Apps en Django

Cada app de Django es independiente, contiene sus propias definiciones, templates y ficheros estáticos.

El caso es que necesitamos crear como mínimo una app para empezar a trabajar y como vamos a estar desarrollando un pequeño **blog** de prueba podríamos crear esta app con ese mismo nombre:

```
pipenv run python manage.py startapp blog
```

Como véis se crea un directorio con el nombre de la app en nuestro proyecto y dentro hay un montón de nuevos ficheros, los iremos descubriendo poco a poco.

Por ahora sólo nos falta activar la app en el **settings.py** para poder utilizarla:

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

Modelos en Django

En esta lección introduciremos los modelos de Django, unas clases donde se define la estructura de los datos para almacenarlos en la base de datos.

Nosotros vamos a crear un **blog** muy sencillo así que tendremos que almacenar las entradas. Estas entradas estarán formadas por varios campos, por ejemplo un título y un contenido:

```
blog/models.py
from django.db import models

class Post(models.Model):
    title = models.CharField(max_length=200)
    content = models.TextField()
```

Una vez tenemos el modelo, que es como la estructura de una tabla en la base de datos, tendremos que crear una migración con un registro de los cambios:

```
pipenv run python manage.py makemigrations
```

Una vez hecha la migración tenemos que aplicar los cambios en la base de datos:

```
pipenv run python manage.py migrate
```

Al ser la primera vez que migramos y al tener activadas las apps genéricas del admin, de autenticación y de sesiones se van a crear un montón de campos en la base de datos.

La base de datos la encontraremos en la raíz del proyecto con el nombre **db.sqlite3**. Este fichero se puede consultar y editar con un programa como DB Browser for SQLite. Dentro encontraremos todas las tablas de la base de datos y sus registros.

Sea como sea ya tenemos el modelo **Post** listo para empezar a añadir registro

Administrador en Django

Existen varias formas de manejar los registros de la base de datos:

- A través del código al responder una petición.
- A través del panel de administrador autogenerado.
- A través de la shell de Django, un intérprete de comandos.

En este pequeño curso no veremos la primera forma, nos limitaremos a utilizar el administrador y experimentar un poco con la shell.

Así que veamos como utilizar el panel de administrador de Django.

Para acceder simplemente tendremos que acceder a la URL **/admin**. Esta dirección no es casual, está definida así dentro del fichero **urls.py** de nuestro proyecto.

```
http://127.0.0.1:8000/admin/login/?next=/admin/
```

Nos pedirá identificarnos un usuario y una contraseña. Este usuario no puede ser un cualquier, debe ser un usuario con permisos para acceder al administrador y como no tenemos ninguno vamos a tener que crearlo.

Para crear nuestro primer superusuario lo haremos desde la terminal:

```
pipenv run python manage.py createsuperuser
```

Una vez creado y con el servidor en marcha podremos identificarnos a acceder al administrador, donde encontraremos una sección llamada **Autenticación y autorización**. Esta sección correspondo a la app **auth** de Django y contiene dos modelos, uno para manejar grupos de permisos y otro con los propios usuarios.

Como véis sin hacer nada ya contamos con un panel donde podemos manejar usuarios cómodamente y añadirles permisos para manejar otras apps. Sin embargo nuestra app **blog** todavía no aparece, eso es porque tenemos que activar los modelos que queramos manejar en el administrador.

Así que vamos a configurar el admin para el modelo **Post** y lo haremos en el fichero **admin.py** de la app **blog**:

```
blog/admin.py
from django.contrib import admin
from .models import Post

admin.site.register(Post)
```

Simplemente con este cambio ya nos aparecerá nuestra app y podremos empezar a crear, editar y borrar nuevos **Post**.

Con esto ya lo tenemos pero antes quiero enseñaros algunas opciones para personalizar los campos que se muestran en los formularios del modelo.

```
blog/models.py
class Post(models.Model):
    title = models.CharField(max_length=200, verbose_name="Título")
    content = models.TextField(verbose_name="Contenido")
```

Con eso podemos mostrar un nombre diferente en los campos y si quisiéramos mostrar un nombre de modelo diferente lo haremos así:

```
class Meta:
    verbose_name = "entrada"
    verbose_name_plural = "entradas"
```

Por último para listar las entradas mostrando su nombre y no esa referencia rara al objeto que aparece, podemos hacerlo sobreescribiendo el método string del modelo:

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

Óbviamente hay mil cosas más que se pueden configurar, pero hablaremos de eso en futuros cursos.

Esto es más que suficiente por ahora, cread un par más de entradas y seguimos.

Shell en Django

La shell de Django es un intérprete de comandos que nos permite ejecutar código directamente en el backend, algo muy útil para hacer pruebas y consultas.

```
pipenv run python manage.py shell
```

Una vez dentro la podremos escribir las instrucciones que queramos siempre que sigamos una lógica.

Por ejemplo, si queremos consultar las entradas de la base de datos antes tendremos que importar el modelo **Post** y luego utilizar la sintaxis para hacer la consulta:

```
from blog.models import Post
Post.objects.all()
```

Esta instrucción **objects.all()** nos permite recuperar todos los registros que hay en la tabla de entradas y los almacena en una lista especial de Django llamada QuerySet.

También podemos recuperar el primer y último registro muy fácilmente:

```
Post.objects.first()
Post.objects.last()
```

O una entrada a partir de su identificador, un número automático que maneja Django internamente:

```
Post.objects.get(id=1)
```

Todo lo que estamos haciendo son consultas a la base de datos, pero se encuentran abstraidas gracias a la API de acceso al mapeador ORM que nos proporciona Django, donde cada registro se puede manejar como un objeto.

De hecho vamos a crear una entrada y a manipularla un poco:

```
post = Post.objects.create(
    title="Otra entrada", content="Texto de prueba")
```

Al ejecutar lo anterior tendremos nuestra entrada creada en la base de datos, podemos consultar el administrador, pero también la tendremos guardada en la variable **post**:

```
post
```

Podríamos editarla simplemente estableciendo el título como si fuera un objeto normal y corriente:

```
post.title = "Otro título diferente"
```

Eso sí, tendremos que llamar al método **save()** del objeto para guardar los cambios:

```
post.save()
```

Por último si quisiéramos borrar esta entrada que tenemos en la variable podemos hacer con el método **delete()**:

```
post.delete()
```

Y para salir de la shell simplemente llamaremos la función **quit()** desde la terminal:

```
quit()
```

Como habéis visto hemos estado interactuando con la base de datos a través de objetos y método, una forma muy sencilla y cómoda de manejar nuestros registros.

Vistas y urls en Django

Todo lo que hemos hecho hasta ahora no ha implicado todavía el manejo de peticiones y respuestas.

Las peticiones son las diferentes URL que los clientes piden ver, por ejemplo **/** es la portada y **/blog/** podría ser nuestro blog.

Para manejar estas peticiones Django utiliza el siguiente flujo:

- El cliente hace la petición a una dirección definida en el **urls.py**.
- Esa dirección está enlazada a una vista, una función definida en el fichero **views.py** y que contiene la lógica que procesará la petición, como por ejemplo consultar el modelo **Post** para ver qué entradas hay en la base de datos.
- Por último esos datos se renderizarán sobre un template HTML y se enviarán al cliente para que éste vea el resultado de la petición.

Este flujo de trabajo se conoce en Django como patrón **MVT** (Modelo - Vista - Template) y nos permite separar muy bien cada proceso de los demás.

Para ver todo esto en acción vamos a programar la vista de nuestra portada y devolviendo un simple texto plano:

```
blog/views.py
from django.shortcuts import render, HttpResponse

def home(request):
    return HttpResponse("Bienvenido a mi blog")
```

Ahora tenemos que enlazar esta vista a una URL para poder hacer la petición:

```
tutorial/urls.py
from django.contrib import admin
from django.urls import path
from blog.views import home

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

Tan simple como esto y si accedemos a la raíz de nuestro sitio ya deberíamos ver como se devuelve el texto plano que devolvemos.

Sin embargo la gracia es renderizar un template HTML bien estructurado, así que vamos a crear uno para nuestra portada.

Prestad mucha atención, para crear un template dentro de la app **blog** tenemos que crear un directorio **templates** en la app y dentro otro directorio llamado con el mismo nombre que la app, en nuestro caso **blog**.

Se hace de esta forma porque Django carga en memoria todos los directorios **templates** de las apps unificándolos en el mismo sitio.

En cualquier caso dentro ya podremos crear nuestra plantilla HTML para renderizarla:

```
templates/blog/home.html
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
```

```
    <title>Blog</title>
</head>
<body>
    <h1>Bienvenidos a mi blog</h1>
</body>
</html>
```

Por último vamos a cambiar la vista para en lugar de devolver el texto plano renderice esta plantilla HTML que hemos creado:

blog/views.py
```
from django.shortcuts import render, HttpResponse

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

Listo, ya hemos completado la parte de la vista y el template. En la siguiente lección incorporaremos una consulta a la base de datos a través del modelo **Post** y devolveremos las entradas al template para renderizarlas.

Variables de contexto

Para recuperar las entradas tendremos que cargar el modelo y hacer exactamente lo que hicimos cuando experientamos en la shell:

blog/views.py
```
from django.shortcuts import render, HttpResponse
from .models import Post

def home(request):
    posts = Post.objects.all()
    return render(request, "blog/home.html")
```

Una vez tenemos las entradas recuperadas tendremos que enviarlas al template y eso lo haremos usando un diccionario de contexto:

```
return render(request, "blog/home.html", {'posts': posts})
```

Los datos que enviamos en el diccionario de contexto se pueden recuperar en el template usando template tags, una de las funcionalidades más atractivas de Django:

templates/blog/home.html
```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
```

```
    <title>Blog</title>
</head>
<body>
    <h1>Bienvenidos a mi blog</h1>
    <p>Estas son las entradas más recientes.</p>
    {{ posts }}
</body>
</html>
```

Lo más increíble de todo es que podemos usar unos template tags especiales que permiten ejecutar lógica de programación en los templates, por ejemplo para recorrer todas las entradas que hay almacenadas en la QuerySet **posts**:

```
{% for post in posts %}
<div>
    <h2>{{ post }}</h2>
</div>
{% endfor %}
```

Por defecto se nos muestra el título porque es lo que devolvemos al sobcreescribir el método string del modelo, pero lo que tenemos son objetos por lo que podríamos acceder a sus campos específicos:

```
{% for post in posts %}
<div>
    <h2>{{ post.title }}</h2>
    <p>{{ post.content }}</p>
</div>
{% endfor %}
```

Con esto hemos completado el flujo del patrón Modelo - Vista - Template, recuperamos los datos del modelo y los enviamos al template a través de la vista.

Django se basa siempre en esa idea.

Páginas dinámicas

En esta última lección vamos a introducir el concepto de las páginas dinámicas añadiendo una nueva página para visualizar las entradas individualmente en lugar de mostrar todo su contenido en la lista.

La clave está en pasar un valor en la URL a través del cual podamos recuperar el registro para renderizarlo. ¿Cuál es ese valor? Pues normalmente será el identificador único del registro (su campo **id**) al que también se puede acceder con el nombre **pk** de primary key:

`blog/views.py`

```
def post(request):
    post = Post.objects.get(id=?)
    return render(request, "blog/post.html", {'post': post})
```

Hemos creado la vista y cargaremos el template **post.html** enviándole el objeto **post** en el diccionario de contexto, pero nos falta lo más importante, una forma de recuperar el identificador de la entrada.

Esto se maneja en dos partes, primero en la URL definiendo un parámetro:

```
tutorial/urls.py
from django.contrib import admin
from django.urls import path
from blog.views import home, post

urlpatterns = [
    path('', home),
    path('blog/<id>', post),
    path('admin/', admin.site.urls),
]
```

Y luego en la vista creando ese parámetro como si formara parte de la función:

```
blog/views.py
def post(request, id):
    post = Post.objects.get(id=id)
    return render(request, "blog/post.html", {'post': post})
```

Con esto ya lo tenemos, aunque si accedemos nos dará error de template no encontrado porque todavía no lo hemos creado:

```
templates/blog/post.html
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>{{post.title}}</title>
</head>
<body>
    <h1>{{post.title}}</h1>
    <p>{{post.content}}</p>
    <a href="/">Volver a la portada</a>
</body>
</html>
```

Ya sólo tenemos que añadir un enlace a los títulos para movernos a las entradas:

blog/templates/blog/home.html

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Blog</title>
</head>
<body>
    <h1>Bienvenidos a mi blog</h1>
    <p>Estas son las entradas más recientes.</p>
    {% for post in posts %}
    <div>
        <h2>{{ post.title }}</h2>
        <a href="/blog/{{post.id}}">Leer más</a>
    </div>
    {% endfor %}
</body>
</html>
```

Nuestra web ahora tendrá tantas páginas como entradas tengamos en el blog pero nosotros solo hemos creado la plantilla base. O lo que es lo mismo, tenemos una web generada dinámicamente a partir de los registros que hay en la base de datos.