subidos al sistema.

Django\_project1/multimedia

1. Ahora habrimos el acrchivo settinngs.py de nuestra app Django\_project1 y al final de todo el contenido ingresamos el sig código...

**#Media files...**

**MEDIA\_URL = '/multimedia/'**

**MEDIA\_ROOT = os.path.join(BASE\_DIR,"multimedia")**

Donde:.. multimedia representa el directorio que acabamos de crear. y en MEDIA\_ROOT especificamos que el directorio multimedia es el dir base para alamcenar archivos multimedia

1. abrimos nuestro modelo donde definimos nuestros proyectos (el archivo models.py de la app services) y a **image** le agregamos el atributo **upload\_to** :...

**image = models.ImageField(upload\_to = 'projects',verbose\_name = 'Imagen')**

Con ese añadido le especificamos a Django, en qué directorio debe guardar las imágenes de cada proyecto añadido y/o modificado

Al guardar cambios y ejecutar la operación de subir imágen para los dos proyectos ya ingresados, podemos ver en el IDE que se ha creado de manera automática el dir “prjoects” y en él se han guardado correctamente las imágenes que subimos.

Aunque si intentamos ver mediante firefox la imagen de nuestro proyecto, aún no se visualiza y lo que genera es unos mensajes de error. Y esto se debe a que por defecto, el servidor de Django no puede servir dichos ficheros, Vamos a corregir esto de manera TEMPORAL y teniendo el modo DEBUD en activo(recordemos que esto se hace únicamente en Desarrollo, no en etapa de producción):...

1. nos dirigimos al archivo urls.py de nuestro proyecto y editamos su código de la sig manera:..

**from django.contrib import admin**

**from django.urls import path**

**from core import views**

**from django.conf import settings**

**urlpatterns = [**

**path('contact/',views.contact,name = 'contacto'),**

**path('services',views.services,name = 'servicios'),**

**path('',views.index,name = 'inicio'),**

**path('info/',views.info,name = 'informacion'),**

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

**]**

**if settings.DEBUG:**

**from django.conf.urls.static import static**

**urlpatterns += static(settings.MEDIA\_URL, document\_root=settings.MEDIA\_ROOT)**

En azul el código añadido.

importando settings con **from django.conf import settings** podemos acceder a las variables MEDIA\_URL y MEDIA\_ROOT y en el condicional estamos preguntando si DEBUG == true. y como en nuestro caso si, importamos statics **from django.conf.urls.static import static** que nos permite servir ficheros estáticos. y finalmente añadimos al urlpattens que si el usuario quiere cargar archivos estáticos desde el dir “multimedia”, se lo permita

**urlpatterns += static(settings.MEDIA.....**

Con esto ya podemos actualizar el navegador y visualizar la imagen sin problemas.

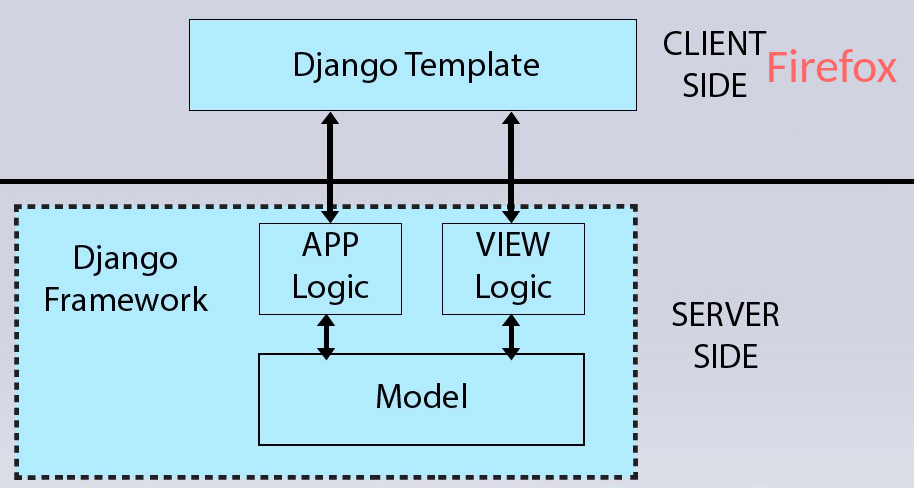
**Optimizando nuestro site para el MTV**:...

Como ya habíamos comentado al inicio de nuestro estudio. Django hace una versión del MVC y lo ha llamado **Model-Template-View**

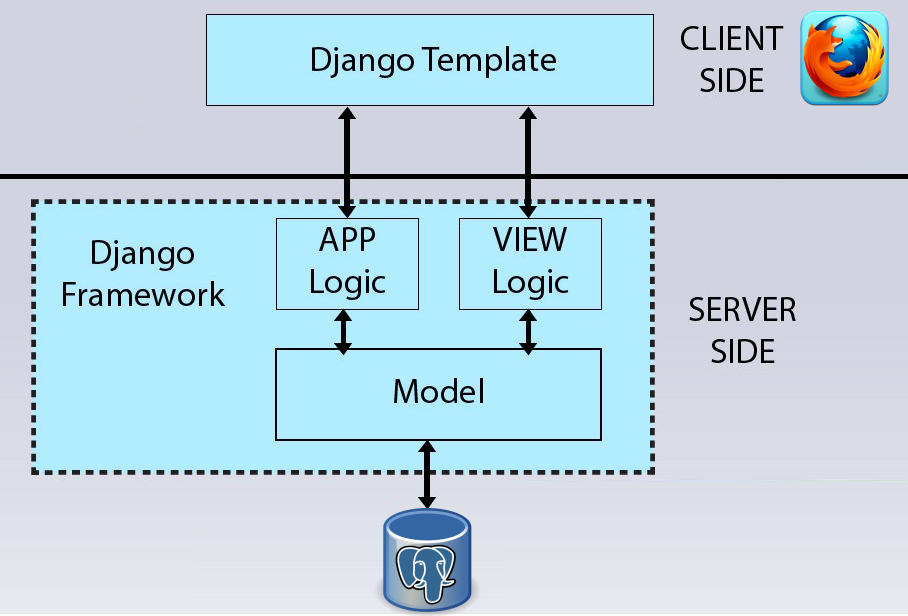
Para finalizar la primera parte de nuestro estudio, vamos a optimizar el código de nuestro proyecto para que funcione con la BD acorde con el MTV:..

para ello, vamos a trabajar sobre el template que muestra los proyectos(services.html), pero que de monento los muestra de manera estática, cuando nuestro interés es que muestre los proyectos que ingresamos en la BD desde el panel de administración:

Hasta ahora, lo que en práctica se puede hacer desde la interfáz gráfica es lanzar una petición HTTP mediante el navegador, inmediatamente el programa llama a la vista que corresponde a dicha solicitud, posteriormente se renderiza el template para mostrar el HTML resultante mediante el Navegador. Es un sitio estático con algún código dinámico que NO se genera desde la DB...



Ahora vamos a implementar en nuestro sitio lo que a grandes rasgos se muestra en el sig diagrama....



De manera que el proyecto responderá con datos de la BD cuando se requiera (en el caso de posicionarnos en services.html porque los proyectos los almacenamos en la BD).

Para empezar a interactuar con datos de nuestra BD, Lo que tenemos que hacer es:..

1. Sabemos que los proyectos se deben mostrar mediante el template **services.html**, entonces para organizar de una mejor manera nuestro código, vamos a definir la vista **services** que de momento está en la app core, y que por obvias razones debe estar en la app services. así, el sig código que define la vista para **services** y que está en **views.py** de core lo cortamos y lo pegamos en **views.py** de la ap services:..

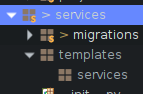
**def services(request):**

**return render(request, "core/services.html")**

Ahora tenemos que mover el template services.html a su correspondiente app...

1. creamos un directorio dentro de la app services llamado **templates** y como en el caso de core, dentro de él creamos un directorio con el nombre de la app(**services**)...

Esto, para que esta app services tenga sus propios templates. Ahora nos dirigimos mediante el IDE a core/templates/core y cortamos el template services.html y lo pegamos en services/template/services



1. ahora modificamos el código de la vista services de la sig manera...

**def services(request):**

**return render(request, "services/services.html")**

Esto porque su tuta ha cambiado, ya no pertenece a la app **core**, sino a **services.**

1. vamos a modificar urls.py de nuestro proyecto para readaptar la URL que ahra debe ir a buscar la vista services en la app **service**s y no en core
2. resulta que Django genera error si llamamos a dos views sin importar si son de apps diferentes...

from core import views

from services import views

Lo anterior genera error. PEro como necesitamos llamar a vistas de las dos Apps (core y services), tenemos que recurrir a la palabra reservada **as** para asignar un nombre único a cada importación...

from core import views as core\_views

from services import views as services\_views

y Más abajo tenemos que modificar los llamados en la var **urlpatterns...**

**urlpatterns = [**

**path('contact/',core\_views.contact,name = 'contacto'),**

**path('services',services\_views.services,name = 'servicios'),**

**path('',core\_views.index,name = 'inicio'),**

**path('info/',core\_views.info,name = 'informacion'),**

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

**]**

Hay otras formas de gestionar o resolver el inconveniente anterior, cosa que veremos más adelante. Ahora ejecutamos el código y vemos que funciona.

El obejtivo de la acción anterior es principalmente el de organizar nuestro código para que el proyecto tenga realmente escalabilidad, y de esta manera no tenemos qu estar importando objetos de una app a otra, cosa que es poco recomendable

Nos queda configurar nuestro proyecto para que muestre desde la BD y mediante la app servies, los proyectos mediante el template services.html **...**

1. en el archivo views.py de la app services, importamos el contenido de models.py de la misma app añadiendo el sig código...

**from services.models import New\_Project**

ahora vamos a obtener la lista de todos los proyectos almacenados en BD...

1. más abajo, Dentro de la vista **services** del archivo views.py agregamos un objeto antes del **return** y añadimos un tercer parámetro en el render que estamos retornando...

**def services(request):**

**projects = New\_Project.objects.all()**

**return render(request, "services/services.html",{'projects':projects})**

con la var-objeto **projects** estamos alamcenando los objetos que obtenemos de la BD(los proyectos almacenados). y con el “diccionario de contexto” que estamos pasando como tercer parámetro, asignamos a la clave-valor el objeto **projects** que acabamos de crear.

ahora vamos al template services.html y dentro del tag **block content** implementamos el sig código para mostrar el objeto **projects** que estámos enviando desde views.py

**{{projects}}**

Guardamos cambios y vamos al navegador web para visitar la página services.html, al actualizar la pág, podemos ver que se muestra una lista de tipo QuerySet(el objeto) con los nombres de los dos proyectos creados en BD, recordemos que esos nombre se muestran gracias al código que implementamos al final del modelo en el archivo models.py

**def \_\_str\_\_(self):**

**return self.title**

1. ahora que podemos ver que si se genera el objeto. vamos a mostrar los datos, recorriendo el objeto(la lista de tipo QuerySet) mediante un bucle, y para ello, vamos a hacer uso del template-tag **for**...

Para ello, lo primero que hacemos es borrar el código **{{projects}}** que acabamos de ingresar dentro del **block content** para mostrar el objeto. Acto seguido, implementamos el **block for** y utilizamos el código html para generar de manera dinámica, nuestros proyectos almacenados en la BD...

**{% block content %}**

**{%for project in projects %}**

**<!-- Proyecto -->**

**<div class="row project">**

**<div class="col-lg-3 col-md-4 offset-lg-1">**

**<img class="img-fluid" src="{{project.image.url}}" alt="">**

**</div>**

**<div class="col-lg-7 col-md-8">**

**<h2 class="section-heading title">{{project.title}}</h2>**

**<p>{{project.description}}</p>**

**<!-- <p><a href="http://google.com">Más información</a></p> -->**

**</div>**

**</div>**

**{% endfor %}**

**{% endblock %}**

Podemos ver que utilizamos los nombres de los campos en la BD como si fueran atributos del objeto projects(el cual estámos recorriendo con el bucle y alamacenando en cada iteración su contenido en **project**)

NOTESE que para mostrar la imagen, tenemos que recurrir a un añadido **project.image.url** Y esto sucede porque al dejar solamente **project.image**, el navegador va a buscar la imágen en

**<img class="img-fluid" src="**[**projects/robot2.jpg**](view-source:http://127.0.0.1:8000/projects/robot2.jpg)**" alt="">**

lo que genera un error.

Y sabemos que la imagen realmente está en “**multimedia/projects/robot2.jpg**”

ya con el añadido .url Django genera la ubicación correcta para el navegador

Con esto nos acercamos al fial de nuestro primer proyecto Django

1. Para finalizar, podemos observar que en la anterior modificación, comentamos el apartado de “más información” porque en el modelo **New\_Project** de nuestro archivo models.py de la app services, no implementamos ese campo. de manera que lo vamos a añadir...

**class New\_Project(models.Model):**

**title = models.CharField(max\_length = 80, verbose\_name = 'Nombre del Proyecto')**

**description = models.TextField(verbose\_name = 'Descripción')**

**more\_info = models.URLField(null = True, blank = True,verbose\_name='Sitio Web')**

**image = models.ImageField(upload\_to = 'projects',verbose\_name = 'Imagen')**

**created = models.DateTimeField(auto\_now\_add = True, verbose\_name = 'F. Creación')**

**modified = models.DateTimeField(auto\_now = True, verbose\_name = 'F. Modificación')**

**class Meta():**

**verbose\_name = "Mi Proyecto"**

**verbose\_name\_plural = "Mis Proyectos"**

**ordering = ["-created"]**

**def \_\_str\_\_(self):**

**return self.title**

En azul el nuevo código añadido, donde **null = true** y **blank = true** especifican que el campo es opcional.

Guardamos cambios y en la terminal registramos y después ejecutamos la migración...

**python3 manage.py makemigrations**

**python3 manage.py migrate**

para hacerlo más sencillo y como los cambios solo ha sido realizados en la app services, podemos ejecutar el proceso unicamente para dicha app...

**python3 manage.py makemigrations services**

**python3 manage.py migrate services**

**Recordemos que:..**

**makemigrations** -> Encuentra los cambios realizados a los modelos

**migrate** -> Aplica esos cambios

1. si nos dirigimos al navegador, en el panel de admin podemos ver que se habilita la opción para ingresar el sitio Web para cada proyecto. Como prueba, ingresamos un sitio web de ejemplo para uno de los proyectos.
2. ahora modificamos el código del template **services.html** donde dentro del template-tag **for** donde está comentado el link del proyecto, implementamos...

**{% if project.more\_info %}**

**<p><a href="{{project.more\_info}}">Más información</a></p>**

**{% endif %}**

Con el anterior código verificamos si el campo “more\_info” de la BD está en blanco, y en caso de que no esté en blanco, muestra el respectivo link.

Guardamos cambios y actualizamos la página services.html donde podemos ver el resultado

Hemos llegado entonces al final de la primera parte de nuestra preparación.

Pudimos instalar y crear entornos virtuales, pudimos instalar y crear proyectos Django, implementamos en un proyecto, archivos estáticos, creamos vistas, configuramos algúnos apartados importantes del proyecto, y después creamos apps con sus respectivas vistas, implementamos datos en la BD que tiene Django por defecto, mostramos datos desde la BD por medio de nuestras vistas.

Y subimos versiones de nuestro proyecto a GitHub por medio del programa GitKraken.

**{{ PARTE 2 }} (Nivel Medio).**..

Creación del segundo proyecto Django, en el cual reviviremos prácticas de la PARTE 1 de nuestra preparación...

crear proyecto, configurarlo, implementar Apps y sus respectivos modelos y Vistas, ingresaremos datos en la BD desde Django mediante Firefox, y de nuevo configuraremos Django para poder mostrar dichos datos.

Pero ahora lo vamos a hacer a un nivel más complejo, utilizando e implementando tablas relacionales(foreign Key)

y creando más numeros de apps que en el proyecto anterior y personalizando en mas detalle el panel de admin, crearemos tipos de usuarios con permisos diferentes a los de admin,.

Nuestro proyecto dará vida a un CoffeeShop, sus servicios, variedades de cannabis, contacto, etc...

1. Creamos nuestro repositorio en GitHub mediante GitKraken con el nombre de CoffeeShop. Posteriormente cerramos GitKraken
2. Ahora nos dirigimos al IDE Eclipse, donde creamos un nuevo proyecto en Django mediante la interfaz Gráfica. Proyecto al cual le damos el nombre “**Coffee\_Shop**” y su hubicación será el directorio Developer/Django2/Coffee\_Shop que se creó al lanzar el repositorio con GitKraken.
3. posicionados en nuestro dir contenedor y mediante la terminal, ejecutamos el comando

python3 manage.py migrate

Vemos que se realizaron las migraciones hacia la BD y todo está funcionando correctamente

1. ahora vamos a crear la app **core**...

sobre el dir de nuestro proyecto en la terminal, ejecutamos:..

**python3 manage.py startapp core**

1. actualizamos el proyecto en la interfaz gráfica del IDE. Creamos ahora las vistas iniciales de nuestras páginas web en el archivo views.py de la app core...

**from django.shortcuts import render,HttpResponse**

**# Create your views here.**

**def index(requets):**

**return HttpResponse("<h1>Home</h1>")**

**def about(request):**

**return HttpResponse("<h1>Nosotros</h1>")**

**def services(request):**

**return HttpResponse("<h1>Servicios</h1>")**

**def store(request):**

**return HttpResponse("<h1>nuestra Tienda</h1>")**

**def conctac(request):**

**return HttpResponse("<h1>Contacto</h1>")**

**def blog(request):**

**return HttpResponse("<h1>Visita nnuestro Blog informativo</h1>")**

**def samples(request):**

**return HttpResponse("<h1>Pruebas</h1>")**

ahora registramos las url respectivas para cada vista en el archivo urls.py del proyecto para mostrarlas mediante Firefox...

1. añadimos primeramente el import correspondiente para llamar a las vistas...

**from core import views as core\_views**

1. ahora buscamos la var **urlpatterns** para añadir las rutas. Demanera que quedó así...

**urlpatterns = [**

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

**path('',core\_views.index,name = 'index'),**

**path('about/',core\_views.about,name = 'nosotros'),**

**path('services/',core\_views.services,name = 'servicios'),**

**path('store/',core\_views.store,name = 'tienda'),**

**path('contact/',core\_views.conctac,name = 'contacto'),**

**path('infoblog/',core\_views.blog,name = 'blog'),**

**path('sample/',core\_views.samples,name = 'prueba'),**

**]**

Con lo anterior podemos guardar cambios y ejecutar el proyecto.

La prueba la podemos hacer en el navegador con cualquiera de las url. por ej....

http://127.0.0.1:8000/infoblog/

y el resultado obtenido será...

**Visita nnuestro Blog informativo**

1. Ahora vamos a implementar una Nueva manera de configurar las URL’s Y es epecificandolas en cada una de las apps creadas. Por ejemplo vamos a implementar las urls de core en esa misma app y las referenciamos después desde el archivo urls.py del proyecto....
2. inicialmente, creamos un archivo urls.py en la app core y pegamos el sig código que cortamos parcialmente de urls.py que está en el proyecto general...

**from django.urls import path**

**from core import views as core\_views**

**#create here the Views**

**urlpatterns = [**

**path('',core\_views.index,name = 'index'),**

**path('about/',core\_views.about,name = 'nosotros'),**

**path('services/',core\_views.services,name = 'servicios'),**

**path('store/',core\_views.store,name = 'tienda'),**

**path('contact/',core\_views.conctac,name = 'contacto'),**

**path('infoblog/',core\_views.blog,name = 'blog'),**

**path('sample/',core\_views.samples,name = 'prueba'),**

**]**

10. Ahora modificamos el código de urls.py del proeyecto gen y lo dejamos así...

**from django.contrib import admin**

**from django.urls import path,include**

**urlpatterns = [**

**#path from core**

**path('', include('core.urls')),**

**#path from admin**

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

**]**

Si analizamos en detalle el código actual comparado con el código anterior, podemos ver que añadimos una nueva importación **“include”** el cual nos permite incluír código de otro archivo, que en este caso es del archivo urls.py de la app core.

Con lo anterior guardamos los cambios y podemos ejecutar de nuevo el proyecto en firefox.

**acoplar el maquetado HTML y los archivos estáticos al proyecto...**

11. Para ello, copiamos temporalmente el directorio que contiene el maquetado de gía(webempresarial-frontend) al directorio base del proyecto.

12. implementando la lógica de gestión de templates en Django, Creamos el directorio **templates** en la app core y dentro del nuevo directorio creamos un nuevo directorio llamado **core.**

regresamos a la app core y creamos el directorio **static** y dentro de este, creamos el directorio **core**

13. Ahora copiamos los 7 templates(about.html, infoblogblog.html, contact.html,etc) del directorio webempresarial-frontend en el directorio **templates/core** de la app core.

14. Acto seguido. Hacemos lo mismo con los directorios, **img** y **vendor** del directorio webempresarial-frontend, los copiamos en el directorio **static/core** de la app core

15. para que Django carque los archivos anteriores que hemos agregado, hay que activar la app core. De manera que lo registramos en la var INSTALLED\_APPS del archivo settings.py perteneciente al proyecto general.

16. Lo siquiente es modificar el archivo views.py de la app core para poder cargar cada archivo en firefox...

**# Create your views here.**

**def index(request):**

**return render(request, 'core/index.html')**

**def about(request):**

**return render(request, 'core/about.html')**

**def services(request):**

**return render(request,'core/blog.html')**

**def store(request):**

**return render(request,'core/store.html')**

**def conctac(request):**

**return render(request,'core/contact.html')**

**def blog(request):**

**return render(request,'core/infoblog.html')**

**def samples(request):**

**return render(request,'core/sample.html')**

podemos ir a firefox y ensayar direcciones como:..

http://127.0.0.1:8000/store/

http://127.0.0.1:8000/infoblog/

cunciona correctamente. Pero el menú de navegación genera error y los archivos estáticos aún no se cargan. Vamos a ello....

17. los ficheros estáticos los configuramos de la sig manera...

inicialmente en el fichero index.html ingresamos el tag template **load static** justo después de la etiqueta **title**...

**{% load static%}**

y ahora modificamos cada etiqueta html que llama a los arcihvos estáticos de nuestro proyecto. Por ejemplo

**<link href="{% static 'core/vendor/bootstrap/css/bootstrap.min.css' %}" rel="stylesheet">**

en lo referente a links, recordemos que utilizamos lo almacenado en la var **name** del **path** de cada link en **urlpatterns** del **archivo urls.py** de la app ...

**href="{% url 'servicios' %}**

en lo referente a imágenes ...

**src="{% static 'core/img/intro.jpg' %}"**

y así con los demás etiquetas de index.html que utilizan archivos que están en el directorio static de la app core

personalizo el sitio con imágenes para mi proyecto

ahora vamos a crear a partir del código del index.html el archivo common.html del cual heredan todos los demás templates...

18. buscamos entonces en el common.html las partes comunes que van a salir en todos las plantillas. como el menú de navegación, la cabecera, el footer

Borramos todo el código que pertenece a **cabecera** y **mensaje** del common.html e implementamos en su lugar el tag-template “block content”

**{% block content %}**

**{% endblock %}**

guardamos los cambios y vamos a implementar la herencia, primeramente en el index.html...

19. archivo del cual buscamos la cabecera y el mensaje. borramos Todo lo demás ya que ese código borrado es el que se comparte desde common.html por medio de la herencia..

20. implementamos en el inicio de index.html la herencia y el bloque de contenido...

**{% extends ‘core/common.html’ %}**

**{% load static %}**

**{% block content %}**

**{% endblock %}**

Recordemos que necesitamos cargar una imágen personalizada en el index, razón por la cual implementamos tambien **load static**

21. el código html generado directamente desde el index(**cabecera**, **mensaje**) lo ponemos dentro del template-tag **block content**

ejecutamos el index en Firefox para verificar que todo funciona correctamente.

Terminado el proceso anterior, hacemos lo que coresponde a las demás plantillas, de la misma manera que lo hicimos para el index...

Ya tenemos implementado todo el contenido estático (HTML,CSS, etc) en todas las plantillas.

22. Ahora vamos a poner en acción la el admin de nuestro proyecto. En primera instancia creamos el superusuario con el comando ....

**python3 manage.py createsuperuser**

Y en los datos requeridos, ingresamos en este caso los mismos datos del anterior proyecto...

nombre de usuario: **Demonscript**

correo electrónico: **demonscript.io@gmail.com**

password: **developer**

Resulta que me ha tomado las modificaciones iniciales del anterior proyecto(por ejemplo la interfaz está perfectamente en Español, ya veremos si al hacer la migración para ingresar nuevos servicios a la BD sigue en español o no)

23. Vamos a configurar el proyecto para almacenar los ficheros multimedia ingresados desde el navegador...

Creamos inicialmente el directorio “**multimedia**” en el dirrectorio contenedor de nuestro proyecto.

24. Ahora vamos al archivo **settings.py** de nuestro proyecto para configurar la carga y almacenamiento de archivos multimedia. Al final del su código ingresamos...

**MEDIA\_URL = '/multimedia/'**

**MEDIA\_ROOT = os.path.join(BASE\_DIR,'multimedia')**

recordemos que el primero especifica el directorio que acabamos de crear

Y el segundo especifica el directorio base para gestionar archivos multimedia.

25 Nos dirigimos ahora al archivo urls.py donde en la sección de imports agregamos...

**from django.conf import settings**

y al final del código, implementamos...

**if settings.DEBUG:**

**from django.conf.urls.static import static**

**urlpatterns += static(settings.MEDIA\_URL,document\_root=settings.MEDIA\_ROOT)**

recordemos que el anterior código le da aurotización al usuario para cargar-visualizar archivos multimedia mediante el navegador.

Ya tenemos activado el servicio de archivos multimedia. Ahora vamos a crear la App mediante la cual vamos a subir y mostrar los primeros archivos multimedia...

26. Creamos la app “services”...

**python3 manage.py startapp services**

Actualizamos nuestro proyecto en la interfáz gráfica y podemos ver nuestra nueva app creada.

27. registramos la nueva app en la var INSTALLED\_APPS del archivo settings.py

28. Vamos a crear nuestro modelo de **servicios**. Para ello vamos al archivo **models.py** de la app **services** y en el apartado “models here”, implementamos el respectivo código...

**class New\_service(models.Model):**

**title = models.CharField(max\_length = 100,verbose\_name = 'Título')**

**subtitle = models.CharField(max\_length = 100,verbose\_name = 'Subtítulo')**

**description = models.TextField(verbose\_name = 'Descripción')**

**image = models.ImageField(verbose\_name = 'Imágen',upload\_to = 'services')**

**created = models.DateTimeField(auto\_now\_add = True, verbose\_name = 'F. Creación')**

**modified = models.DateTimeField(auto\_now = True,verbose\_name = 'F. Modificación')**

**class Meta:**

**verbose\_name = "Servicio"**

**verbose\_name\_plural = "Servicios"**

**ordering = ['-created']**

**def \_\_str\_\_(self):**

**return self.title**

Reapasando el código anterior, implementamos los campos escenciales para cada servicio ingresado, auto\_now\_add se utiliza porque es para registrar por primera y única vez, mientras que auto\_now es para registrar la fecha y hora de cada nueva actualización del servicio.

La “class Meta”, evidentemente la implementamos para colaborar con la visualización en español del formulario y la manera u orden en la que se van a mostar los servicios ya ingresados.

29. Ahora hacemos la migración, para que se creen en la BD la tabla y sus campos respectivos...

**python3 manage.py makemigrations**

**python3 manage.py migrate**

Recordemos que el primer comando es de utilidad para que Django registre los nuevos modelos creados, de manera que se pueda hacer versionamiento del proyecto

y el segundo comando se encarga de migrar los nuevos modelos a la BD

30. configuramos Django para que nuestro modelo “New\_service” sea accesible desde el panel de administración...

Para ello abrimos el archivo admin.py de la app services y añadimos...

**from services.models import New\_Service**

**# Register your models here.**

**class Admin\_Services(admin.ModelAdmin):**

**readonly\_fields = ('created','modified')**

**admin.site.register(New\_Service,Admin\_Services)**

con **admin.site.register** registramos la app para el apartado de admin, le pasamos el servicio **New\_service** y la configuración que le añadimos **Admin\_Services**

31. procedemos a ejecutar el servidor e ingresamos a la interfaz de admin. Podemos ver que ya está habilitada la app de servicios.

Notese que el idioma español ya está implementado, vamos a traducir el nombre de la app en el panel(está de momento como SERVICES)...

32. ingresamos al archivo **apps.py** de la app **services** y dentro de su clase llamada **ServicesConfig** agregamos un verbose\_name...

**class ServicesConfig(AppConfig):**

**name = 'services'**

**verbose\_name = 'Administrador de Servicios'**

En azul, el código añadido

33. Para aplicar esta configuración, vamos al archigo **settings.py** del proyecto general y en la variable **INSTALLED\_APPS** cambiamos ‘services’ por **‘services.apps.ServicesConfig’**

guardamos los cambios y verificamos su funcionamiento mediante Firefox.

34. trasladamos el template **services.html** a un nuevo directorio **templates/services** que vamos a crear el la app services y modificamos su ruta de core a services...

**return render(request,'services/services.html')**

35. creamos un arcuivo urls.py en la app services y le implementamos el código...

**from django.urls import path**

**from services import views as services\_views**

**#create here the Views**

**urlpatterns = [**

**path('',services\_views.services,name = 'servicios'),**

**]**

36. borramos el **path** correspondiente a **services** de **urls.py** en la app core

37. Ahora nos dirigimos a urls.py del proyecto general y añadimos los paths de services...

**#path from services**

**path('services',include('services.urls')),**

reiniciamos el servidor y verificamos que funciona el link de services

38. ahora vamos a añdir a la BD los servicios que tenemos de manera estática en el template services.html

podemos ver que el directorio **Coffee\_Shop/multimedia/services** tiene ahora las imágenes que añadimos mediante el panel de administración, con lo que, podemos borrar dichas imágenes de **Coffee\_Shop/core/statics/core/img** que estamos utilizando para mostrar de manera estática pues quedarán sin uso alguno

39. modificamos el código de la vista services para capturar los servicios de la BD amanera de Objetos...

**from django.shortcuts import render**

**from services.models import New\_Service**

**# Create your views here.**

**def services(request):**

**allservices = New\_Service.objects.all()**

**return render(request,'services/services.html',{'services':allservices})**

En azul, el código añadido.

Vamos a modificar el código html del template services.html para mostrar su contenido de forma dinámica desde la BD....

40 Borramos los **section** que implementamos para mostrar temporalmente los servicios. Dejamos un solo section que vamos a utilizar en nuestro bucle...

**{% for services in services %}**

**{% endfor %}**

dentro del bucle ponemos todo a la etiqueta section y su contenido, donde, para mostrar por ejemplo el title de la BD..

**<span class="section-heading-upper">{{services.title}}</span>**

para mostrar el subtitulo..

**<span class="section-heading-lower">{{services.subtitle}}</span>**

Y así mismo para el campo description.

para mostrar la imagen agregamos el método **url** en el atributo src..

**src="{{services.image.url}}"**

guardamos los cambios y podemos verificar el funcionamiento en el navegador.

Vemos que se muestra como lo configuramos, del último servicio ingresado al primer servicio ingresado en la BD. Si queremos mostrarlos al contrario, podemos añadir al tag-template for, **reversed,** que mostrará todo pero en el orden contrario...

{% for services in services reversed %}

Con ello damos por terminada la app **services**

Ahora vamos a oimplementar una nueva appen nuestro proyecto Coffee\_Shop..

41. Vamos a implementar la app **infoblog** para que se pueda ingresar noticias o artículos informativos y de paso vemos un nuevo tema donde podemos relacionar modelos mediante el potente sistema ORM de Django. veremos la relación foranea y many to many, donde la primera nos permite relacionar instancias de modelos diferentes y la segunda nos permite relacionar instancias de un mismo modelo. Nos dirigimos entonces al directorio contenedor de nuestro proyecto y lanzamos el comando ...

**python3 manage.py startapp infoblog**

42. vamos al archivo models de la app infoblog y crearemos dos modelos...

en primera instancia importamos timezone y User que nos sirben para obtener fecha-hora y la segunda para jugar con los usuarios del sistema.

**ya veremos cual es la diferencia entre usar now\_add y timezone** **now**

**from django.utils.timezone import now**

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

43. a continuación creamos dos modelos, uno para las categorías de las noticias o publicaciones y otro para el contenido de la noticia...

**class Category(models.Model):**

**name\_cat = models.CharField(max\_length = 100,verbose\_name ='Nombre')**

**created = models.DateTimeField(auto\_now\_add = True, verbose\_name = 'F. Creación')**

**modified = models.DateTimeField(auto\_now = True,verbose\_name = 'F. Modificación')**

**class Meta:**

**verbose\_name = "Categoría"**

**verbose\_name\_plural = "Categorías"**

**ordering = ['-created']**

**def \_\_str\_\_(self):**

**return self.name\_cat**

**class Post(models.Model):**

**title = models.CharField(max\_length = 100, verbose\_name = 'Título')**

**content = models.TextField(verbose\_name = 'Contenido')**

**categories = models.ManyToManyField(Category,verbose\_name = 'Categorías')**

**creation\_date = models.DateTimeField(verbose\_name = 'F. elaboración', default = now)**

**image = models.ImageField(verbose\_name = 'Imagen', blank = True, null = True, upload\_to ='infoblog')**

**author = models.ForeignKey(User,on\_delete = models.CASCADE, verbose\_name = 'Autor')**

**created = models.DateTimeField(auto\_now\_add = True, verbose\_name = 'F. Creación')**

**modified = models.DateTimeField(auto\_now = True,verbose\_name = 'F. Modificación')**

**class Meta:**

**verbose\_name = "Publicación"**

**verbose\_name\_plural = "Publicaciones"**

**ordering = ['-created']**

**def \_\_str\_\_(self):**

**return self.title**

en el modelo **post** implementamos los dos tipos de relación....

**categories = models.ManyToManyField...**

la cual hace referencia a un modelo del mismo archivo

**author = models.ForeignKey**...

que hace referencia a un modelo de otra app, en este caso por medio del import... **from django.contrib.auth.models import User**

44. registramos la app onfoblog en el archivo settings.py del proyecto...

**INSTALLED\_APPS = [**

**'django.contrib.admin',**

**'django.contrib.auth',**

**'django.contrib.contenttypes',**

**'django.contrib.sessions',**

**'django.contrib.messages',**

**'django.contrib.staticfiles',**

**'core',**

**'infoblog',**

**'services.apps.ServicesConfig',**

**]**

45. y aplicamos las migraciones correspondientes..

**Python3 manage.py makemigrations** para aplicar los cambios en Django y ayudar al control de versiones

**python3 manage.py migrate** para aplicar los cambios en la BD

46. Ahora vamos a generar la parte administrativa de esta app en su archivo **admin.py,** en el cual añadimos...

**from infoblog.models import Category,Post**

**# Register your models here.**

**class AdminCat(admin.ModelAdmin):**

**readonly\_fields = ('created','modified')**

**class AdminPost(admin.ModelAdmin):**

**readonly\_fields = ('created','modified')**

**admin.site.register(Category,AdminCat)**

**admin.site.register(Post,AdminPost)**

47. guardamos los cambios y ejecutamos el servidor para ir al admin e ingresar Categorías e infoblogs, de manera que ingresamos las categorías **General**, **Ocio** y **Salud**

y en cuanto a post, vamos a crear 4, a saber:...

**Vencer El Insomnio** categoría.. **Salud**

**Cannabis para la Migraña** categoría.. **Salud**

**El aceite de CBD** categoría.. **Salud**

**Maneras de consumir marihuana** categoría.. **Ocio**

**Ofertas de Primavera** categoría.. **General**

Es curiosos que en categorías yo puedo seleccionar una o más en el momento de crear un nuevo post.

Podemos ver la manera sencilla y poderosa en que django hace relaciones entre modelos.

**Vamos a personalizar un poco más el Administrador de Django**:...

1. nos dirigimos al archivo admin.py de la app infoblog
2. y en el modelo adminPost añadimos...

**class AdminPost(admin.ModelAdmin):**

**readonly\_fields = ('created','modified')**

**list\_display = ('title','author', 'creation\_date')**

En azul nuestro código añadido. Con esto, guardamos los cambios y regargamos la página para ver un mejor resultado en la administración de publicaciones

1. si añadimos **odering(‘author’,’created’)** en el modelo **AdminPost** nos ordena de una mejor manera los resultados
2. Implementemos yn formulario de búsqueda. Añadimos el sig código al modelo **AdminPost**...

**search\_fields = ('author\_\_username','title','categories\_\_name\_cat')**

Notese que al dar busqueda en campos relacionales debemos poner doble guión bajo seguido del nombre del campo en el respectivo modelo

1. Podemos agregar un organizador, para navegar por ejemplo por fechas...

**date\_hierarchy = 'created'**

Al guardar cambios y actualizar el navegador podemos ver el añadido

1. También podemos agregar el campo de filtrado por categorías o por autor...

En este caso por categorías... **list\_filter = ('categories\_\_name\_cat',)** Notese que cuando en una tupla envío un solo parámetro, debo colocar a continuación una “coma” delo contrario genera error. Guardamos los cambios y verificamos el resultado en el navegador. Es una herramienta muy útil con la que podemos mostrar resultados en este caso por “categorías”. Es mucho más útil al ponerle otro filtro, por ejemplo...

**list\_filter = ('categories\_\_name\_cat','author\_\_username')**

1. Implemento dentro de la clase-modelo **AdminPost** un método..

**def post\_cat(self,obj):**

**return ", ".join([c.name\_cat for c in obj.categories.all().order\_by('name\_cat')])**

1. Y modifico la var **list\_display**...

**list\_display = ('title','author', 'creation\_date',’post\_cat’)**

**En azul la modificación añadida. Podemos ver que se llama a def post\_cat().**

**Al guardar cambios y actualizar el navegador vemosla nueva columna que nos muestra las categorías a las que pertenece cada publicación, separadas por coma**

1. Pongamos en español el nombre de la nueva columna...

añadimos después del método post\_cat ...

post\_cat.short\_description = 'Categorías'

con esto ya nos pone el nombre de la columna en español

Vamos a mostrar el contenido anterior que ye tenemos en la BD, por medio de nuestras plantillas

1. Lo primero que debemos hacer es mover el template infoblog.html de core hacia el directorio infoblog/templates/infoblog de la app infoblog
2. Ahora trasladamos la vista de info blog que está en vews.py de core hacia views.py de infoblog ...

**def blog(request):**

**return render(request,'infoblog/infoblog.html')**

1. como este cambio genera errores en urls.py de core. Borramos el path correspondiente a infoblog, y lo implementamos en un archivo urls.py que vamos a crear en la app infoblog...

**from django.urls import path**

**from infoblog import views**

**urlpatterns=[**

**path('',views.blog,name='Blog'),**

**]**

1. ahora vamos al urls.py del proyecto general, y en la var urlpatterns añadimos..

**path('blog/',include('infoblog.urls')),**

guardamos cambios y probamos el apartado de infoblog en nuestra web

1. ahora vamos a mostrar mediante la plantilla infoblog los datos de la BD porque de momento la info es estática....

En el archivo views.py de la app infoblog añadimos el sig código

**from django.shortcuts import render**

**from infoblog.models import Post**

**# Create your views here.**

**def blog(request):**

**posts = Post.objects.all()**

**return render(request,'infoblog/infoblog.html',{'post':posts})**

en azul el código añadido. con el import obtenemos la información de los posts creados. en la var **posts** obtenenmos un objeto con la información cargada de la BD y en el render añadimos un queryset

1. vamos al template infoblog.html y modificamos su código para mostrar de manera dinamica el contenido...

para resolver... GEnera error cuando intento cargar un infloblog que no tiene imagen

1. implementamos dentro del tag-template bock content un tag for ...

**{% for post in posts %}**

**{% endfor %}**

y dentro de este tag, ponemos el código html correspondiente a la etiqueta **section** donde vamos reemplazando el código donde se debe mostrar de manera dinámica, por ejemplo en la fecha... **{{post.creation\_date}}**

para el titulo... **{{post.title}}**

para la imágen... **{{post.image.url}}**

en el parrafo del contenido.. **{{post.content}}**

en publicado por.... **{{post.author}}**

y para la categoría, de momento lo dejamos con html estático

Si guardamos y actualizamos el navegador, pordemos ver que la fecha sale en un formato largo. Lo queremos corto al estilo DD/MM/AA asi que...

1. vamos a formatear la fecha añadiendo en su código correspondiente...

**{{post.creation\_date|date:”SHORT\_DATE\_FORMAT”}}**

En azul el código añadido. Hay muchas otras formas de formatear la fecha, eso lo podemos ver buscando en google la documentación

1. si miramos detalladamente el contenido. podemos ver que se nos presenta el parrafo sin saltos de linea, cuando varios de ellos si tienen esos saltos de linea, pero Django no los pmuestra por defecto, vamos a modificar ese detalle añadiendo en el parrafo...

**{{post.content|linebreaks}}**

1. para mostrar la o las categorías a las que pertenece cada infoblog, creamos un for dentro del for ya establecido, justo donde queremos mostrar en nombre de la categoría...

**{% for category in post.categories.all %}**

**{{category.name\_cat}} {% if not forloop.last %},{% endif %}**

**{% endfor %}**

el if que implementamos nos sirve para mostrar una coma siempre que NO sea el fianl del bucle. Eso nos muestra separadas por comas las categorías, cuando sea el caso de que el infoblog tenga mas de una

1. Nos queda generar enlaces en las categorías presentadas...