Tutorial Django Inicial
Jordi Virgili Gomà

Índice

En este tutorial, crearemos una aplicación de librería que permita almacenar información sobre libros, así como reviews de estos. Los libros tendrán información sobre su autor, género, número de páginas y título.

Antes de empezar, asegúrate de tener Python 3 y Django 3.2 o posterior instalados en tu sistema.

Paso 1: Crear un proyecto Django

Para crear un proyecto Django, abre la terminal y escribe el siguiente comando:

```
django-admin startproject nombre_del_proyecto
```

Reemplaza `nombre_del_proyecto` por el nombre que desees darle a tu proyecto. Este comando creará un directorio con el nombre que hayas especificado, que será el directorio principal de tu proyecto.

Necesiaremos un superadmin, para tareas de mantenimiento:

```
python manage.py createsuperuser
```

Paso 2: Crear una aplicación dentro del proyecto

Dentro del directorio principal de tu proyecto, crea una nueva aplicación con el siguiente comando:

```
python manage.py startapp nombre_de_la_aplicacion
```

Reemplaza `nombre_de_la_aplicacion` por el nombre que desees dar a tu aplicación. Este comando creará un nuevo directorio con el nombre que hayas especificado, que será el directorio de tu aplicación.

Una vez que tengas el proyecto y la aplicación creados, puedes continuar con la definición de los modelos y la creación de las vistas y plantillas.

Esta aplicación se debe dar de alta en el fichero settings.py, en caso de que `nombre_de_la_aplicación sea library, la lista de INSTALLED_APPS quedaría:`

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

Paso 3: Definir el modelo de la aplicación

En este paso, definiremos los modelos de Libro y Revisión para almacenar la información en la base de datos. Los modelos en Django son similares a las tablas en una base de datos relacional y representan los datos con los que trabajará la aplicación.

Abre el archivo `models.py` en la carpeta de la aplicación y agrega el siguiente código:

```
from django.db import models

class Libro(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=100)
    genre = models.CharField(max_length=100)
    pages = models.IntegerField()

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

En este código, estamos creando un modelo de libro que tendrá un título, autor, género y número de páginas, todos ellos como campos de texto (`CharField`) o números enteros positivos (`PositiveIntegerField`).

Paso 4: Definir el modelo de la review

Continúa abierto el archivo `models.py` y agrega el siguiente código para definir el modelo de review:

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

class Review(models.Model):
    book = models.ForeignKey(Libro, on_delete=models.CASCADE)
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    content = models.TextField()
    score = models.IntegerField()

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

En este código, estamos creando un modelo de review sobre un libro que tendrá un contenido y puntuación, y relaciones hacia el libro y el usuario que ha creado la review.

Explicación de los componentes:

- `models.Model`: Este es el modelo principal de Django y todos los modelos de nuestra aplicación deben heredar de él.
- `models.CharField`: Este tipo de campo se utiliza para almacenar cadenas de caracteres de tamaño limitado, como el título, autor y género de los libros.
- `models.IntegerField`: Este tipo de campo se utiliza para almacenar valores enteros, como el número de páginas de los libros.
- `models.TextField`: Este tipo de campo se utiliza para almacenar texto largo, como el contenido de las reseñas.

- `models.ForeignKey`: Este tipo de campo se utiliza para crear una relación entre los modelos. Por ejemplo, en el modelo de Revisión, book es una clave foránea que se relaciona con el modelo de Libro y `user` es una clave foránea que se relaciona con el modelo de usuario.
- `models.CASCADE`: Este argumento especifica la acción que se realizará cuando se elimine un objeto relacionado. En este caso, cuando se elimine un libro, todas sus reseñas también serán eliminadas.
- `__str__`: Este método se utiliza para definir una representación en cadena de texto para cada objeto del modelo.

Para poder añadir y quitar objetos de la base de datos, tenemos la interfaz de administración. Pero los elementos no aparecerán solos en el panel de control una vez iniciada la aplicación. Para ello debemos registrar los modelos:

Para registrar los modelos en el archivo `admin.py`, deberás realizar los siguientes pasos:

```
from .models import Libro, Review
```

Crea una clase admin para cada modelo, que se utilizará para personalizar cómo se muestra el modelo en el sitio de administración de Django:

```
class LibroAdmin(admin.ModelAdmin):
    list_display = ('title', 'author', 'genre', 'pages')


class ReviewAdmin(admin.ModelAdmin):
    list_display = ('book', 'user', 'content', 'score')
```

Registra cada modelo con su clase admin correspondiente usando el método `admin.site.register`:

```
admin.site.register(Libro, LibroAdmin)
admin.site.register(Review, ReviewAdmin)
```

Ahora, deberías poder ver los modelos en el sitio de administración de Django y agregar, editar o eliminar instancias de los modelos.

Paso 5: Creación de las tablas en la base de datos

En este paso, haremos que Django cree las tablas en la base de datos correspondientes a los modelos que hemos definido.

Ejecuta el siguiente comando en la línea de comandos dentro del directorio del proyecto:

```
python manage.py makemigrations
```

Este comando creará un archivo de migración en la carpeta `migrations` de la aplicación, que describe los cambios que se deben realizar en la base de datos.

A continuación, ejecuta el siguiente comando para aplicar las migraciones y crear las tablas en la base de datos:

```
python manage.py migrate
```

Explicación de los componentes:

- `makemigrations`: Este comando genera un archivo de migración que describe los cambios que deben realizarse en la base de datos.
- `migrate`: Este comando aplica las migraciones y crea las tablas en la base de datos.

Con estos dos comandos, podemos crear y mantener la estructura de la base de datos en sincronía con los modelos de nuestra aplicación. Cada vez que hagamos cambios en los modelos, podemos generar una nueva migración y aplicarla para actualizar la base de datos.

Paso 6: Creación de formularios

En este paso, crearemos los formularios que nos permitirán ingresar y editar información en la base de datos.

- Creación de formularios para los modelos

En Django, podemos crear formularios a partir de los modelos. Para hacer esto, debemos crear una clase en la aplicación que herede de `forms.ModelForm`. Esta clase describirá el formulario y sus campos, y estará asociada con un modelo específico.

Aquí está un ejemplo de un formulario para el modelo Book:

```
from django import forms
from .models import Book

class BookForm(forms.ModelForm):
    class Meta:
        model = Book
        fields = ['title', 'author', 'genre', 'pages']
```

- Integración de formularios en las vistas

Una vez que tenemos el formulario creado, podemos utilizarlo en nuestras vistas. En una vista, podemos renderizar el formulario en una plantilla HTML para que el usuario pueda ingresar o editar información.

Aquí está un ejemplo de código de una vista que muestra un formulario para crear un nuevo libro:

```
from django.shortcuts import render, redirect
from .forms import BookForm

def create_book(request):
    if request.method == 'POST':
        form = BookForm(request.POST)
        if form.is_valid():
            book = form.save()
            return redirect('book_detail', pk=book.pk) # Redirigir al usuario a
```

```
detalles del libro
    else:
        form = BookForm()
    return render(request, 'create_book.html', {'form': form})
```

Explicación de los componentes:

- `forms.ModelForm`: Es una clase de Django que nos permite crear formularios a partir de modelos.
- `Meta`: Es una clase interna en la que se define el modelo al que está asociado el formulario y los campos que deben incluirse en el formulario.
- `form.is_valid()`: Es un método en el formulario que se utiliza para validar los datos ingresados por el usuario. Si los datos son válidos, se guardan en la base de datos.
- Usamos `redirect` para redirigir a la vista `book_detail` con el argumento pk que contiene el valor de la clave primaria del libro recién creado. Esto redirigirá al usuario a la página de detalles del libro que acaba de crear.
- `render`: Es una función de Django que renderiza una plantilla HTML y le pasa contexto (datos) para ser utilizados en la plantilla. En este ejemplo, estamos pasando el formulario a la plantilla.

Paso 7: Generación de plantillas HTML con formulario

Creación de plantillas HTML

En este paso, crearemos las vistas que controlan cómo se manejarán las solicitudes HTTP y se renderizarán las plantillas HTML.

- Funciones de vista

Las vistas en Django son funciones que manejan las solicitudes HTTP y renderizan las plantillas HTML. Cada vista está asociada con una URL específica y puede tener lógica para manejar diferentes tipos de solicitudes (GET, POST, etc.).

- Manejo de formularios

Las vistas pueden manejar formularios HTML para capturar y procesar información del usuario. En Django, podemos crear formularios utilizando formularios de Django, que proporcionan validación y protección contra ataques en el lado del servidor.

- Redireccionamiento

Las vistas pueden redirigir a los usuarios a otras páginas después de que se haya completado una acción (por ejemplo, después de crear un nuevo libro). En Django, podemos redirigir a los usuarios utilizando la función `redirect`.

- Renderizado de plantillas

Las vistas pueden renderizar plantillas HTML para mostrar información dinámica al usuario. En Django, podemos renderizar plantillas utilizando la función `render`. La función `render` toma una plantilla HTML y un contexto (datos) y devuelve una respuesta HTTP con el HTML renderizado.

En este paso, crearemos las plantillas HTML que se mostrarán al usuario cuando visite nuestras páginas web.

- Creación de archivos HTML

En Django, podemos crear plantillas HTML utilizando un lenguaje de marcado similar a HTML. La diferencia es que podemos incluir variables y lógica en las plantillas para mostrar información dinámica.

Aquí está un ejemplo de una plantilla HTML para crear un nuevo libro:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Crear Libro - LIBRARY</title>
</head>
<body>
<form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Create">
</form>

</body>
</html>
```

Explicación de los componentes:

- `{% csrf_token %}`: Es una etiqueta de Django que protege las solicitudes de formularios contra el ataque de Cross-Site Request Forgery (CSRF).
- `{{ form.as_p }}`: Es un método en el formulario que renderiza el formulario en HTML con los campos envueltos en etiquetas `<p>`.
- `input type="submit"`: Es un elemento HTML que envía el formulario al servidor cuando se hace clic en el botón "Create".
- Integración de plantillas en las vistas

Una vez que tenemos nuestra plantilla HTML creada, podemos utilizarla en nuestras vistas. En una vista, podemos renderizar la plantilla y pasarle contexto para que se muestre información dinámica.

Paso 7: Generación de plantillas HTML (Book detail)

El paso 7 consiste en crear una plantilla HTML que será utilizada para mostrar los detalles del libro. La plantilla es un archivo que define el formato y el contenido de la página web que se mostrará al usuario.

Es necesaria la creación de una vista que alimentará la página de detalles del libro. Esta vista será responsable de recuperar los detalles del libro específico a partir de la base de datos y proporcionarlos a la plantilla HTML que se encargará de mostrarlos.

Aquí hay un ejemplo de código de la vista que se utilizará para mostrar los detalles del libro:

```
from django.shortcuts import render, get_object_or_404
from .models import Book

def book_detail(request, pk):
    book = get_object_or_404(Book, pk=pk)
    return render(request, 'book_detail.html', {'book': book})
```

En este ejemplo, la vista `book_detail` utiliza la función `get_object_or_404` para recuperar un objeto Book a partir de su identificador (pk). Si el libro no existe, se devuelve una respuesta 404. De lo contrario, se utiliza la función `render` para mostrar la plantilla HTML `library/book_detail.html` y proporcionar los detalles del libro en un contexto llamado book.

La plantilla HTML correspondiente se almacenará en el archivo: `templates/book_detail.html` y tendrá acceso a los detalles del libro a través del contexto proporcionado por la vista.

Un ejemplo de una plantilla para mostrar los detalles de un libro en la aplicación de la librería es el siguiente:

```
<h1>{{ book.title }}</h1>
<p>Autor: {{ book.author }}</p>
<p>Género: {{ book.genre }}</p>
<p>Número de páginas: {{ book.pages }}</p>
<h2>Comentarios</h2>
{% for review in book.review_set.all %}
  <p>{{ review.content }}</p>
  <p>Escrito por {{ review.user.username }}</p>
{% endfor %}
```

En esta plantilla, utilizamos la sintaxis de Django para incluir dinámicamente los detalles del libro y las revisiones asociadas al libro. La variable book se pasa a la plantilla desde la vista, y luego utilizamos esta variable para mostrar el título, autor, género y número de páginas del libro. También utilizamos un bucle `for` para mostrar cada una de las revisiones asociadas al libro.

El uso del método "_set" en este código se refiere a la relación de muchos a uno entre las clases "Book" y "Review". En Django, cuando tienes una relación de muchos a uno, puedes acceder a los objetos relacionados desde el objeto principal mediante el uso de "_set".

En este caso, la clase "Book" es el objeto principal y la clase "Review" es la relación de muchos a uno. El código está iterando sobre "review_set.all", lo que significa que está recuperando todas las revisiones asociadas con un libro específico y las está mostrando en la página de detalles del libro.

Cada vez que accedes a "review_set", estás accediendo a un conjunto de objetos "Review" relacionados con el objeto "Book" actual. Y "all" es un método que recupera todos los objetos del conjunto (se podría filtrar también con el método filter()).

Paso 8: Reviews

Tenemos ya un model review. Podemos crear un formulario con el módulo `forms` de Django:

```
from django import forms
```

```
from .models import Libro, Review

class ReviewForm(forms.ModelForm):
    class Meta:
        model = Review
        fields = ['content', 'score']
```

Este formulario se utilizará en la vista para procesar los datos enviados por el usuario y crear una nueva revisión en la base de datos.

Finalmente, puedes crear una nueva vista en tu archivo `views.py` para manejar la creación de nuevas revisiones:

```
from django.shortcuts import render, redirect

def create_review(request, pk):
    book = get_object_or_404(Libro, id=pk)
    # lo mismo que book = Libro.objects.get(id=pk) pero con control de fallos si
no lo encuentra.
    if request.method == 'POST':
        form = ReviewForm(request.POST)
        if form.is_valid():
            review = form.save(commit=False)
            review.book = book
            review.user = request.user
            review.save()
            return redirect('book_detail', pk=book.id)
    else:
        form = ReviewForm()
    return render(request, 'create_review.html', {'form': form, 'book': book})
```

En esta vista, se comprueba si se ha enviado una solicitud POST y, en caso afirmativo, se valida el formulario y se crea una nueva revisión en la base de datos. Si la solicitud es una solicitud GET, simplemente se renderiza el formulario vacío. Finalmente, se redirige al usuario a la página de detalles del libro. Esta vista maneja tanto el procesamiento de los datos enviados a través del formulario como la presentación del formulario para que los usuarios puedan ingresar sus revisiones. La vista toma como argumento un `book_id`, que se utiliza para recuperar el libro específico para el que se está agregando la revisión.

Si el método HTTP es POST, significa que se están enviando datos a través del formulario. En ese caso, se crea una instancia del formulario `ReviewForm` con los datos del `request.POST`. Si el formulario es válido, se crea una nueva revisión, se asigna al libro correcto y se guarda en la base de datos. Finalmente, se redirige al usuario a la página de detalles del libro.

Si el método HTTP es GET, significa que el usuario está solicitando una página para ingresar una revisión. En ese caso, se crea una nueva instancia del formulario `ReviewForm` y se proporciona al usuario una página HTML para ingresar su revisión. La página HTML se renderiza utilizando la plantilla `create_review.html`.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
```

```
    <title>Crear Review - LIBRARY</title>
</head>
<body>
<h2>Agregar un nuevo comentario</h2>
  <form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">Guardar</button>
  </form>

</body>
</html>
```

Paso 9: Creación de listado de libros existentes

Crear una nueva vista en el archivo views.py de tu aplicación de libros para motrar un listado de libros existentes en la base de datos:

```
from django.shortcuts import render
from django.views.generic import ListView
from .models import Libro

class BookListView(ListView):
    model = Libro
    template_name = 'book_list.html'
    context_object_name = 'books'
```

archivo HTML llamado book_list.html en tu carpeta de plantillas:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Lista de libros - LIBRARY</title>
</head>
<body>

<h1>Lista de Libros</h1>
  <ul>
    {% for book in books %}
      <li>{{ book.title }}</li>
    {% endfor %}
  </ul>

</body>
</html>
```

aaa

Paso 10: Llamar a las vistas desde la web

Aquí está un ejemplo de código que podría ser utilizado en el archivo `urls.py` de la aplicación para poder acceder a las vistas descritas anteriormente:

```
from django.urls import path
from library.views import create_review, book_detail, BookListView, create_book

app_name = 'library'

urlpatterns = [
    path('', BookListView.as_view(), name='book_list'),
    path('books/<int:pk>/', book_detail, name='book_detail'),
    path('books/create/', create_book, name='book_detail'),
    path('books/<int:pk>/review/create/', create_review, name='review_create'),
    path('admin/', admin.site.urls),
    ]
```

En este ejemplo, se están definiendo tres URL diferentes para manejar la lista de libros, los detalles de un libro y la creación de una revisión. Cada URL está asociada con una vista específica, que se encargará de manejar la lógica y la renderización de la página correspondiente.

Es importante señalar que estos patrones de URL están utilizando captura de parámetros para permitir que se pase información adicional a las vistas, como el identificador (pk) de un libro específico.

Final: Estructura del proyecto Django

Aquí está un ejemplo de la estructura de archivos que podría tener el proyecto de la librería:

```
my_project/
    library/
        migrations/
        templates/
            library/
                book_list.html
                book_detail.html
                review_form.html
        models.py
        forms.py
        views.py
        urls.py
    my_project/
        settings.py
        urls.py
        wsgi.py
        ...
```

En este ejemplo, la aplicación de la librería se llama `my_app` y se encuentra dentro del proyecto llamado `my_project`.

La carpeta `migrations` contiene los archivos necesarios para mantener el estado de la base de datos a través de los cambios en los modelos.

La carpeta `templates` contiene los archivos HTML que se utilizarán para renderizar las vistas de la aplicación de la librería. La carpeta `library` dentro de `templates` contiene los archivos HTML específicos para cada vista en la aplicación.

El archivo `models.py` contiene la definición de los modelos de la aplicación, incluyendo el modelo Book y el modelo `Review`.

El archivo `forms.py` contiene la definición del formulario de revisión, que se utiliza para crear nuevos formularios.

El archivo `views.py` contiene las vistas que manejan la lógica y la renderización de las páginas en la aplicación.

El archivo `urls.py` contiene la definición de las URL que se utilizan en la aplicación de la librería.

El proyecto `my_project` también incluye otros archivos necesarios para configurar y ejecutar el proyecto, como `settings.py`, `urls.py`, `wsgi.py`, etc.