***Django***

**Qué es?**

Es un **framework web de código abierto escrito en Python**

**Qué es un Framework?**

Es un **entorno de trabajo formado por un conjunto de herramientas y librerías** que nos ayudan a crear sitios web

**Para qué sirve Django?**

**\*Para crear sitios web COMPLEJOS de forma rápida y sencilla** (Si necesitamos un sitio web pequeño es mejor usar HTML)

**\*Facilita y simplifica la realización de tareas repetitivas y pesadas** que son comunes a la hora de crear todos los sitios web

\***Permite reutilizar códigos de forma sencilla**, ya que hay códigos que podemos reutilizar de un sitio web a otro

**\***Permite que **utilicemos una** **DataBase** (Base de Datos), que **autentiquemos a nuestros usuarios** (AUTH), **crear formularios** (Form), **manejar la seguridad de la aplicación** (Protected Routes) e incluso **tener un Panel de Administración ya creado**

**\*Nos permite como desarrolladores centrarnos la mayor parte del tiempo en crear funcionalidades para nuestros proyectos sin necesidad de reinventar la misma funcionalidad que todos los proyectos necesitan**, como leer archivos, cargar imágenes, crear formularios, etc

***Modelo Vista Controlador (MVC)***

**Django se basa en cuanto a su estructura en este patrón, si bien NO LO USA EXACTAMENTE, hay unos pequeños cambios en el diseño**

**Este patrón consiste en dividir cualquier aplicación (general o web) en 3 módulos**:

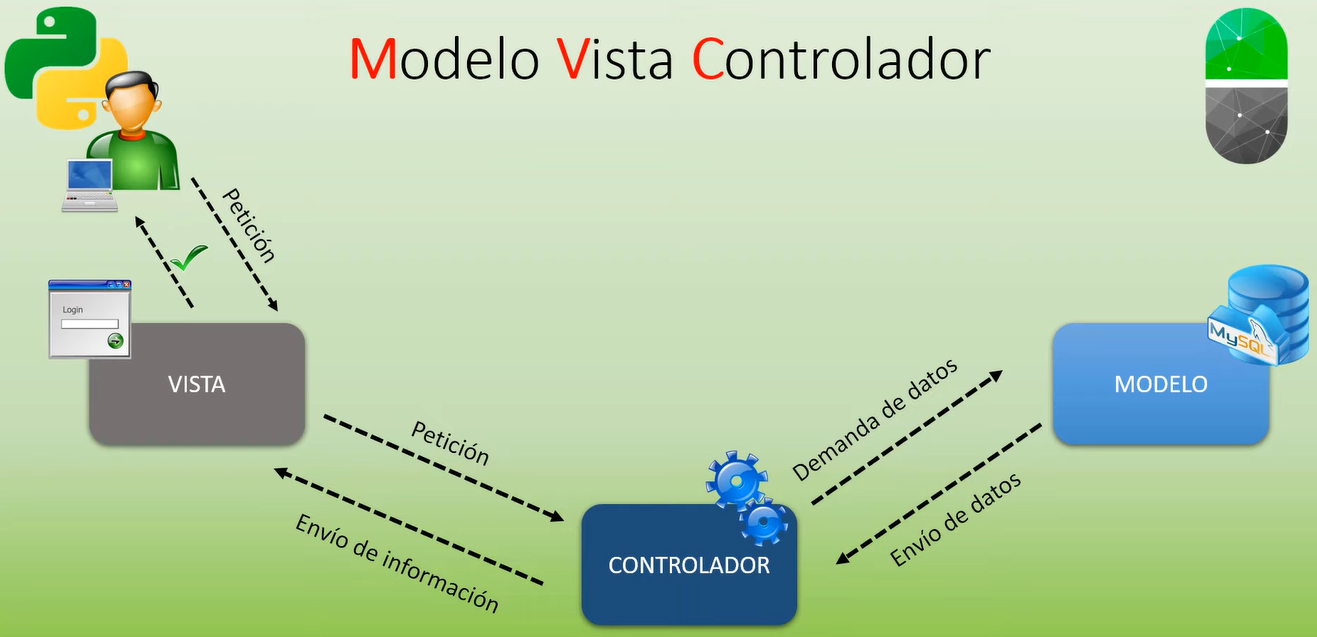
Modelo, Vista y Controlador

El **Modelo** es el módulo que **se encarga de gestionar los datos** (obtener información de una base de datos)

La **Vista** es el módulo **encargado de mostrar la información al usuario** (lo que el usuario ve o interactúa)

El **Controlador** es el módulo que **se encarga de gestionar todas las comunicaciones que existen entre la Vista y el Modelo**

**Pasos del MVC**

1. Lo primero que hace el **usuario** es **hacer una petición a la Vista** (ej: formulario web)
2. **Esa petición es registrada por el controlador**, y **al recibirla hace una Demanda de Datos al Modelo** (ej: si la petición es para reservar un hotel, el Modelo se encarga de ver cuáles son las habitaciones libres)
3. **El Modelo recibe esa demanda de datos y envía esos datos demandados al Controlador, el cual los envía a la Vista, quien es la encargada de mostrar esa información al usuario**

**Ventajas del MVC**

Las ventajas que ofrece son:

**Aplicaciones más funcionales, mantenibles y escalables** (facilita agregar más funciones a futuro, ya que las aplicaciones estarán divididas en esos 3 grandes módulos)

**Modelo que utiliza Django**

Django se basa en el Modelo Vista Controlador (MVC), pero con unos pequeños cambios en el nombre de los módulos:

**El modelo de Django se llama Model Template View (MTV)**

"Model-Template-View" (MTV) es un patrón de diseño utilizado en el desarrollo de aplicaciones web.

Model se refiere a la capa que maneja los datos y su lógica de negocio. Esta capa incluye la definición de clases de objetos y la interacción con la base de datos.

Template se refiere a la capa que se encarga de la presentación visual de la información. Esta capa utiliza un lenguaje de marcado para definir cómo se mostrarán los datos.

View se refiere a la capa que actúa como intermediario entre las capas de Modelo y Template. Esta capa maneja la lógica de la aplicación y actúa como controlador, recibiendo las solicitudes del usuario y coordinando la interacción entre el modelo y la plantilla.

En resumen, MTV es una arquitectura de software que separa en módulos la aplicación web, dividiéndola en tres partes: la capa de Modelo, la capa de Template y la capa de View, lo que permite una mejor organización y mantenimiento del código.

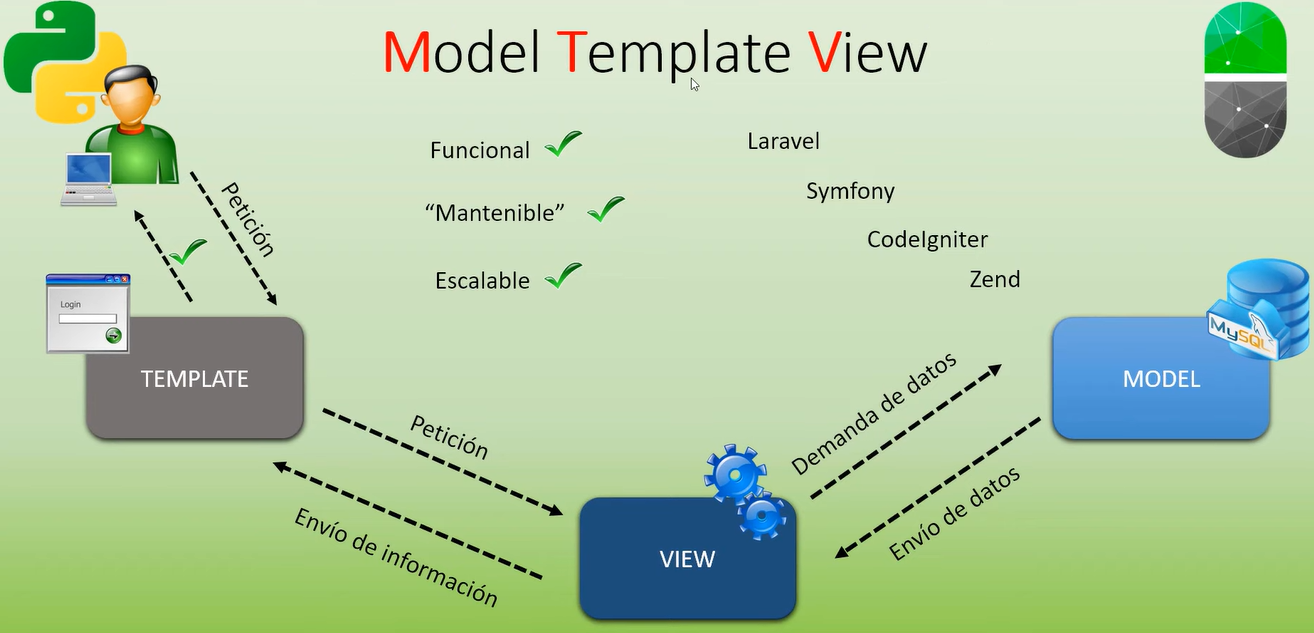
Principio del formulario

Si bien cambian los nombres de los 3 módulos principales, **los pasos son los mismos**

**Se sustituyen:**

La **Vista** por **Template**

El **Controlador** por **View**

El **Modelo** por **Model**

***Crear proyecto en Django***

**Crear e instalar el entorno virtual para cada proyecto**

Para evitar que nuestros proyectos de Django interfieran con otros, es necesario crear un Entorno Virtual para cada proyecto, teniendo así en cada carpeta que creemos nuestra propia versión de Python, pip y paquetes

Nos ayuda a que cada carpeta tenga sus propias instalaciones sin interferir entre ellas

**Comandos desde CMD**

Para **crear un entorno**: **mkvirtualenv nombre\_del\_entorno**

Para **ver todos los entornos que hemos creado**: **WORKON**

Para **entrar a un entorno**: **workon nombre\_del\_entorno**

Para **salir de un entorno**: **deactivate**

Para **borrar un entorno**: **rmvirtualenv nombre\_del\_entorno**

Para **crear un proyecto y además inicializarlo al mismo tiempo en la carpeta donde está situada la carpeta PROJECT\_HOME**: **mkproject nombre\_del\_entorno**

Aclaración: es el más completo ya que no solo nos crea el entorno sino que nos crea al mismo tiempo el proyecto

**Comandos desde Visual Studio Code para crear proyectos**

**Para cada Entorno Virtual que usemos debemos instalar Django poniendo en la terminal de Visual Studio:**

**pip install django**

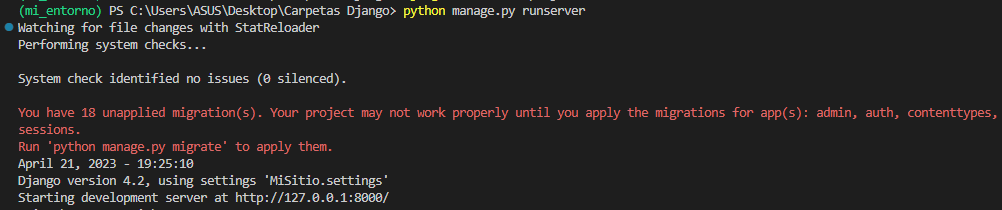
Instalará Django tan solo en ese proyecto

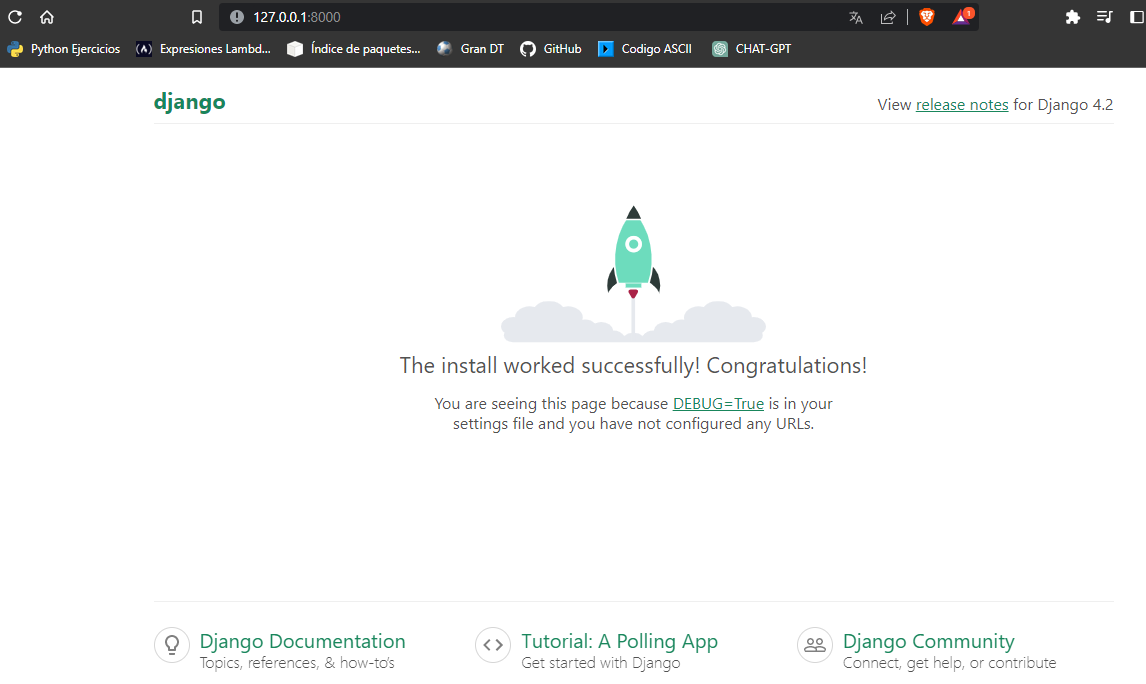
Luego debemos crear e inicializar el proyecto, poniendo: **django-admin startproject nombre\_del\_proyecto .**

Importante: no utilizar nombres de proyectos como django, Python, test, etc

Así evitaremos errores o fallas de funcionamiento

**Junto con el proyecto de creará un archivo llamado manage.py**, que se encarga de administrar el proyecto

**Luego debemos ejecutar el proyecto** usando: **python manage.py runserver**

**Si ingresamos a la dirección http que nos muestra ahí, ingresaremos al proyecto**

También, el comando **nos crea un archivo llamado db.sqlite3**, el cual **es la base de datos por defecto que utiliza Django para desarrollo**

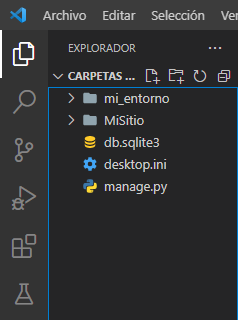
**Aclaración importante:** **si trabajamos con más de 1 proyecto web puede que haya algún proyecto que ya esté usando el mismo puerto y de esa forma no funcione**

En el ejemplo el puerto es http://127.0.0.1:**8000/**

**Si queremos evitar eso, sencillamente cambiemos el puerto 8000 por otro puerto**

Para eso debemos **limpiar la consola del proyecto** con **ctrl + c** y luego poner:

**python manage.py runserver Numero\_de\_puerto**

***Estructura de un proyecto de Django***

La carpeta **mi\_entorno** es la **carpeta del entorno virtual**

La carpeta **MiSitio** contiene el **código fuente de la aplicación**

**Db.sqlite3** es la **base de datos de la aplicación**

**Manage.py** nos ayuda a **ejecutar comandos administrativos**

***Creación de una App en Django***

**Los proyectos de Django están conformados por aplicaciones (apps)**, las cuales **son partes del proyecto y están organizadas por django en** **apps**, **las cuales contienen funcionalidades agrupadas**

Ej: una puede ser un blog, otra la autenticación de usuario, otra artículos, otra una tienda, etc

**Las apps se pueden acoplar o desacoplar del proyecto principal, dependiendo si las necesitamos o no**

El archivo manage.py nos ayuda a crear estas apps

**Crear una App dentro de un proyecto**

En la terminal de Python ponemos **python manage.py startapp nombre\_de\_la\_app**

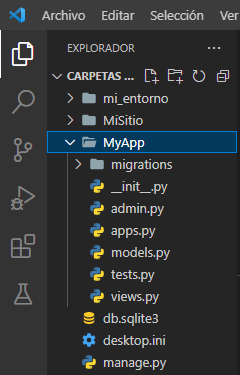
**Al crear la app, se nos creará una carpeta con el nombre que le hayamos puesto** a la misma, permitiéndonos **usar esa app o partecita del proyecto para ese fin determinado**

**Podemos crear la cantidad de apps que querramos, lo importante es conectarlas a la carpeta base de mi sitio**, cosa que veremos más adelante

**Cada carpeta de las apps no está vacía, sino que contienen sus propios archivos que nos permiten especificar las funcionalidades de la app**, y que al final se acoplarán a la carpeta principal del proyecto, llamando los archivos de la app desde la carpeta principal de mi sitio (ya que es el núcleo de todo el proyecto y desde la cual podemos configurar todas las apps que estén conectadas a ella)

Podemos eliminar las apps que querramos siempre y cuando no estén enlazadas aún a la carpeta principal

***Estructura de una App en Django***

**Cada App tiene sus propios archivos**, los cuáles son:

**Views.py**: Desde ahí podemos **enviar archivos HTML**, ya que **decidiremos que**

**archivos enviar al cliente o al navegador para que pueda verlos en pantalla**

**\_init\_.py**: Está **para que las carpetas sean consideradas módulos de Python**

**Migrations.py**: si bien al principio solo tiene un archivo init, **se va a ir llenando**

**a medida que nosotros vayamos modificando la base de datos**

Si bien podemos usar código de SQL en esta subcarpeta, **la misma viene con**

**un módulo especial que nos permite interactuar con la base de datos a través**

**de código de Python, sin necesidad de escribir en SQL**

**Admin.py**: contiene un panel de administrador que **nos permite administrar**

**nuestro proyecto y apps**

También permite añadir nuestras apps a la carpeta principal

**Apps.py**: Nos permite **configurar la app en la cual se encuentra**

**Models.py**: Nos permite **crear clases**, las cuales se convertirán en tablas SQL

También se encarga de **realizar las modificaciones que hagamos**, mediante unos archivos de los que **se encargará de ponerlos en la carpeta Migrations.py**

**Test.py**: Sirve para que podamos escribir testing de nuestras vistas o archivos que enviemos al navegador

***Hello World en Django***

Lo hacemos mediante el archivo Views.py de mi app

Allí creamos una función que retorne un mensaje al cliente o navegador

Para hacer un "Hola Mundo" en Django, sigue los siguientes pasos:

**1)** **Instala Django** si aún no lo tienes instalado en tu sistema. Puedes hacerlo usando pip en tu terminal:

**pip install Django**

**2)** **Crea un nuevo proyecto Django**. Para ello, **en tu terminal**, ve al directorio en el que deseas crear el proyecto y escribe lo siguiente:

**Django-admin startproject proyecto**

"proyecto" es el nombre que le darás a tu proyecto. Puedes elegir cualquier otro nombre que prefieras.

**3)** Una vez que el proyecto ha sido creado, navega al directorio del proyecto y **crea una nueva aplicación**. Para ello, escribe **en tu terminal**:

**python manage.py startapp myapp**

"myapp" es el nombre que le darás a tu aplicación. De nuevo, puedes elegir cualquier otro nombre que prefieras.

**4)** Ahora, **abre el archivo views.py en la carpeta "myapp"** y agrega lo siguiente:

**From django.http import HttpResponse**

**def hello\_world(request):**

**return HttpResponse("¡Hola Mundo!")**

HttpResponse en Django es una clase que se utiliza para devolver una respuesta HTTP a una solicitud realizada por un cliente. La clase HttpResponse permite enviar datos en el cuerpo de la respuesta, configurar los encabezados HTTP y establecer el código de estado HTTP

**El parámetro request nos permite recibir información del usuario que el cliente nos esté enviando cuando ejecute esta función**

**5)** Después, vamos a la **carpeta mi sitio, al archivo urls.py** y agregamos:

**From myapp.views import hello\_world**

**Urlpatterns = [**

**…**

#cuando el usuario ingrese al link de la página llamaremos automáticamente a la función hello\_world

**Path(‘ ’, hello\_world)**

**]**

**6)** **Ejecuta el servidor de desarrollo de Django en tu terminal**:

**python manage.py runserver**

***Include***

Sabemos que **en** el archivo urls.py de mi sitio (donde ponemos las configuraciones globales) podemos agregar las urls de todas nuestras apps, pero **si tenemos muchas apps *es mejor que cada app guarde sus propias urls en sus propios archivos urls.py***

Para eso **debemos crear un archivo urls.py en cada app que vayamos a usar**

**Que poner dentro de los archivos urls.py de cada app**

Primero debemos ***importar desde* django.urls *el módulo* path**

Luego ***creamos una lista llamada urlpatterns que va a contener todos los módulos* path *que necesitemos agregar***

#en cada app creamos este archivo e importamos el módulo path

from django.urls import path

#además importamos las funciones que usará cada app

from MyApp.views import hello\_world, about

urlpatterns = [

    #cuando el usuario ingrese al link de la página llamaremos automáticamente a la función hello\_world

    path('', hello\_world),

    #cuando el usuario ingrese a la pagina/about llamará a la función

    path('about/', about)

]

**Observación: el path(‘about/’, about) significa que si el usuario ingresa a la página y pone /about se mostrará esa función**

**Poner el primer parámetro del path vacío significa que esa función se mostrará sin necesidad de que el usuario agregue otra dirección, ya que por defecto está “conectada” a la dirección anterior del path**, en este caso a la página principal

Si el parámetro vacío estuviera bajo el path about, pertenecería a él y solo podría verse si se ingresa /about

**Como añadir las urls de cada app a mi sitio**

**En el archivo urls.py de mi sitio agregamos la función Include** (la cual es parte de django.urls al igual que el módulo path), la cual **sirve para incluir un bloque de urls que vienen por parte de una app**

#importamos la función include para poder incluir el paquete de urls de cada app que usemos

from django.urls import path, include

urlpatterns = [

    path('admin/', admin.site.urls),

    #incluimos el paquete de urls que está en esa ruta

    path('', include('MyApp.urls'))

]

**Modelos y Base de Datos con SQLite3**

**Para actualizar nuestra base de datos debemos usar las migraciones**

Las migraciones son una de las formas para poder **actualizar nuestras bases de datos a partir de código de Python**

Las migraciones **nos simplifican mucho**, ya que al crear un nuevo proyecto deberíamos por nuestra cuenta crear las tablas, bases de datos, etc. En cambio, **las migraciones se encargan de esas tareas**, y es muy importante ya que **cuando vayamos modificando la base de datos o creando datos nuevos tendremos que ir guardando esos cambios y actualizando la base de datos**

**Usar DB browser for SQLite**

Este programa **nos permite tener una interfaz gráfica para poder ver la base de datos de SQLite**

En el programa vamos a open database y allí buscamos el archivo **db.sqlite3**

Una vez hecho esto **nos mostrará todo lo que contiene ese archivo**, que en un inicio tendrá todo en cero

**Antes de seguir debemos tocar ctrl + c en la terminal para volver a escribir comandos en la terminal**

**Para generar migraciones debemos ingresar en la terminal el comando:**

**Python manage.py makemigrations**

**Para actualizar la base de datos con las migraciones pendientes debemos ingresar en la terminal:**

**Python manage.py migrate**

**Este comando se encarga de crear las tablas y demás cosas en nuestra base de datos**

Ahora, **al apretar F5 dentro del DB browser, nos mostrará todo lo que ha creado por defecto dentro de la base de datos de SQLite**

Con click derecho en la tabla que elijamos y browse table, veremos todo lo que contiene

Hay tablas que son muy usadas, como por ejemplo la de autenticación de usuario (auth\_user)

**Crear nuestras propias tablas**

Si bien a esta altura ya tenemos creadas las tablas por defecto, **si nosotros agregamos nuestras propias tablas debemos crear un modelo (código de python que se transforma en una tabla de SQL) en el archivo models.py de la aplicación en la que querramos hacerlo**

**En models.py creamos las clases que necesitemos**

Supongamos que la app se basa en proyectos y tareas. Creamos una clase que se llame Project

**Class Project(models.Model)**

**La clase hereda los modelos de django**

**Dentro de la clase colocamos atributos que nos permiten especificar los datos que guardaremos en la tabla**

Supongamos que queremos generar una tabla que guarde un nombre, pero que ese nombre no supere los 100 caracteres

#la tabla creada hereda los modelos de django

class Project(models.Model) :

    #supongamos que queremos que la tabla guarde un nombre, pero que este no supere los 100 caracteres

    name = models.CharField(max\_length=100)

    #CharField significa que es un string

**Conectar cada app con el sitio principal**

**Para que el Sitio global conozca estos modelos que agregamos debemos conectarlos**, para eso vamos a la configuración global (mi sitio) y allí al archivo **settings.py**

Una vez en el archivo vamos a la sección installed\_apps, la cual **tiene dentro todas las conexiones del proyecto principal con las apps**

Simplemente **agregamos dentro de esa lista el nombre de las apps que deseemos conectar con el sitio principal** y de esa simple forma guardamos y listo

Luego de eso **debemos ejecutar las migraciones para poder dejar actualizada la base de datos**, para eso en la terminal ponemos: **python manage.py makemigrations**

Esto **nos creará un archivo en la carpeta Migrations de la app correspondiente, en la cual estará creada automáticamente la tabla en la lista operations** (no necesitamos tocar nada)

Para **terminar de migrar los datos debemos poner en la terminal**: **python manage.py migrate**

Una vez hecho todo esto, **al actualizar en DB Browser tendremos la nueva tabla creada llamada**:

**Nombre\_de\_la\_app\_nombre\_de\_tabla**

En resumen… **Que es una tabla?**

En Django, una tabla se refiere a una clase modelo (Model) que define la estructura y el comportamiento de una tabla en una base de datos relacional. Cada tabla en una base de datos se corresponde con un modelo de Django y cada columna en la tabla se corresponde con un campo en el modelo.

La definición de una tabla en Django se realiza mediante la creación de una clase que hereda de la clase **models.Model**. Esta clase representa la estructura de la tabla y define los campos que se utilizarán en ella. Por ejemplo, si quisiéramos definir una tabla llamada "Persona" con campos como "nombre", "edad" y "correo electrónico", crearíamos una clase modelo como esta:

from django.db import models

class Persona(models.Model):

    nombre = models.CharField(max\_length=100)

    edad = models.IntegerField()

    correo\_electronico = models.EmailField()

En este ejemplo, la clase **Persona** define tres campos: **nombre**, **edad** y **correo\_electronico**. El campo **nombre** es una cadena de caracteres con un máximo de 100 caracteres (**CharField**), **edad** es un entero (**IntegerField**) y **correo\_electronico** es un campo de correo electrónico (**EmailField**).

Una vez definido el modelo, Django puede generar automáticamente la tabla en la base de datos correspondiente mediante el uso de migraciones. Además, el modelo se puede utilizar para realizar operaciones en la base de datos, como la creación, lectura, actualización y eliminación de registros.

**Tipos de modelos de Django**

En Django, existen varios tipos de modelos que se utilizan para definir diferentes tipos de campos y comportamientos en una tabla de base de datos. Algunos de los tipos de modelos más comunes son:

1. **CharField**: Este tipo de campo se utiliza para definir una cadena de caracteres con una longitud máxima determinada.
2. **TextField**: Este tipo de campo se utiliza para definir una cadena de texto larga, sin una longitud máxima determinada.
3. **IntegerField**: Este tipo de campo se utiliza para definir un número entero.
4. **BooleanField**: Este tipo de campo se utiliza para definir un valor booleano (verdadero/falso).
5. **DateField**: Este tipo de campo se utiliza para definir una fecha.
6. **DateTimeField**: Este tipo de campo se utiliza para definir una fecha y hora.
7. **ForeignKey**: Este tipo de campo se utiliza para definir una relación entre la tabla con otra tabla. Entre paréntesis ponemos el nombre de la tabla con la que está relacionada.
8. **ManyToManyField**: Este tipo de campo se utiliza para definir una relación muchos a muchos entre dos modelos. El campo contiene una tabla intermedia que almacena las relaciones entre los registros de los dos modelos.

Además de estos tipos de campos, también existen otros tipos de modelos que se utilizan para definir comportamientos específicos, como **AbstractBaseUser** y **PermissionMixin** para la implementación de sistemas de autenticación y permisos en aplicaciones web.

Ej: #la tabla creada hereda los modelos de django

class Project(models.Model) :

    #supongamos que queremos que la tabla guarde un nombre, pero que este no supere los 100 caracteres

    name = models.CharField(max\_length=100)

    #CharField significa que es un string

#creamos otra tabla, esta vez de tareas

class task(models.Model) :

    #el título será un string con un máximo de 150 caracteres

    title = models.CharField(max\_length=150)

    #la descripción será un string largo sin un máximo de caracteres

    description = models.TextField()

    #de esta forma decimos que esta tabla está relacionada con la tabla Project

   project = models.ForeignKey(Project, on\_delete=models.CASCADE)

#el parámetro on\_DELETE=models.CASCADE sirve para que cuando se elimine una tabla, se eliminen todos sus elementos

**Django Shell**

Ya hemos visto como crear tablas en Django a través de las migraciones, ahora debemos ver como interactuar con estas tablas y crear datos en ellas

El shell de Django es una interfaz de línea de comandos interactiva que permite interactuar con una aplicación de Django y su base de datos. El shell de Django se inicia ejecutando el comando **python manage.py shell** en la línea de comandos desde la carpeta raíz del proyecto Django.

Una vez iniciado el shell de Django, se pueden realizar operaciones en la base de datos y en los modelos de la aplicación de Django mediante comandos y consultas de Python. Esto puede ser útil para realizar pruebas, experimentar con diferentes consultas y operaciones en la base de datos, y para realizar tareas administrativas de la aplicación

Empezamos ejecutando en el terminal el comando **python manage.py shell**

**Desde ahí podemos importar los modelos desde cada App**

Para poder **interactuar con las clases que creamos en cada app** debemos poner:

**From nombre\_de\_la\_app.models import nombre\_de\_clases**

Luego, si queremos **crear un objeto dentro de una clase particular** ponemos:

**Variable = Nombre\_de\_la\_clase (name = “aplicación móvil”)**

Luego debemos **guardar esa variable, para crear así una fila en la tabla de la app correspondiente y así nos aparezca en el DB Browser de SQL**

**Variable.save ()**

**Como mostrar todos los objetos de una clase**

**Nombre\_de\_la\_clase.objects.all ()**

**Como mostrar un objeto en particular de una clase**

**Nombre\_de\_la\_clase.objects.get (id=nro\_del\_objeto)**

O también:

**Nombre\_de\_la\_clase.objects.get (name = “nombre\_del\_objeto”)**

**Como cerrar el Shell de Django**

**Exit ()**

**Como ejecutar tareas**

Como tenemos una relación de las tareas con un proyecto, primero debemos tener el campo del proyecto y luego crear una tarea allí

Primero importamos las clases que usaremos: **from nombre\_de\_app import nombre\_de\_clases**

Luego guardamos el objeto que queremos utilizar en una variable:

**Variable = nombre\_de\_clase.objects.get (id=nro\_de\_id)**

O

**Variable = nombre\_de\_clase.objects.get (name=”nombre\_del\_objeto”)**

Si queremos ver todas las tareas que tiene una clase ponemos: **variable.nombre\_de\_clase\_set.all ()**

Si queremos crear una tarea ponemos: **variable.nombre\_de\_clase\_set.create (title= “título\_de\_la\_tarea”)**

**Params (Parámetros)**

**Aprenderemos a recibir datos desde el navegador para luego poder pasarlos por operaciones en una base de datos**

**Una de las formas de recibir datos desde la página es a través de los parámetros en la URL**

Ej: queremos crear una URL en donde el usuario al entrar ingrese /hello/username y nos quede guardado ese username a nosotros.

Además de eso queremos modificar la clase hello World para que diga hello y el username que el usuario puso

Para eso **en el archivo urls.py de la App agregamos**:

path('hello/<str:username>', hello\_world)

**<str: username> significa que el usuario ingresara allí un string y que haremos que una variable llamada username guarde ese valor**

Luego vamos al archivo **views.py** y modificamos (o creamos en caso de que la función aún no exista) la función que habíamos puesto como hello\_world

#función hello

def hello(request, username) : #Django se encarga de pasarnos este parámetro request

    return HttpResponse("<h2>Hello %s</h2>" %username)

    #concatenamos la variable username luego del hello

    #usamos el %s para indicar que allí concatenaremos una variable, y luego decimos que cambiaremos el % por username

**Lo que hace el template** **%s** **es hacer una concatenación, y luego del string que ponemos en la función debemos poner que variable reemplazaremos en el %, en este caso reemplazamos el % por la variable username previamente extraída de lo que el usuario ingresó en la URL de la página**

**Params y Models**

Veremos cómo **usar parámetros para consultar los modelos que hemos creado en nuestra base de datos**

**Como mostrar los objetos que querramos de cada clase a través de código de Python**

Para poder hacerlo **necesitamos hacer consultas**, esto lo hacemos **a través de los modelos**

**Primero importamos en el archivo views.py de la App:**

#JsonResponse nos permite devolver un formato que el navegador pueda entender fácilmente

from django.http import HttpResponse, JsonResponse

#importamos las vistas o clases de las cuales mostraremos sus objetos

from .models import Project,Task

Luego, en el mismo archivo **views.py**, **creamos una clase que nos permita mostrar en la página todos los objetos de una clase que esté en la base de datos**, en este caso de la clase Project

def projects (request) :

    #creamos una lista llamada projects que va a contener todos los objetos o valores de la clase Project

    projects = list(Project.objects.values())

    return JsonResponse(projects, safe=False)

    #nos mostrará todos los objetos, mediante la lista projects. El parámetro safe=False es para evitar errores nomás

Por último **lo agregamos a la lista urlpatterns del archivo urls.py de la app**

path('projects/', projects)

De esta forma, **cuando el usuario ingrese a la página y agregue /projects podrá ver el nombre de todos los proyectos que contiene la base de datos**

**Cómo mostrar el objeto que quiera el usuario de forma individual**

Tenemos una lista de tareas (Task) en la base de datos. Ahora queremos que el usuario ingrese /tasks/nro\_id y que se le muestre ese objeto en específico, asi que para eso:

Al igual que el caso anterior, **debemos agregarlo a la lista urlpatterns del archivo urls.py de la app**

#cuando ingresemos a la URL y pongamos la dirección tasks/nro\_de\_ID\_buscado nos retornará el nombre de esa tarea

path('tasks/<int:id>', tasks)

**Estamos avisando que el usuario deberá ingresar /tasks/id y que el id si o si debe ser entero**

Además, **debemos crear la función que nos permita realizar esa acción, eso lo hacemos en el archivo views.py de la app**

def tasks (request, id) :

    task = Task.objects.get(id=id)

    return HttpResponse("Tasks: %s" %task.title)

**La función tomará el id que el usuario ingrese y lo usará para buscar ese mismo id en los objetos de la clase Task, y al encontrarlo nos devolverá una concatenación (la cual la hacemos usando el %s dentro del string y luego del string aclarar por qué cosa vamos a intercambiar ese %)**

Pero… **y si no encuentra el id y queremos que no se caiga el servidor, sino que en vez de eso le muestre al usuario un error 404?**

Esto sirve para que cuando el usuario ingrese un objeto que no exista le salga ese cartel y pueda entender cuál es el error que está cometiendo

En primer lugar, **en el archivo views.py de la app, importamos el módulo de django que se encargará de ver si el objeto existe, y en el caso de que no exista nos muestre un error 404 (page not found)**

from django.shortcuts import get\_object\_or\_404

Luego **modificamos la función tasks en el mismo archivo views.py**

def tasks (request, id) :

    task = get\_object\_or\_404(Task, id=id)

    return HttpResponse("Tasks: %s" %task.title)

**De esta forma, si el objeto existe nos devolverá el title, y si no existe nos devolverá un error 404**

**Y si en vez de buscar por id queremos que el usuario busque por el nombre del objeto de una clase?**

En primer lugar, **importamos la función tasks\_nombre y agregamos a la lista urlpatterns del archivo urls.py de la app la dirección que el usuario deberá ingresar**

En este caso:

#cuando ingresemos a la URL y pongamos la dirección tasks/nombre\_de\_tarea nos retornará el id del objeto (en este caso tarea)

path('tasks/<str:title>', tasks\_nombre)

Luego, **en el archivo views.py de la app**, **definimos la función que se encargará de buscar por nombre el objeto de una clase determinada**

def tasks\_nombre (request, title) :

    task = Task.objects.get(title=title)

    return HttpResponse("task: %s" %task.title)

**El título que el usuario ingrese en la página será utilizado para buscar el objeto que tenga exactamente ese mismo nombre y lo mostrará en la página**

**Django Admin**

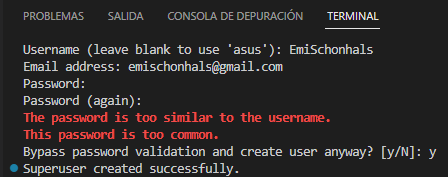
**Como ingresar al panel de administrador de Django**

**Django tiene un panel de administrador por defecto en la lista urlpatterns del archivo urls.py del SITIO, al cual ingresamos con /admin en la página web, ya que la misma llama a un módulo integrado llamado admin.site.urls que se encarga de permitirnos usar el panel de administrador en la web**

**Al ejecutar el servidor local y acceder a esa URL nos pedirá ingresar un usuario y una contraseña, las cuáles debemos crear previamente, para eso usamos el script manage.py el cual nos permite crear un usuario y encriptar la contraseña**

**Como crear un usuario para usar el panel de administrador de Django**

Para eso **en la terminal de Django ponemos** **python manage.py createsuperuser**

Luego, nos permitirá ingresar los datos para crear el usuario:

**El comando nos encripta la contraseña y nos pide confirmación en el caso de que la misma sea similar al nombre de usuario o sea muy común**

Una vez creado nuestro usuario, ejecutamos el servidor e ingresamos con nuestra cuenta al panel de administrador de Django

**Funcionalidades del panel de administrador de Django**

Nos permite **ver todos los usuarios de nuestra aplicación**, además de también **ver los grupos**, los cuales se basan en los permisos que tenga cada usuario de la app

Además podemos **importar las tablas y clases al panel de administrador**

**Como agregar Clases o Tablas al panel de administrador de Django**

**Debemos importar los modelos que tengamos al archivo admin.py en la carpeta de la app**

Para eso debemos **importar en ese archivo los modelos que hayamos creado en la carpeta Models.py de la app**, en este caso:

#importamos los modelos que queremos agregar al panel de administrador de Django

from .models import Task,Project

Luego los **agregamos al panel de administrador de Django** poniendo:

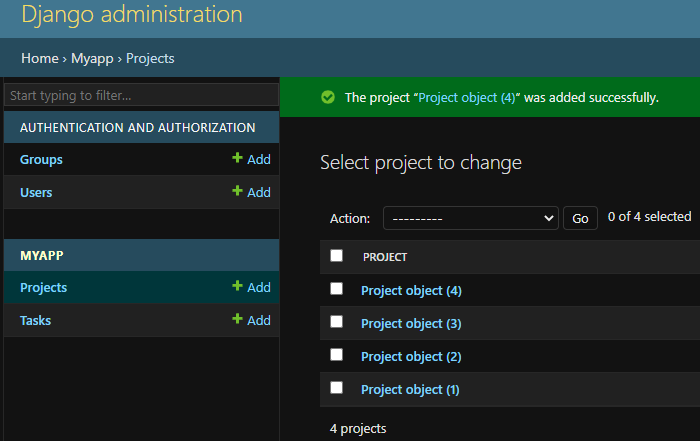
#los agregamos al panel de administrador de Django usando el módulo admin importado por defecto en este archivo

admin.site.register(Project)

admin.site.register(Task)

**Al guardar y ejecutar de nuevo el servidor ya nos aparecerán las clases que hemos añadido**, en este caso las clases Project y Task

**Como crear Proyectos dentro de la clase que deseemos en Django admin**

**Vamos al proyecto y tocamos la opción add, allí nos pedirá el nombre que querramos ponerle al objeto creado**

Como vemos, **al entrar a la clase que querramos nos mostrará la cantidad de proyectos que contenga esa clase**

Pero… **Como hacer si queremos que en vez de nombre\_de\_clase object (id) nos muestre el nombre de ese objeto?**

Para eso **vamos al archivo Models.py de la app, y en la función donde creamos la clase en la que queremos que nos muestre el nombre de sus objetos agregamos la función que nos lo permite**, la cual es:

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

**Return self.name**

De esta forma, el modelo de la clase quedaría asi:

#la tabla creada hereda los modelos de django

class Project(models.Model) :

    #supongamos que queremos que la tabla guarde un nombre, pero que este no supere los 100 caracteres

    name = models.CharField(max\_length=100)

    #CharField significa que es un string

    #función que nos permite ver en Django admin los nombres de los objetos de la clase Project

    def \_\_str\_\_(self):

        return self.name #devuelve los valores almacenados en la variable name

**Si queremos que en el panel de administrador de Django nos salgan los nombres de los objetos en vez de nombre\_de\_clase object (id) debemos repetir este proceso en las clases en las que deseemos que pase**

Por ejemplo si queremos hacer lo mismo con la clase Task quedaría así:

class Task(models.Model) :

    #el título será un string con un máximo de 150 caracteres

    title = models.CharField(max\_length=150)

    #la descripción será un string largo sin un máximo de caracteres

    description = models.TextField()

    #de esta forma decimos que esta tabla está relacionada con la tabla Project

    project = models.ForeignKey(Project, on\_delete=models.CASCADE)

    #el parámetro on\_DELETE=models.CASCADE sirve para que cuando se elimine una tabla, se eliminen todos sus elementos

    def \_\_str\_\_(self):

        return self.title #devuelve los valores almacenados en la variable title

Pero… **y si queremos que el panel de administrador de Django nos muestre no solo el nombre de la tarea, sino también a qué proyecto pertenece?**

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

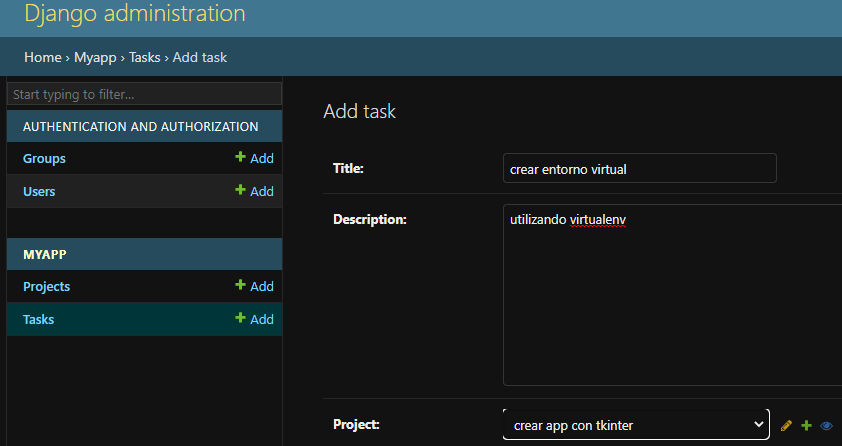
        return self.title + '-->' + self.project.name

        #devuelve los nombres de las tareas y luego a que proyecto pertenecen

#desde la función Project mostramos el name

**Como crear Tareas dentro de una clase en Django Admin**

Vamos a la sección de Tareas (Task) y allí agregamos las tareas que deseemos

**Nos pedirá el titulo de la tarea, una descripción (opcional) y a qué proyecto está ligada la tarea**

Además, **Django admin nos permite renombrar tareas, eliminarlas y demás funcionalidades útiles**

**Si eliminamos un proyecto, también se eliminarán de forma automática las tareas asociadas al mismo, lo cual se llama eliminación en cascada**

**Render**

Hasta ahora solo hemos enviado strings al navegador a través del HttpResponse, pero eso se haría difícil de mantener en páginas grandes

Para eso **usaremos templates o plantillas para enviar archivos HTML**

En primer lugar **creamos una carpeta llamada templates (es el nombre reservado que debemos poner) dentro de la carpeta de la app**. **En ella podemos enviar archivos HTML**

Dentro de la carpeta templates crearemos a modo de prueba una carpeta llamada index**.html**, que se enlazará con la clase del mismo nombre

En esa carpeta pondremos un texto de prueba, en este caso:

<h1>welcome to Django app</h1>

<p>Lorem ipsum, dolor sit amet consectetur adipisicing elit. Tenetur suscipit laboriosam nobis soluta cumque eius voluptatum, ipsum sed minus veniam harum non alias quia quam similique blanditiis! Explicabo, eos placeat.</p>

**Aclaración: si ponemos <p>lorem30</p> nos pondrá un texto random, se suele usar para pruebas como en este caso (el nro que va luego de lorem es para indicar la cantidad de palabras, podemos poner lo que querramos)**

Luego **modificamos la función index del archivo views.py de la app, para que no envíe ya un HttpResponse, sino que nos permita renderizar los archivos HTML que pusimos en el navegador**

Para eso primero **importamos desde django.shortcuts el módulo render**

from django.shortcuts import render

Luego **hacemos que la función index envíe al navegador un render con el archivo HTML:**

def index\_page(request) :

    #enviamos al navegador el archivo index.html para que se muestre al ingresar a esa URL

    return render(request, "index.html")

**Aclaración: debemos pasarle dos parámetros al render, el request y el nombre del archivo que enviaremos al navegador**

**Asi podemos hacer con todas las funciones que querramos**, por ejemplo en about:

Creamos un archivo llamado about.html en la carpeta templates, importamos render desde django.shortcuts (cosa que ya hicimos en el caso anterior) y modificamos la función about para que envíe un render con el archivo HTML

def about(request) :

    #enviamos al navegador el archivo about.html para que se muestre al ingresar a esa URL

    return render(request, "about.html")

**Templates Pass Data (“Las Plantillas Pasan Datos”)**

En la sección anterior enviamos archivos HTML al navegador

Ahora lo que queremos hacer es usar código de backend para poder usar variables que hayamos creado en las funciones y poder **pasarlas junto al archivo HTML hacia el navegador**

**El código de backend nos sirve para pasar datos de una base de datos (database) a los HTML con la finalidad de generar una interfaz**

Para empezar pasaremos los datos manualmente, mas adelante lo haremos directamente desde una base de datos

Ej: crearemos una variable llamada title dentro de la función index\_page en views.py

Queremos que esa variable sea el h1 del archivo index.html, para eso debemos pasarlo desde views.py

**Lo hacemos pasando el dato como parámetro adicional en el return, en la forma de diccionario, los cuales son pares de clave y valor. La clave es el nombre con el que queremos pasar la variable al HTML y el valor es la variable title en sí**

La función quedaría de esta forma:

def index\_page(request) :

    title = "Django Course!!"

    #enviamos al navegador el archivo index.html y además la variable titulo en forma de diccionario

    return render(request, "index.html", {

        'titulo':title

    })

**Luego guardamos y modificamos el HTML (en este caso el index.html), ya que al guardar el paso anterior el HTML ya hizo suya esa variable que le pasamos. Por ende solo queda poner la variable donde deseemos dentro del HTML**

En este caso pondremos la variable titulo como el titulo que le pasaremos al navegador

**Aclaración importante: para usar variables que pasamos al HTML debemos usar llaves dobles {{ }}, sino no nos lo tomará como tal**

El archivo index.html quedaría de esta forma:

<h1>{{titulo}}</h1>

<p>Lorem ipsum, dolor sit amet consectetur adipisicing elit. Tenetur suscipit laboriosam nobis soluta cumque eius voluptatum, ipsum sed minus veniam harum non alias quia quam similique blanditiis! Explicabo, eos placeat.</p>

**La variable titulo es la que importamos desde la función index\_page de views.py, y se usan doble llaves para enviarla al navegador**

En sí **los archivos HTML que escribimos en Django no son HTML como tal**, ya que **los que nos permiten trabajar con ellos, procesarlos y modificarlos son los Templates Engine (motores de plantilla), los cuales son módulos de Python**

**Gracias a estos módulos se generan los HTML que se envían al cliente o navegador, ya que al navegador le llega el HTML ya procesado gracias a los Templates Engine**

**Por ende podemos usar las variables que querramos en el HTML, ya que serán procesadas por estos motores, y al frontend le llegarán archivos HTML reales**

Ej 2: queremos pasarle al about.html una variable llamada username

**Quedaría de esta forma en views.py:**

def about(request) :

    username = "EmiSchonhals"

    #enviamos al navegador el archivo about.html para que se muestre al ingresar a esa URL

    return render(request, "about.html", {

        'username':username #pasamos como diccionario la variable username para que about.html pueda usarla

    })

**Y el about.html quedaría asi:**

<h1>About</h1>

<p>Lorem ipsum, dolor sit amet consectetur adipisicing elit. Ut obcaecati tempore dolorem, nemo ipsa, autem id facere, odio impedit ab quo sapiente. Non minus omnis quidem veniam quas officia ducimus ipsam porro dicta neque iure illo inventore et mollitia illum quibusdam numquam dolore nam, autem dolorum harum? Necessitatibus aperiam in quod alias, doloremque cumque eveniet ullam veniam, fugit, quos neque assumenda exercitationem maxime veritatis. Alias optio, excepturi accusantium similique error laudantium deleniti nihil animi ullam beatae sequi vero totam quo expedita veritatis dolorem porro, soluta odio hic? Voluptatem, ab corporis voluptas ullam dicta, aspernatur unde voluptates, amet autem reprehenderit laudantium!</p>

<p>Website created for {{username}}</p>

**Jinja Loops (Bucles Jinja)**

Los bucles en Jinja son una característica de la plantilla del motor de Jinja que permiten iterar sobre una colección de elementos o ejecutar un bloque de código repetidamente mientras se cumpla una condición. Los bucles son esenciales ya que permiten automatizar tareas repetitivas y reducir la cantidad de código necesario para realizar una operación.

En Jinja, los bucles se implementan utilizando las palabras clave **for** y **while**. Los bucles **for** permiten iterar sobre una colección de elementos, como una lista, un diccionario o un objeto iterable, y realizar una acción en cada elemento. Por otro lado, los bucles **while** permiten ejecutar un bloque de código repetidamente mientras se cumpla una condición determinada.

Los bucles en Jinja también permiten el uso de filtros y variables, lo que permite una mayor flexibilidad en la manipulación de datos y en la generación de la salida.

**Sintaxis de los Bucles Jinja**

**For:**

**{% for item in items %}**

**{{ item }}**

**{% endfor %}**

**While:**

**{% set i = 0 %}**

**{% while i < 10 %}**

**{{ i }}**

**{% set i = i + 1 %}**

**{% endwhile %}**

Ej: mostraremos en el servidor cada objeto de la clase projects

Para eso creamos un iterable que guarde todos los objetos de la clase Project, y luego pasamos ese iterable en forma de diccionario al projects.html para que allí se itere mediante el bucle For y le llegue al servidor ese HTML procesado

def projects (request) :

    #la variable projects contendrá todos los objetos de la clase Project

    proyectos = Project.objects.all()

    return render(request, "projects.html",{

        #le pasamos la lista projects que creamos al archivo projects.html en forma de diccionario para que lo pueda iterar con el bucle For

        'projects':proyectos

    })

Y en el projects.html ponemos el bucle For que va a ir mostrando el nombre de cada objeto de la clase Project

<h1>Projects</h1>

{% for proyecto in projects %}

<h2>{{proyecto.name}}</h2>

{% endfor %}

De esa forma ya podremos ver en el navegador todos los objetos que contiene la clase Project

**Lo mismo podemos hacer con las tareas, incluso mostrando su descripción y a qué proyecto pertenece cada una,** y además agregaremos dos buttons, uno de borrar tarea y otro de que completamos la misma

def tasks (request) :

    #el iterable tareas contendrá todos los objetos de la clase Task

    tareas = Task.objects.all()

    return render(request, "tasks.html",{

        'tasks':tareas

    })

<h1>Tasks</h1>

{% for tarea in tasks %}

<div>

    <h3>{{tarea.title}}</h3>

    <p>{{tarea.description}}</p>

    <p>Proyecto: {{tarea.project.name}}</p>

</div>

<button>

   delete

</button>

{% endfor %}

**Jinja Conditionals (Condicionales Jinja)**

Los condicionales en Jinja son una característica de la plantilla del motor de Jinja que permiten evaluar una expresión o una variable y ejecutar diferentes bloques de código dependiendo del resultado de la evaluación. En Jinja, los condicionales se implementan utilizando la palabra clave “if”, que permite evaluar una expresión y ejecutar un bloque de código si la expresión es verdadera. También se pueden utilizar palabras clave adicionales, como “elif” y “else”, para especificar bloques de código adicionales que se ejecutan si la evaluación de la expresión no cumple con la condición anterior

Ej:

**{% if user.is\_authenticated %}**

**Bienvenido {{user.name}}!**

**{% else %}**

**Por favor, inicia sesión para continuar.**

**{% endif %}**

En este ejemplo, se evalúa si el usuario está autenticado o no. Si el usuario está autenticado, se muestra un mensaje de bienvenida que incluye el nombre del usuario. Si el usuario no está autenticado, se muestra un mensaje que indica que deben iniciar sesión para continuar

Ahora, **haremos que la sección tasks en el navegador tenga dos botones, uno de tarea completada y otro de delete de forma funcional**

En primer lugar vamos al archivo **tasks.html** y agregamos dos botones, uno de delete y otro de Done, en este caso usaremos símbolos Unicode

<button>

   X Delete

</button>

<button>

    ✅ Done

</button>

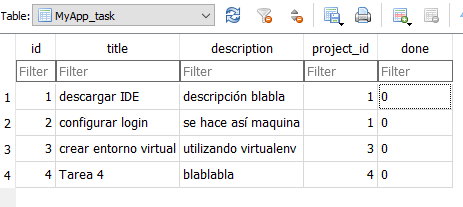
Luego vamos al archivo **Models.py** de la app y agregamos un objeto llamado Done a la clase Task (**el cual será el encargado de mostrar si la tarea está o no terminada, dependiendo de si su valor es True o False**)

#el objeto done servirá para saber si completamos la tarea o no

    #este campo estará en 0 por defecto en la base de datos

    done = models.BooleanField(default=False)

Ahora debemos **ejecutar las migraciones** a través de python manage.py makemigrations y luego python manage.py migrate

**De esta forma creamos una nueva columna u objeto en la clase Task de la base de datos**

**Como pusimos que el objeto done tenga por defecto el valor False, se mostrará con valor 0 por defecto**

Por último **queremos mostrar un string luego del nombre de cada tarea que indique si la tarea está terminada o está pendiente**, para eso volvemos al archivo tasks.html

**Para mostrar el texto indicado dependiendo del valor del objeto Done que tiene cada tarea usaremos el condicional If**

**Preguntamos si Task en su propiedad done es falsa o verdadera para saber qué texto mostrar en el navegador** (de paso usaremos otro if en el nombre de cada tarea para agregar un símbolo Unicode que indique si la misma está en proceso o terminada)

**Si la propiedad done que tiene cada tarea es False mostrará un símbolo Unicode de tiempo en el nombre de la tarea y además de eso mostrará el texto “tarea pendiente”**

**Si la propiedad done que tiene cada tarea es True mostrará un símbolo Unicode de check en el nombre de la tarea y además de eso mostrará el texto “tarea realizada”**

<div>

    <h2>{% if tarea.done == False %} ⏲  {% else %} ✅   {% endif %}

    {{tarea.title}}</h2>

    {% if tarea.done == False %}

    <p>Tarea Pendiente</p>

    {% else %}

    <p>Tarea Realizada</p>

    {% endif %}

    <p>{{tarea.description}}</p>

    <p>Proyecto: {{tarea.project.name}}</p>

</div>

**Aclaración: En el for en el cual tenemos incluido este condicional pusimos for tarea in task, por ende debemos poner tarea.done al evaluar los condicionales, ya que asi definimos el nombre de la variable que irá tomando todos los valores de los objetos de la clase Task**

De esta forma, se mostrará en el navegador el ícono y la frase que corresponda dependiendo de la condición:



**Como hacer comentarios en Jinja o en HTML**

Tenemos dos formas, en Jinja (la cual no se mostrará si ingresamos a la herramienta para desarrollador del navegador) y en HTML (la cual si podremos ver en la herramienta para desarrollador del navegador)

**En Jinja:**

{# esto es un comentario Jinja #}

**En HTML:**

<!--esto es un comentario HTML-->

**Es mejor usar comentarios HTML por que contrastan con el resto del código y podemos verlos más fácilmente. Además de eso podremos verlos si ingresamos a las herramientas para desarrolladores del navegador**

**Template Inheritance (Herencia de Plantilla)**

Nos permite tener una interfaz que puede ser heredada en otra interfaces

**Como crear una lista de navegación en el sitio web**

Por ejemplo, si nosotros queremos **tener una forma de navegar por el sitio sin necesidad de que el usuario vaya escribiendo las diferentes URL´s**, debemos crear una lista de navegación

Para eso usaremos **4 etiquetas principales: <nav>, <ul>, <li> y <a href>**

En HTML, las etiquetas **<nav>**, **<ul>** y **<li>** se utilizan para estructurar y crear listas de navegación en una página web.

La etiqueta **href** en HTML se utiliza dentro de los elementos **<a>** (ancla) para especificar el destino de un enlace o hipervínculo. La abreviatura "href" significa "hipertext reference" (referencia de hipertexto)

**<nav>**: La etiqueta **<nav>** se utiliza para definir una sección de navegación en un documento HTML. Esta etiqueta se utiliza comúnmente para agrupar los enlaces de navegación principales de un sitio web, como una barra de navegación

**<ul>**: La etiqueta **<ul>** se utiliza para crear una lista no ordenada en HTML. "UL" es una abreviatura de "unordered list" (lista no ordenada). Los elementos de la lista se definen utilizando la etiqueta **<li>** (elemento de lista)

**<li>**: La etiqueta **<li>** se utiliza para definir un elemento de lista en HTML. Esta etiqueta se utiliza dentro de una lista ordenada (**<ol>**) o una lista no ordenada (**<ul>**). Cada elemento de lista se muestra generalmente con un marcador o un número, dependiendo del tipo de lista

Ej:

<nav>

    <ul>

      <li><a href="/">Inicio</a></li>

      <li><a href="/acerca">Acerca</a></li>

      <li><a href="/contacto">Contacto</a></li>

    </ul>

  </nav>

**El href será el la URL y luego de ella va el nombre que queremos que se vea en el link**

**href** en HTML se utiliza dentro de los elementos **<a>** (ancla) para especificar el destino de un enlace o hipervínculo. La abreviatura "href" significa "hipertext reference" (referencia de hipertexto).

La sintaxis básica de un enlace con la etiqueta **href** es la siguiente:

**Sintaxis básica de un enlace con la etiqueta href**

**<a href=”URL”> texto del enlace </a>**

**Usos comunes de la etiqueta href**

Enlace a una página web: Puedes utilizar la etiqueta **href** para enlazar a una página web específica proporcionando la URL como valor del atributo **href**. Al hacer clic en el enlace, el navegador redirigirá al usuario a la URL especificada

Ej:

<a href="https://www.ejemplo.com">Visita Ejemplo.com</a>

Enlace interno: Si deseas enlazar a una sección específica de la misma página o a otra sección dentro del mismo sitio web, puedes utilizar enlaces internos. Para ello, debes proporcionar la ID del elemento de destino en el atributo **href**. Al hacer clic en el enlace, el navegador se desplazará hasta el elemento correspondiente

Ej:

<a href="#seccion">Ir a la sección</a>

...

<h2 id="seccion">Sección de ejemplo</h2>

Enlace de correo electrónico: Puedes utilizar la etiqueta **href** para crear enlaces que abran el cliente de correo electrónico del usuario y le permitan enviar un correo electrónico a una dirección específica

<a href="mailto:info@example.com">Contáctanos</a>

EN RESUMEN…**<nav>** se utiliza para agrupar la navegación principal, **<ul>** se utiliza para crear una lista no ordenada, **<li>** se utiliza para definir cada elemento de la lista y **href** se utiliza para definir el destino de un enlace, ya sea a una página web, una sección interna de la página o una dirección de correo electrónico. Estas etiquetas se combinan para crear menús de navegación en HTML

**Como heredar una lista de navegación a los demás archivos HTML**

Es una buena práctica crear una carpeta ***layouts*** dentro de la carpeta templates, para allí poner nuestros archivos de herencia de plantillas

**Lo que haremos será crear una lista de navegación y luego heredarla a cada dirección URL que usemos en el sitio web, para que el usuario pueda moverse por el sitio de una forma rápida y sencilla**

**Crearemos un archivo de base que iremos heredando a cada archivo HTML que querramos que contenga esa lista de navegación que pondremos dentro**

Ej: creamos un archivo base.html dentro de la carpeta layouts

<nav>

    <ul>

        <li>

            <a href="/">Home</a>

        </li>

        <li>

            <a href="/projects">Projects</a>

        </li>

        <li>

            <a href="/tasks">Tasks</a>

        </li>

        <li>

            <a href="/about">About</a>

        </li>

    </ul>

</nav>

**Luego iremos a los archivos HTML que deseemos y le haremos saber que queremos heredar el archivo de base que creamos previamente en la carpeta layouts de templates**

Para heredar se usa la sintaxis **{% extends ´nombre\_de\_archivo\_que\_hereda´ %}**

**Extends se refiere a que heredamos desde el archivo HTML que estamos usando lo que contiene el archivo que pongamos como parámetro**, por poner un ejemplo el archivo ´base.html´

Si queremos heredar al archivo index.html desde base.html en la carpeta layouts ponemos:

<!--desde la carpeta layouts heredamos el archivo base.html con la lista de navegación-->

{% extends "layouts/base.html" %}

**Lo que hace esto es suplantar todo el contenido de esa página por lo heredado en base.html**

Pero… **Como hacemos para que no se nos elimine de la página la demás información que ya tenia o que queremos agregar además de la plantilla heredada?**

**Para eso debemos usar bloques**

**En el archivo de la plantilla que queremos heredar**, en este caso base.html, **debemos indicarle que pondremos un bloque en el cual irán más cosas** (en este caso pondremos que existirá un bloque luego de la lista de navegación). **En el archivo base.html solo pondremos que existirá un bloque, el contenido del mismo lo pondremos en el archivo HTML pertinente, ya que en general cada HTML cumple una función distinta** (en nuestro caso lo haremos en tasks, projects, index y about)

<!--así le indicamos que existirá un bloque de contenido luego de que se muestre la lista de navegación. no debemos poner lo que llevará dentro, eso debemos hacerlo en cada archivo HTML individualmente-->

{% block contenido %}

{% endblock %}

**En este caso lo ponemos luego de la lista de navegación, ya que queremos indicarle que el bloque de contenido irá luego de ella en cada HTML del sitio web**

**Luego vamos al HTML en el cual deseamos heredar la plantilla y ponemos el bloque que irá en su URL (debemos hacer esto con cada HTML, ya que cada uno tendrá un contenido distinto en teoría)**

Por ende, el archivo projects.html (por poner un ejemplo) quedaría asi:

<!--desde la carpeta layouts heredamos el archivo base.html con la lista de navegación-->

{% extends "layouts/base.html" %}

  <!--ponemos en el bloque el contenido que se mostrará en el sitio (en este caso se mostrará dsp de la lista de navegación, ya que en el archivo base.html indicamos que el bloque irá luego de ella)-->

  {% block contenido %}

  <h1>Projects</h1>

   {% for proyecto in projects %}

   <h2>{{proyecto.name}}</h2>

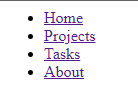
    {% endfor %}

    <!--fin del bloque-->

  {% endblock %}

**Y así con todos los HTML que hayamos agregado a la lista de navegación**

Si lo hacemos de esta forma, **todas las URL´s que tengamos agregadas a la lista de navegación tendrán esto en su vista web de forma funcional:**



**Formularios**

Hasta el momento hemos creado páginas que muestran información, pero ahora queremos que el **usuario nos envíe información. Para eso utilizaremos inputs a través de formularios**

Para eso creamos una nueva ruta que le permita al usuario enviarnos datos

En este ejemplo crearemos en templates un archivo llamado **create\_task.html**, en el cual haremos que el usuario cree sus propias tareas en el sitio web

En **views.py** creamos una función llamada create\_task cuyo parámetro es un request, la cual retornara un Render con el archivo **create\_task.html**

def create\_task (request) :

    return render(request, "create\_task.html")

Luego agregamos en **urls.py** la ruta que usará esa función

#al ingresar a esa direccion le permitiremos al usuario crear su propia tarea

    path('create\_task/', create\_task)

Luego agregamos la ruta a la lista de navegación de **base.html**

<li>

            <a href="/create\_task">Crear tarea</a>

        </li>

Luego heredaremos la lista de navegación de **base.html** en el archivo **create\_task.html**

<!--Heredamos la lista de navegación que está en la carpeta layouts en el archivo base.html-->

{% extends 'layouts/base.html' %}

<!--este bloque será mostrado (en este caso luego de la lista de navegación, ya que así lo pusimos en base.html)-->

{% block contenido %}

<h1>Crea tu nueva tarea</h1>

{% endblock %}

**Dentro de este bloque creamos los formularios que necesitemos para que el usuario nos envíe información**

**Como crear formularios**

Para eso tenemos dos formas:

Forma básica: usando **input** y **textarea**

<input type="text">

<textarea name="descripcion" id="" cols="30" rows="10"></textarea>

**Usando Django:**

En primer lugar dentro de la carpeta de mi app creamos un archivo llamado **forms.py**, y en el mismo **importamos desde Django el módulo Forms**

from django import forms

Este modulo **permite que de una clase podamos hacer un formulario de HTML**

Luego creamos una clase llamada **CreateNewTask** que tendrá dos propiedades: **title** y **description**

#creamos una clase CreateNewTask para que sea el formulario, y le pasamos dos propiedades: title y description

#como parámetro le pesamos desde forms la propiedad Form que viene por defecto

class CreateNewTask (forms.Form) :

    #al enviarlos los transformará en un HTML de tipo texto

    title = forms.CharField(label="Título de la tarea", max\_length=200)

    description = forms.CharField(label="Descripción de la tarea", required=False, widget=forms.Textarea)

#label es el encargado de que el usuario entienda que debe poner en el formulario

#el parámetro required=False indica que el campo es opcional, el usuario puede no ingresar nada en él y no habrá un problema o error en la página

    #widget=forms.Textarea indica que el chardfield se transformará a textarea, permitiendonos ingresar mas caracteres

**Aclaraciones**

**Qué es un label?:** es un elemento utilizado para asociar texto descriptivo con un elemento de formulario. El propósito principal de un label es proporcionar una etiqueta legible para los usuarios que describa el propósito o la función del elemento de formulario al que está asociado

**Qué significa el parámetro required=False?:** se utiliza al definir un campo en un formulario. Este parámetro indica que el campo no es obligatorio y que se puede dejar en blanco al enviar el formulario. Por ende, el campo puede dejarse vacío sin que se genere un error de validación

Luego vamos a **views.py** e **importamos desde el módulo forms.py la clase CreateNewTask**

#importamos el formulario creado en forms.py

from .forms import CreateNewTask

Además, **agregamos un parámetro más al return de la función create\_task**

def create\_task (request) :

    return render(request, "create\_task.html",{

        'form': CreateNewTask()

})

**De esta forma ya tenemos el formulario listo, solo resta pasárselo al create\_task.html para que lo envíe al sitio web y terminar los últimos retoques**

**Ahora ponemos el formulario dentro de una etiqueta form** en **create\_task.html**

<form>

    <!--mostramos el formulario, en este caso queremos que se muestren como etiquetas p, por ende estarán uno debajo del otro-->

    {{form.as\_p}}

    <button>

        Save

    </button>

</form>

Como el formulario se está mostrando mediante la función create\_task, los datos que envíen los usuarios se guardarán allí

**Para guardar los datos en donde querramos debemos hacer que guarde tanto el valor del título, descripción y nombre del proyecto al que queremos que pertenezca la tarea**.

Para el **titulo y descripción** usamos un **request** ya que **nos lo enviará el usuario**

def create\_task (request) :

    #si el usuario nos visita desde el metodo GET(ingresando los datos desde la URL y no desde el mismo formulario) renderizaremos la interfaz

    if request.method == 'GET' :

        return render(request, "create\_task.html",{

        'form': CreateNewTask()

    })

    #si nos visita por el metodo POST(ingresando los datos desde el formulario y no desde la URL) guardaremos los datos en el projectkey que le ponemos

    else :

        #obtenemos los valores de title y description e indicamos al final en que proyecto se crearán, y los enviamos automáticamente a la base de datos de ese proyecto

        Task.objects.create(title = request.POST['title'], description = request.POST['description'], projectkey = 2)

        #cuando el usuario complete el formulario se lo redireccionará a la URL de tasks para que pueda ver todas las tareas creadas y el estado de las mismas

**Luego que el usuario complete el formulario, queremos que lo redirija a otra URL dentro de nuestro sitio**, en este caso a la parte task para que pueda ver las tareas existentes

Para eso en primer lugar **agregamos el método redirect en shortcuts**

from django.shortcuts import get\_object\_or\_404, render, redirect

Y luego hacemos que en el Else de la función create\_task (**si el usuario usa el método POST e ingresa los datos desde el formulario y no desde la sección de URL) retorne un redirect hacia la parte de tasks creadas**

else :

        #obtenemos los valores de title y description e indicamos al final en que proyecto se crearán, y los enviamos automáticamente a la base de datos de ese proyecto

        Task.objects.create(title = request.POST['title'], description = request.POST['description'], project\_id = 1)

        #cuando el usuario complete el formulario se lo redireccionará a la URL de tasks para que pueda ver todas las tareas creadas y el estado de las mismas

        return redirect('/tasks/')

**En este caso los datos que el usuario ingrese se guardaran en el proyecto con id = 1, ya que asi lo pusimos**

Por defecto Django nos envía un error llamado **“CSRF verification failed”**, que es para evitar que copien nuestro sitio y envíes datos desde otro lugar. Para arreglarlo debemos poner dentro de la etiqueta form:

<!--para arreglar el error por defecto de Django al ingresar los datos -->

    {% csrf\_token %}

**De esta forma, si entramos al DB Browser for SQL podremos ver que la base de datos se habrá actualizado con las tareas que agregue el usuario**

**Formularios POST**

**En la sección anterior creamos un formulario que nos permite crear tareas, ahora en esta sección crearemos un formulario que nos permita crear proyectos**

Para tener una **mejor organización** crearemos dos carpetas dentro de la carpeta templates: una de tasks y otra de projects, y **pondremos cada archivo templates en su respectiva carpeta** (para eso también debemos actualizar las rutas que devuelven las views de cada función que hayamos movido a esas respectivas carpetas, ya que ahora hay archivos que están dentro de una de las dos subcarpetas)

**Para crear un formulario que le permita al usuario crear proyectos a través del navegador**, empezaremos creando un archivo dentro de la carpeta projects que se llama **create\_project.html** y una vista llamada **create\_project** que nos devolverá un render

def create\_project(request) :

    return render(request, 'projects/create\_project.html')

Luego **agregamos la URL** al archivo **urls.py**

#al ingresar a esa direccion le permitiremos al usuario crear su propio proyecto

    path('create\_project/', create\_project)

Agregamos la **etiqueta <li>** en **base.html**

<li>

            <a href="/create\_project">Crear proyecto</a>

        </li>

Creamos el **formulario CreateNewProject** en **forms.html**

#creamos el formulario CreateNewProject para que el usuario pueda crear sus proyectos

class CreateNewProject (forms.Form) :

    #en este caso el modelo de proyectos tienen sólo un name

    name = forms.CharField(label="Nombre del Proyecto", max\_length=200)

    #label es el encargado de que el usuario entienda que debe poner en el formulario

En **create\_project.html** ponemos

{% extends 'layouts/base.html' %}

{% block contenido %}

<!--creamos un formulario con el método POST-->

<form method="POST">

    <!--SIEMPRE debemos poner {% csrf\_token %} para evitar errores del navegador-->

    {% csrf\_token %}

    {{form}}

    <button>

        Save

    </button>

</form>

{% endblock %}

Modificamos la **vista** **create\_project**

def create\_project(request):

    # si el usuario nos visita desde el metodo GET(ingresando los datos desde la URL y no desde el mismo formulario) renderizaremos la interfaz pero sin guardar los datos

    if request.method == 'GET':

        return render(request, 'projects/create\_project.html', {

            'form': CreateNewProject()

        })

    # si nos visita por el metodo POST(ingresando los datos desde el formulario y no desde la URL) también mostraremos la interfaz pero guardaremos los datos en la base de datos

    else :

        #en el caso de esta app, los proyectos solo poseen un name

        Project.objects.create(name = request.POST["name"])

        return render(request, 'projects/create\_project.html', {

            'form': CreateNewProject()

        })

**Por último nos falta redireccionar al usuario a la página de projects para que una vez ingrese un proyecto nuevo pueda ver todos los proyectos existentes**

En la sección anterior vimos una forma de hacerlo **(a través del nombre de la URL a la que queremos redireccionarlo)**, en la siguiente sección veremos otra forma

**URL Names & Redirect**

En esta sección aprenderemos a como nombrar nuestras URL’s para evitar errores cuando tengamos una app más extensa

**Como nombrar nuestras URL’s para evitar errores**

En **urls.py** tenemos todas las URL’s que vamos agregando a la app, pero si queremos actualizarlas (por ejemplo cambiándole el nombre de la dirección) nos generará errores en las páginas que dependan de esa dirección, ya que las programamos para que se direccionen hacia una ruta que ya no estaría existiendo

Para evitar esto **agregamos un parámetro a las URL’s** de **urls.py** llamado **name = “nombre\_de\_url”**

**Lo que hace este parámetro es que ya no llamemos a las URL´s por su dirección, sino que las llamemos por su name (o identificador), permitiéndonos cambiar la dirección URL cuando querramos sin que afecte a las redirecciones que hacen las demás páginas hacia ella**

La lista **urlpatterns** quedaría de esta forma:

urlpatterns = [

    #cuando el usuario ingrese al link de la página llamaremos automáticamente a la función index\_page

    path('', index\_page, name = "index"),

    #cuando el usuario ingrese a la pagina/about llamará a la función

    path('about/', about, name = "about"),

    #creamos una ruta que cuando el usuario ingrese a ella y le agregue su username, nos quede guardado su user

    #y también lo salude usando ese username

    path('hello/<str:username>', hello, name = "hello"),

    path('projects/', projects, name = "projects"),

    #cuando ingresemos a la URL y pongamos la direccion tasks/nro\_de\_ID nos retornará el nombre del objeto (en este caso la tarea)

    path('tasks/', tasks, name = "tasks"),

    #cuando ingresemos a la URL y pongamos la direccion tasks/nombre\_de\_tarea nos retornará el id del objeto (en este caso tarea)

    path('tasks/<str:title>', tasks\_nombre, name = "tasks\_title"),

    #al ingresar a esa direccion le permitiremos al usuario crear su propia tarea

    path('create\_task/', create\_task, name = "create\_task"),

    #al ingresar a esa direccion le permitiremos al usuario crear su propio proyecto

    path('create\_project/', create\_project, name = "create\_project")

]

Por último **falta modificar la forma que tenemos de redireccionar a estas URL´s desde el archivo views.py para que cada vista retorne un redirect con el name de su dirección URL correspondiente y también desde el archivo base.html**

El **base.html** quedaría asi:

<nav>

    <ul>

        <li>

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

        </li>

        <li>

            <a href="{% url 'projects' %}">Projects</a>

        </li>

        <li>

            <a href="{% url 'tasks' %}">Tasks</a>

        </li>

        <li>

            <a href="{% url 'about' %}">About</a>

        </li>

        <li>

            <a href="{% url 'create\_task' %}">Crear tarea</a>

        </li>

        <li>

            <a href="{% url 'create\_project' %}">Crear proyecto</a>

        </li>

    </ul>

</nav>

Por último debemos actualizar las views de ser necesario

**Static Files (Archivos Estáticos)**

Hasta este momento hemos estado enviando archivos dinámicos al servidor, ya que el servidor consulta a la base de datos, genera el HTML y luego lo envía al cliente. Por ende trabaja con contenido dinámico

Pero ahora veremos cuáles son los contenidos estáticos, es decir que **no cambian dinámicamente**

**Como cargar los Archivos Estáticos en Django**

Supongamos que queremos cargar una imagen en el index de nuestra app web

**Si vamos a la carpeta de mi sitio al archivo settings.py, e ingresamos a la URL que está bajo el nombre de Static files, nos enviará a la documentación de Django sobre cómo subir imágenes a la app web**

**Django nos permite crear en cada app que creemos una carpeta Static, la cual nos permitirá crear archivos estáticos, como por ejemplo imágenes o audio, y cargarlos en archivos de Jinja** (los que venimos usando con la extensión .html)

En primer lugar **creamos en la app una carpeta llamada Static**, **metemos la imagen dentro de la carpeta del proyecto** que estamos usando **y la pegamos luego en la carpeta Static que acabamos de crear**

Luego, como queremos que la imagen aparezca en la página index, vamos a index.html y **creamos una etiqueta img** donde escribiremos el nombre de la imagen con su extensión (ya sea **.jpg** o **.png**)

Además antes del block pero después del extends debemos poner **{% load static %}**

<!--desde la carpeta layouts heredamos el archivo base.html con la lista de navegación-->

{% extends "layouts/base.html" %}

<!--para poder cargar la imagen que va dentro del block, debemos poner en este lugar el {% load static %}-->

  {% load static %}

  {% block contenido %}

  <h1>Home</h1>

  <!--insertamos la imagen que deseamos que se muestre en la pantalla de inicio de la app-->

  <img src="{% static 'emi.jpg' %}" alt="">

  <!--fin del bloque-->

  {% endblock %}

**Como ajustar una imagen al tamaño que deseemos**

**Por defecto si subimos una imagen con la extensión .png la misma se verá bastante grande en la web, y con las imágenes con extensiones .jpg puede ocurrir lo contrario**

Para ajustarla al tamaño que deseemos solamente basta con **agregarle el parámetro** **width = “valor”** a la etiqueta **img**

<img src="{% static 'emi.jpg' %}" alt="" width="200">

<img src="{% static 'bombonera.png' %}" alt="" width="500">

En este ejemplo tenemos dos imágenes, emi.jpg con un tamaño de 200 y otra llamada bombonera.png con un tamaño de 500

**Mejoras Finales: Estilos y Funcionalidad**

Para mejorar los estilos visuales de la web debemos meternos en el frontend

………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………….

(Para saber esto hay que ver la última sección del curso de Django para principiantes de Fazt)