DESARROLLO DE SOFTWARE

ESTUDIANTE

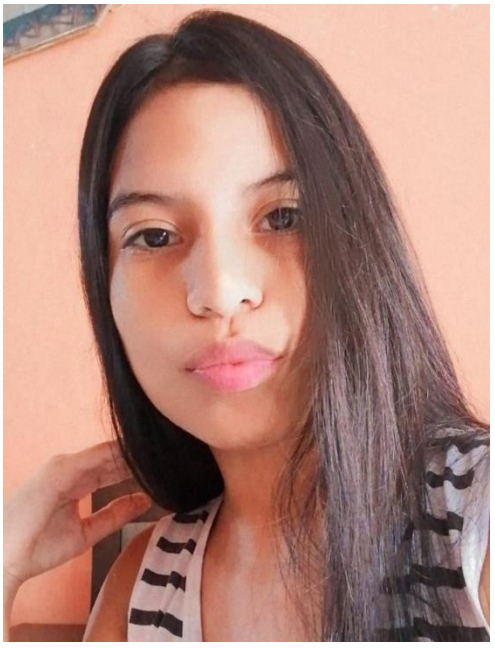
LEON ALVIA JOSELYN PAMELA

CURSO

3 ‘E’

DOCENTE

CARLOS LUIS PAZMIÑO PALMA



TEMAS:

1) ¿Qué es Django?

2) ¿Qué es la máquina virtual en Django?

3) ¿Qué es MVT en Django?

4) Crear un proyecto con la máquina virtual.

5) Descargar los instaladores de Django al proyecto

6) Crear un proyecto para programar en Django

7) Ejecutar el proyecto y el mensaje de felicitaciones.

8) Crear una Apps Core.

9) ¿Qué es la Carpeta Templates?

10) ¿Qué es la Carpeta stactic?

11)Crear un archivo base HTML en la APPS Core.

12)Como se llaman a los CSS desde el archivo base HTML.

13) Como consume un archivo hijo html al utilizar la herencia del archivo base html.

14) Crear un view que llame al html hijo

15)Crear la urls que llame al views.

16) Integrar la aplicación APPS Core al proyecto principal.

17) Crear las tablas del sistema de usuarios para utilizar el panel de administración.

18) Crear un usuario para poder ingresar al Panel de Administración

19)Que es un modelo en Django

20) Crear un modelo en Django.

21) Migrar el Modelo a la base del Panel de Administración.

22)Integrar el Modelo al Panel de Administración.

23)Ingresar información al modelo por el Panel de Administración.

24)Realizar la consulta de todo lo ingresado en el modelo desde el views.

25)Mostrar los datos guardados en el modelo al html hijo.

¿Qué es Django?

Django es un framework web de alto nivel que permite el desarrollo rápido de sitios web seguros y mantenibles. Desarrollado por programadores experimentados, Django se encarga de gran parte de las complicaciones del desarrollo web, por lo que puedes concentrarte en escribir tu aplicación sin necesidad de reinventar la rueda. Es gratuito y de código abierto, tiene una comunidad próspera y activa, una gran documentación y muchas opciones de soporte gratuito y de pago.

Django te ayuda a escribir software que es:

Completo:

Django sigue la filosofía "Baterías incluidas" y provee casi todo lo que los desarrolladores quisieran que tenga "de fábrica". Porque todo lo que necesitas es parte de un único "producto", todo funciona a la perfección, sigue principios de diseño consistentes y tiene una amplia y actualizada documentación.

Versátil:

Django puede ser (y ha sido) usado para construir casi cualquier tipo de sitio web — desde sistemas manejadores de contenidos y wikis, hasta redes sociales y sitios de noticias. Puede funcionar con cualquier framework en el lado del cliente, y puede devolver contenido en casi cualquier formato (incluyendo HTML, RSS feeds, JSON, XML, etc). ¡El sitio que estás leyendo actualmente está basado en Django!

Internamente, mientras ofrece opciones para casi cualquier funcionalidad que desees (distintos motores de base de datos , motores de plantillas, etc.), también puede ser extendido para usar otros componentes si es necesario.

Seguro:

Django ayuda a los desarrolladores evitar varios errores comunes de seguridad al proveer un framework que ha sido diseñado para "hacer lo correcto" para proteger el sitio web automáticamente. Por ejemplo, Django, proporciona una manera segura de administrar cuentas de usuario y contraseñas, evitando así errores comunes como colocar informaciones de sesión en cookies donde es vulnerable (en lugar de eso las cookies solo contienen una clave y los datos se almacenan en la base de datos) o se almacenan directamente las contraseñas en un hash de contraseñas.

Un hash de contraseña es un valor de longitud fija creado al enviar la contraseña a una cryptographic hash function. Django puede validar si la contraseña ingresada es correcta enviándola a través de una función hash y comparando la salida con el valor hash almacenado. Sin embargo, debido a la naturaleza "unidireccional" de la función, incluso si un valor hash almacenado se ve comprometido es difícil para un atacante resolver la contraseña original.

¿Qué es la máquina virtual en Django?

Visión general del entorno de desarrollo Django:

Django hace muy fácil configurar tu computadora de manera que puedas empezar a desarrollar aplicaciones web. Esta sección explica qué consigues con el entorno de desarrollo y proporciona una visión general de algunas de tus opciones de puesta en marcha y configuración. El resto del artículo explica el método *recomendado* de instalación del entorno de desarrollo de Django en Ubuntu, Mac OS X, y Windows, y cómo puedes probarlo.

¿Qué es el entorno de desarrollo Django?

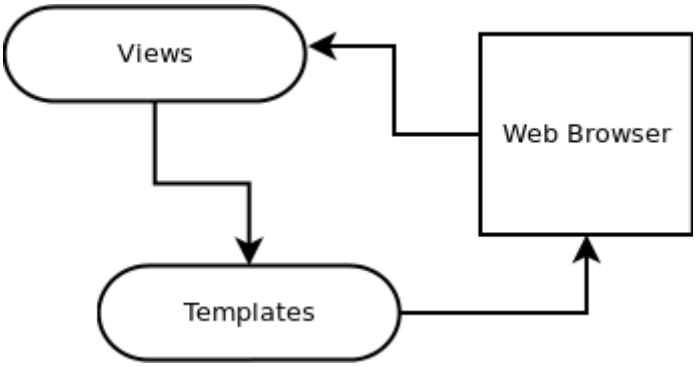
El entorno de desarrollo es una instalación de Django en tu computadora local que puedes usar para desarrollar y probar apps Django antes de desplegarlas al entorno de producción. Las principales herramientas que el mismo Django proporciona son un conjunto de scripts de Python para crear y trabajar con proyectos Django, junto con un simple servidor web de desarrollo que puedes usar para probar de forma local (es decir en tu computadora, no en un servidor web externo) aplicaciones web Django con el explorador web de tu computadora.
Hay otras herramientas periféricas, que forman parte del entorno de desarrollo, que no cubriremos aquí. Estas incluyen cosas como un editor de textos o IDE para editar código, una herramienta de gestión del control de fuentes como Git para gestionar con seguridad las diferentes versiones de tu código. Asumimos que tienes ya un editor de textos instalado.
¿Cuáles son las opciones de puesta en marcha de Django?
Django es extremadamente flexible en términos de cómo y dónde puede instalarse y configurarse. Django puede ser:
instalado en diferentes sistemas operativos.
ser usado con Python 3 y Python 2.
instalado desde las fuentes, desde el Python Package Index (PyPi) y en muchos casos desde la aplicación de gestión de paquetes de la computadora.
configurado para usar una de entre varias bases de datos, que también pueden necesitar ser instaladas y configuradas por separado.
ejecutarse en el entorno Python del sistema principal o dentro de entornos virtuales Python separados.

¿Qué es MVT en Django?

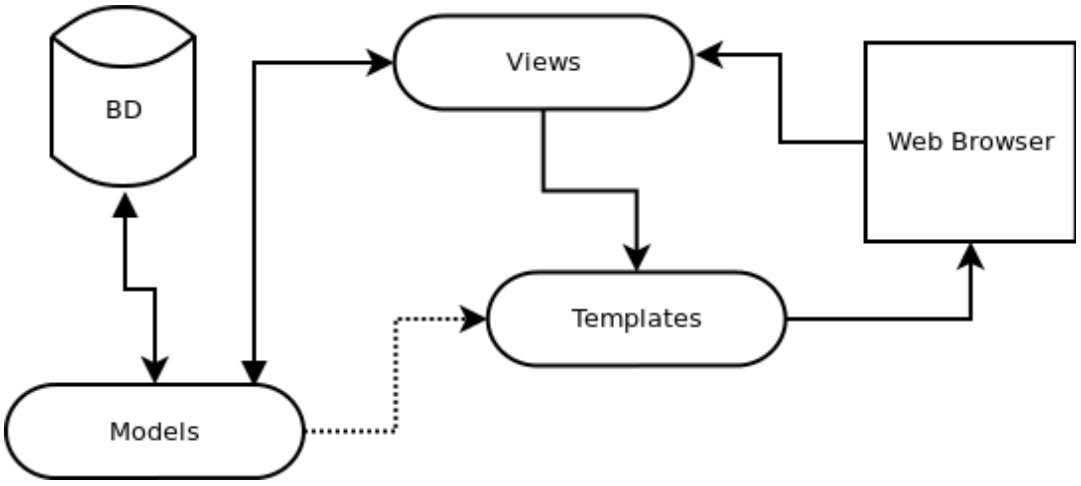
Patrón MVT: Modelo-Vista-Template:

Si tenéis experiencia en el mundo de la programación seguro que habéis oído hablar del famoso patrón MVC: Modelo-Vista-Controlador. Django redefine este modelo como MVT: Modelo-Vista-Template.

Hasta ahora lo que hemos hecho no requería de interactuar con la base de datos. Podríamos decir que simplemente se recibe una petición del navegador, se ejecuta la vista correspondiente y se renderiza el Template para que el navegador muestre el HTML resultante:



Sin embargo, en el momento en que aparecen la base de datos y los modelos, este proceso se extendiendo. Ahora se recibirá la petición, se pasará a la vista, en la vista recuperaremos los datos del modelo correspondiente, y finalmente la renderizaremos el Template, pero esta vez integrando los datos dinámicos recuperados del modelo, antes de enviar el HTML final al navegador:



Crear un proyecto con la máquina virtual.

Visión general:

Este artículo muestra cómo puedes crear un sitio web "esqueleto", que puedes luego llenar con configuraciones específicas del sitio, urls, modelos, vistas y plantillas (trataremos ésto en artículos posteriores).
El proceso es sencillo:
Usar la herramienta `django-admin` para crear la carpeta del proyecto, los ficheros de plantillas básicos y el script de gestión del proyecto (manage.py).
Usar manage.py para crear una o más *aplicaciones*.
Nota: Un sitio web puede consistir de una o más secciones, ej. sitio principal, blog, wiki, area de descargas, etc. Django te recomienda encarecidamente que desarrolles estos componentes como *aplicaciones* separadas que podrían ser reutilizadas, si se desea, en otros proyectos.
Registrar las nuevas aplicaciones para incluirlas en el proyecto.
Conectar el mapeador url de cada aplicación.
Para el sitio web de la Biblioteca Local la carpeta del sitio y la carpeta de su proyecto se llamarán *locallibrary*, y tendremos sólo una aplicación llamada *catalog*. El nivel más alto de la estructura de carpetas quedará por tanto como sigue:

```
locallibrary/          # Carpeta del sitio web
    manage.py          # Script para ejecutar las herramientas de
Django para este proyecto (creadas usando django-admin)
    locallibrary/      # Carpeta del Sitio web/Proyecto (creada usando
django-admin)
    catalog/           # Carpeta de la Aplicación (creada usando
manage.py)
```

Las siguientes secciones abordan los pasos del proceso en detalle, y muestran cómo puedes comprobar los cambios. Al final de cada artículo trataremos alguna de los otros ajustes aplicables al sitio entero que deberías también efectuar en esta etapa.

Creación del proyecto:

En primer lugar, abre una ventana de comandos/terminal, navega hasta donde quieres almacenar tus apps Django (hazlo en algún lugar que sea fácil de encontrar, como dentro de la carpeta de tus *documentos*), y crea una carpeta para tu nuevo sitio web (en este caso: *locallibrary*). Entra en el directorio a continuación usando el comando cd:

```
mkdir locallibrary
cd locallibrary
```

Crear el nuevo proyecto usando el comando `django-admin startproject` como se muestra, y navega luego dentro de la carpeta.

```
django-admin startproject locallibrary .
cd locallibrary
```

La herramienta `django-admin` crea una estructura de carpetas/ficheros como se muestra abajo:

```
locallibrary/
    manage.py
    locallibrary/
        settings.py
        urls.py
        wsgi.py
```

La subcarpeta del proyecto *locallibrary* es el punto de entrada al sitio web:
settings.py contiene todos los ajustes del sitio. Es donde registramos todas las aplicaciones que creamos, la localización de nuestros ficheros estáticos, los detalles de configuración de la base de datos, etc.
urls.py define los mapeos url-vistas. A pesar de que éste podría contener *todo* el código del mapeo url, es más común delegar algo del mapeo a las propias aplicaciones, como verás más tarde.
wsgi.py se usa para ayudar a la aplicación Django a comunicarse con el servidor web. Puedes tratarlo como código base que puedes utilizar de plantilla.
El script manage.py se usa para crear aplicaciones, trabajar con bases de datos y empezar el desarrollo del servidor web.

Creación de la aplicación catalog:

A continuación, ejecuta el siguiente comando para crear la aplicación *catalog* que vivirá dentro de nuestro proyecto locallibrary (éste debe ejecutarse en la misma carpeta que el manage.py de tu proyecto):

```
python3 manage.py startapp catalog
```

Nota: el comando de arriba es para Linux/Mac OS X. En Windows el comando debería ser: `py -3 manage.py startapp catalog`

Si estás trabajando en Windows, reemplaza `python3` por `py -3` a lo largo de este módulo o simplemente python:
`python manage.py startapp catalog`.
La herramienta crea una nueva carpeta y la rellena con ficheros para las diferentes partes de la aplicación (mostradas en negrilla abajo).
La mayoría de los ficheros se nombran de acuerdo a su propósito, para que sea más útil (ej. las vistas se deberán guardar en views.py, los Modelos en models.py, las pruebas en tests.py, la configuración del sitio de administración en admin.py, el registro de aplicaciones en apps.py) y contienen algo de código base mínimo para trabajar con los objetos asociados.

El directorio actualizado del proyecto debería tener ahora el aspecto siguiente:

```
locallibrary/
    manage.py
    locallibrary/
    catalog/
        admin.py
        apps.py
        models.py
        tests.py
        views.py
        __init_.py
        migrations/
```

Además ahora tenemos:

Una carpeta *migrations* que se utiliza para guardar las "migraciones"— ficheros que te permiten actualizar tus bases de datos a medida que modificas tus modelos.
__init_.py — Un fichero vacío creado aquí para que Django/Python reconozca la carpeta como un Paquete Python y te permita usar sus objetos dentro de otras partes del proyecto.
Nota: ¿Te has dado cuenta qué es lo que falta en la lista de ficheros de arriba? Si bien hay un lugar para que coloques tus vistas y modelos, no hay nada para que pongas los mapeos url, las plantillas y los ficheros estáticos. Te mostraremos cómo crearlos más adelante (éstos no se necesitan en todos los sitios web, pero se necesitan en este ejemplo).

Descargar los instaladores de Django al proyecto

Una vez que se descomprime el archivo descargado, debemos acceder a la carpeta desde una terminal o ventana de comandos (en caso de los usuarios de Windows). Suponiendo que la versión que elegimos es la 1.4, se tendría que digitar:

```
cd Django-1.4
```

Ya en la carpeta de instalación de Django, se debe digitar la siguiente instrucción: (Debes de tener permisos de administrador)

```
python setup.py install
```

Si usas Ubuntu GNU/Linux, sería algo así:

```
sudo python setup.py install
```

Listo eso es todo, ya tienes Django instalado. Si se desea mayor información de como instalarlo o quizás algunas otras opciones, siempre está la documentación del mismo proyecto para guiarnos. Aquí pueden encontrar la Guía completa de instalación de Django.

Si usas OS X Lion, quizás lo más recomendado sería instalar Django usando pip:
Primero, debemos tener Setup Tools para instalarlo, lo descargamos, lo descomprimimos y dentro del directorio (via el terminal)

```
sudo python setup.py install
```

Una vez con Setup Tools instalado usamos pip:

```
sudo pip install Django
```

Crear un proyecto para programar en Django

Antes de empezar es bueno aclarar que la versión que vamos a utilizar en esta guía es la más reciente (1.4), y varias cosas han cambiado, la información que podrían encontrar en Internet probablemente se encuentre desactualizada.

Para crear nuestro primer proyecto, abrimos una terminal (o ventana de comandos si así lo conoces en Windows), nos ubicamos en la carpeta en donde queremos crear nuestro proyecto y digitamos:

```
django-admin.py startproject recetario
```

Esta instrucción creará dos directorios con el nombre del proyecto (en este caso: recetario) y 5 archivos distribuidos de la siguiente manera:

manage.py

recetario

__init_.py

settings.py

urls.py

wsgi.py

Para ver que el proyecto está funcionando en la terminal debemos escribir:

```
python manage.py runserver
```

Al ejecutar esa instrucción debemos visualizar un resultado como el siguiente:

python manage.py runserver

Abrimos el navegador web la dirección **http:// 127.0.0.1:8000/** y debemos ver lo siguiente:

Crear una Apps core.

Django apuesta por un sistema de reutilización de código organizado en *apps*, algo así como aplicaciones internas que implementan funcionalidades específicas.

Las Apps activas en un proyecto de Django, las encontramos definidas en el fichero de configuración **settings.py**, en la lista *INSTALLED_APPS*:

```
webpersonal/settings.py

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

No sé si lo recordaréis, pero al migrar algunas de estas apps no aparecían: Messages y StaticFiles. Eso es porque no necesariamente todas las apps requieren utilizar la base de datos, aunque por contra sí requieran estar activadas en esta lista.

La genialidad de Django recae en que que a parte de incluir muchas apps genéricas también nos permite crear las nuestras propias, y eso estimados alumnos es la mejor idea de este framework, pues una app no tiene que limitarse a un solo proyecto, sino que se puede reutilizar en varios. Sin ir más lejos, en los repositorios de PyPy existen miles de apps de Django creadas por la comunidad y que en pocos minutos podríamos estar utilizando sin mucha complicación.

En este curso vamos a crear un montón de apps, desde un portafolio hasta un blog, pasando por gestores de contenidos con páginas dinámicas y otras apps para manejar el registro de usuarios y sus perfiles. Todas y cada una de ellas son reutilizables y os servirán para un montón de proyectos.

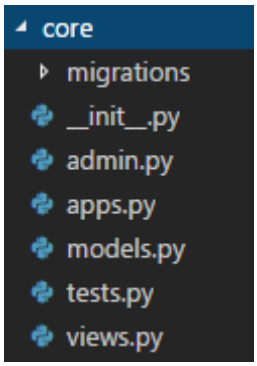
Por lo tanto podríamos concluir en que, mientras una app es una aplicación web que implementa una funcionalidad y por sí misma no sirve para nada, un proyecto es un conjunto de configuraciones a las que se "conectan" esas apps para que todo unido de lugar a un sitio web completo. Un proyecto puede contener múltiples apps, y una app puede ser incluida en múltiples proyectos.

Ahora que conocemos la diferencia entre proyecto y app, vamos a crear nuestra primera app. Será el núcleo de nuestra web personal (el **core** en inglés) y nos servirá como base para aprender como fluyen los datos en Django.

Así que vamos a la terminal, presionamos *Control + C* para cancelar la ejecución del servidor y escribimos:

```
python manage.py startapp core
```

Al hacerlo podréis observar como se ha creado un nuevo directorio core en nuestro proyecto:



Vamos a ir descubriendo los ficheros que conforman la app core sobre la marcha, no tiene mucho sentido explicar cosas que no utilizaremos hasta dentro de varias horas.

De todos estos ficheros el que nos interesa es ese llamado **views.py**. Este fichero es uno de los más importantes y en él se definen las vistas de la app. Una vista hace referencia a la lógica que se ejecuta cuando se hace una petición a nuestra web, y lo que vamos a hacer es crear una vista para procesar la petición a la raíz de nuestro sitio, lo que sería la portada.

Vamos a ir arriba del todo y vamos a importar un método del módulo **django.http** llamado **HttpResponse**:

core/views.py

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

Este método que nos permite contestar a una petición devolviendo un código, así que vamos a definir una vista para la portada y devolveremos algo de HTML de ejemplo:

core/views.py

```
def home(request):
    return HttpResponse("<h1>Mi Web Personal</h1><h2>Portada</h2>")
```

Cada vista se corresponde con una función del fichero **views.py**. Podéis usar el nombre que queráis pero como esta es la portada yo le llamo home. Además notad que se recibe un argumento llamado request, se trata de la petición y contiene mucha información, más adelante haremos uso de ella.

Ahora ya tenemos la vista con la portada, pero todavía no le hemos dicho a Django en qué URL tiene que mostrarla.

¿Recordáis el fichero **webpersonal/urls.py** dentro del directorio de configuración del proyecto? Pues es momento de volver ahí. Siguiendo la documentación superior, vamos a hacer lo que nos indican las instrucciones:

webpersonal/urls.py

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

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

¿Qué significa esto? Pues que del package **core** (notad ese **__init.py_**) importamos el módulo **views**, es decir, de la app **core** importamos las vistas. Y a continuación creamos un **patrón url**, justo en la raíz del sitio (cadena vacía) desde el que llamaremos a la vista **views.home** a la que damos el nombre **home**.

¿Qué es la Carpeta Templates?

En la carpeta Templates, las subcarpetas contienen las plantillas que se muestran en el cuadro de diálogo Crear nuevo archivo. Puede crear y guardar plantillas personalizadas en la carpeta Templates. Contiene plantillas para la creación de archivos. Se puede guardar el archivo activo como una plantilla.

¿Qué es la Carpeta stactic?

Cuando se desarrolla una aplicación web se manejan recursos estáticos como logos, videos, archivos javascript y css los cuales se agregan directamente a las plantillas html generadas, estos, normalmente son almacenados en un directorio llamado “public”.

Crear un archivo base html en la APPS core

Ahora que hemos definindo nuestros modelos y hemos creado los primeros registros en la librería para trabajar, es hora de escribir código para presentar información a los usuarios. Lo primero que necesitamos es determinar que información queremos mostrar en nuestras páginas, y definir una URL apropiada hacia estos recursos. Vamos a necesitar crear el mapeador de URLs, las vistas y plantillas para mostrar estas páginas.

El siguiente diagrama es un recordatorio del principal flujo de datos y cosas necesarias para ser implementadas cuando se maneja una respuesta/petición en HTTP.
Los principales elementos que necesitamos crear son:

Mapeadores URL para reenviar las URLs admitidas (y cualquier información codificada en las URLs) a las funciones de vista apropiadas.

Funciones de vista para obtener los datos solicitados desde los modelos, crear una página HTML que muestre los datos, y devolverlo al usuario para que lo vea en el navegador.

Plantillas usadas por las vistas para renderizar los datos.

Como verás en la siguiente sección, vamos a tener 5 páginas para mostrar, que es mucho que documentar en un artículo. Por lo tanto, en la mayor parte de este artículo nos concentraremos en mostrar como implementar solo la página de inicio (nos moverermos a otras páginas en un artículo subsecuente). Esto debe darte un buen entendimiento de extremo a extremo sobre como los mapeadores URL, vistas y modelos funcionan en la práctica.

Definiendo el recurso URL

Como esta versión de *LocalLibrary* es escencialmente solo de lectura para usuarios finales, debemos proveer una página de llegada para el sitio (una página de inicio), y páginas que *desplieguen* listas y vistas detalladas para libros y autores.

Las URL que vamos a necesitar para nuestras páginas son:

`catalog/` — La página home/index.

`catalog/books/` — La lista de todos los libros.

`catalog/authors/` — La lista de todos los autores.

`catalog/book/<id>` — La vista detallada para el libro específico con un campo de clave primaria de `<id>` (el valor por defecto). Así por ejemplo, `/catalog/book/3`, para el tercer libro añadido.

`catalog/author/<id>` — La vista detallada para el autor específico con un campo de clave primaria llamada `<id>`. Así por ejemplo, `/catalog/author/11`, para el 11vo autor añadido.

La tres primeras URLs son usadas para listar el índice, los libros y autores. Esto no codifica ninguna información adicional, y mientras los resultados retornados dependerán del contenido en la base de datos, las consultas que se ejecutan para obtener la información siempre serán las mismas.

En contraste las 2 URLs finales son usadas para mostrar información detallada sobre un libro o autor específico — estas codifican la identidad de los ítemes a mostrar en la URL (mostrado arriba como `<id>`). El mapeador URL puede extraer la información codificada y pasársela a la vista, donde se detarminará que información extraer de la base de datos. Al codificar la información en nuestra URL solo necesitamos un mapeador de URL, una vista, y un plantilla para manejar cada libro (o autor).

Como se llaman a los CSS desde el archivo base html.

Para este tutorial, te sugiero que utilices las herramientas más sencillas. Por ejemplo, Notepad (Windows), TextEdit (Mac) o HTML Kit, serán suficiente. Una vez comprendido lo básico, probablemente se quieran utilizar herramientas más complicadas, o incluso programas comerciales como Style Master, Dreamweaver o GoLive. Pero para el desarrollo de una primera hoja de estilos, es mejor no distraerse con características avanzadas de otras herramientas.
No utilices procesadores de texto como Microsoft Word u OpenOffice. Con ellos, normalmente se generan archivos que los navegadores no pueden interpretar. Para HTML y CSS, lo único que necesitamos son archivos en texto plano.
El paso 1 consiste en abrir tu editor de texto (Notepad, TextEdit, HTML Kit o el que desees), comenzar con una ventana vacía y escribir lo siguiente:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN">

<html>

<head>

  <title>Mi primera página con estilo</title>

</head>

<body>

<!-- Menú de navegación del sitio -->

<ul class="navbar">

  <li><a href="indice.html">Página principal</a>

  <li><a href="meditaciones.html">Meditaciones</a>

  <li><a href="ciudad.html">Mi ciudad</a>

  <li><a href="enlaces.html">Enlaces</a>

</ul>

<!-- Contenido principal -->

<h1>Mi primera página con estilo</h1>

<p>¡Bienvenido a mi primera página con estilo!

<p>No tiene imágenes, pero tiene estilo.

<p>Debería haber más cosas aquí, pero todavía no sé

qué poner.

<!-- Firma y fecha de la página, ¡sólo por cortesía! -->

<address>Creada el 5 de abril de 2004<br>

  por mí mismo.</address>

</body>

</html>
```

Como consume un archivo hijo html al utilizar la herencia del archivo base html.

Herencia de plantillas:

Nuestras plantillas de ejemplo hasta el momento han sido fragmentos de HTML, pero en el mundo real, usarás el sistema de plantillas de Django para crear páginas HTML enteras. Esto conduce a un problema común del desarrollo web: ¿Cómo reducimos la duplicación y redundancia de las áreas comunes de las páginas, como por ejemplo, los paneles de navegación?

Una forma clásica de solucionar este problema es usar *includes*, insertando dentro de las páginas HTML a "incluir" una página dentro de otra. Es más, Django admite esta aproximación, con la etiqueta `{% include %}` anteriormente descrita. Pero la mejor forma de solucionar este problema con Django es usar una estrategia más elegante llamada *herencia de plantillas*.

En esencia, la herencia de plantillas te deja construir una plantilla base "esqueleto" que contenga todas las partes comunes de tu sitio y definir "bloques" que los hijos puedan sobreescribir.

Veamos un ejemplo de esto creando una plantilla completa para nuestra vista `current_datetime`, editando el archivo `current_datetime.html`:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
    <title>The current time</title>
</head>
<body>
    <h1>My helpful timestamp site</h1>
    <p>It is now {{ current_date }}.</p>
    <hr>
    <p>Thanks for visiting my site.</p>
</body>
</html>
```

Esto se ve bien, pero ¿Qué sucede cuando queremos crear una plantilla para otra vista — digamos, ¿La vista `hours_ahead` del Capítulo 3? Si queremos hacer nuevamente una agradable, válida, y completa plantilla HTML, crearíamos algo como:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
```

```
    <title>Future time</title>
</head>
<body>
    <h1>My helpful timestamp site</h1>
    <p>In {{ hour_offset }} hour(s), it will be {{ next_time }}.</p>

    <hr>
    <p>Thanks for visiting my site.</p>
</body>
</html>
```

Claramente, estaríamos duplicando una cantidad de código HTML. Imagina si tendríamos más sitios típicos, incluyendo barra de navegación, algunas hojas de estilo, quizás algo de JavaScript — terminaríamos poniendo todo tipo de HTML redundante en cada plantilla.

La solución a este problema usando includes en el servidor es sacar factor común de ambas plantillas y guardarlas en recortes de plantillas separados, que luego son incluidos en cada plantilla. Quizás quieras guardar la parte superior de la plantilla en un archivo llamado `header.html`:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
```

Y quizás quieras guardar la parte inferior en un archivo llamado `footer.html`:

```
<hr>
    <p>Thanks for visiting my site.</p>
</body>
</html>
```

Crear un view que llame al html hijo

Con una estrategia basada en includes, la cabecera y la parte de abajo son fáciles. Es el medio el que queda desordenado. En este ejemplo, ambas páginas contienen un título — `<h1>My helpful timestamp site</h1>` — pero ese título no puede encajar dentro de `header.html` porque `<title>` en las dos páginas es diferente. Si incluimos `<h1>` en la cabecera, tendríamos que incluir `<title>`, lo cual no permitiría personalizar este en cada página. ¿Ves a dónde queremos llegar?

El sistema de herencia de Django soluciona estos problemas. Lo puedes pensar a esto como la versión contraria a la del lado del servidor. En vez de definir los pedazos que son *comunes*, defines los pedazos que son *diferentes*.

El primer paso es definir una *plantilla base* — un "esqueleto" de tu página que las *plantillas hijas* llenaran luego. Aquí hay una platilla para nuestro ejemplo actual:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
    <title>{% block title %}{% endblock %}</title>
</head>
<body>
    <h1>My helpful timestamp site</h1>
    {% block content %}{% endblock %}
    {% block footer %}
    <hr>
    <p>Thanks for visiting my site.</p>
    {% endblock %}
</body>
</html>
```

Esta plantilla, que llamamos `base.html`, define un documento esqueleto HTML simple que usaremos para todas las páginas del sitio. Es trabajo de las plantillas hijas sobreescribir, agregar, dejar vacío el contenido de los bloques. (Si estás lo siguiendo desde casa, guarda este archivo en tu directorio de plantillas).

Usamos una etiqueta de plantilla aquí que no hemos visto antes: la etiqueta `{% block %}`. Todas las etiquetas `{% block %}` le indican al motor de plantillas que una plantilla hijo quizás sobreescriba esa porción de la plantilla.

Ahora que tenemos una plantilla base, podemos modificar nuestra plantilla existente `current_datetime.html` para usar esto:

```
{% extends "base.html" %}

{% block title %}The current time{% endblock %}

{% block content %}
<p>It is now {{ current_date }}.</p>
{% endblock %}
```

Como estamos en este tema, vamos a crear una plantilla para la vista `hours_ahead` del Capítulo 3. (Si lo estás siguiendo junto con el código, te dejamos cambiar `hours_ahead` para usar el sistema de plantilla). Así sería el resultado:

```
{% extends "base.html" %}

{% block title %}Future time{% endblock %}

{% block content %}
<p>In {{ hour_offset }} hour(s), it will be {{ next_time }}.</p>
{% endblock %}
```

Crear la urls que llame al views.

Para ver como gestiona Django las **vistas** , las relaciona con el **modelo de datos** y envía esa información al **navegador** vamos a empezar creando una vista sencilla, simplemente vamos a mostrar un mensaje por pantalla.

Editamos el fichero **./album/views.py**

```
from django.http import HttpResponse



def first_view(request):
    return HttpResponse('Esta es mi primera vista!')
```

Una **vista** es una función o un método que básicamente hace dos cosas, toma como **argumento un objeto HttpRequest** , en el que va la información referente a la solicitud que estamos haciendo, como por ejemplo si el método empleado es POST o GET o el directorio donde está la página solicitada; y **devuelve un objeto HttpResponse** con la información de la página que va a mostrar o una excepción si algo ha ido mal.

Ahora tenemos que asociar la vista que acabamos de definir con una **dirección** , para hacerlo vamos a manejar dos ficheros, uno lo crea Django al iniciar el proyecto y está en **./myapps/urls.py** , el otro lo tendremos que crear nosotros dentro del directorio de nuestra aplicación, **./album/urls.py** .

Vamos a editar el primero **./myapps/urls.py** .

```
from django.conf.urls import patterns, include, url
from django.contrib import admin



urlpatterns = patterns('',
    url(r'^admin/', include(admin.site.urls)),
)
```

Si nos olvidamos de las líneas que tiene comentadas vemos que tiene una **relación** entre la dirección **/admin** y el módulo **admin.site.urls** . Esta es la forma que tiene Django de **enlazar** la **dirección** con el fichero **urls.py** propio de la aplicación admin que ya hemos usado en los capítulos anteriores. De esta forma al dirigirnos a http://127.0.0.1:8000/admin/ lo que hacemos es empezar a usar dicho fichero para los siguientes enlaces que usemos, como por ejemplo http://127.0.0.1:8000/admin/album/photo/.

Integrar la aplicación APPS core al proyecto principal

Cada vista se corresponde con una función del fichero **views.py**. Podéis usar el nombre que queráis pero como esta es la portada yo le llamo home. Además notad que se recibe un argumento llamado request, se trata de la petición y contiene mucha información, más adelante haremos uso de ella.

Ahora ya tenemos la vista con la portada, pero todavía no le hemos dicho a Django en qué URL tiene que mostrarla.

¿Recordáis el fichero **webpersonal/urls.py** dentro del directorio de configuración del proyecto? Pues es momento de volver ahí. Siguiendo la documentación superior, vamos a hacer lo que nos indican las instrucciones:

webpersonal/urls.py

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


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

¿Qué significa esto? Pues que del package **core** (notad ese **__init.py_**) importamos el módulo **views**, es decir, de la app **core** importamos las vistas. Y a continuación creamos un **patrón url**, justo en la raíz del sitio (cadena vacía) desde el que llamaremos a la vista **views.home** a la que damos el nombre **home**.

Crear las tablas del sistema de usuarios para utilizar el panel de administración.

La *LocalLibrary* actualmente lista todos los autores usando el nombre generado por el método `__str__()` del modelo. Esto funciona bien cuando solo tienes unos pocos autores, pero una vez que tienes muchos puedes terminar teniendo duplicados. Para diferenciarlos, o simplemente para mostrar información más interesante sobre cada autor, puede usar list_display para agregar otros campos a la vista.

Reemplaza tu clase `AuthorAdmin` con el código de abajo. Los nombres de campos a ser desplegados en la lista están declarados en una tupla en el orden requerido, como se muestra (estos son los mismos nombres especificados en tu modelo original).

```
class AuthorAdmin(admin.ModelAdmin):

    list_display = ('last_name', 'first_name', 'date_of_birth',
'date_of_death')
```

Copiar al portapapeles

Recarga el sitio y navega hacia la lista de autores. Ahora deben desplegar los campos de arriba, así:

Para nuestro modelo `Book`desplegaremos adicionalmente el `author`y `genre`. El `author`es un campo de relación tipo `ForeignKey`(uno a uno), y por tanto estará representado por el valor `__str__()` del registro asociado. Reemplace la clase BookAdmin con la versión de abajo.

```
class BookAdmin(admin.ModelAdmin):

    list_display = ('title', 'author', 'display_genre')
```

Desafortunadamente, no podemos especificar directamente el campo `genre`en `list_display`porque es un campo `ManyToManyField`(Django previene esto porque habría un alto "costo" de acceso a base de datos si lo hizo). En lugar de eso, definiremos una función `display_genre`para obtener la información como una cadena (esta es la función que hemos llamado arriba; la definiremos más abajo).

Crear un usuario para poder ingresar al Panel de Administración

Para iniciar sesión en el sitio de administración, necesitamos una cuenta de usuario con estado de *Personal* habilitado. Para ver y crear registros tambien necesitamos que este usuario tenga permisos para administrar todos nuestros objetos. Puedes crear una cuenta "administrador" que tenga acceso total al sitio ya todos los permisos necesarios usando **manage.py** .

Usa el siguiente comando, en el mismo directorio de **manage.py** , para crear al administrador. Deberá ingresar un nombre de usuario, dirección de correo electrónico y una contraseña *fuerte* .

```
python3 manage.py createsuperuser
```

Copiar al portapapeles

Una vez que el comando termine un nuevo administrador será agregado a la base de datos. Ahora reinicia el servidor de desarrollo para que podamos probar el inicio de sesión:

```
python3 manage.py runserver
```

Que es un modelo en Django

Un Django model **generalmente se refiere a una tabla de la base de datos**, los atributos de ese modelo se convierten en las columnas de esa tabla esos atributos reciben el nombre de django fields los cuales manejan automáticamente las conversiones de tipos de datos para la base de datos que estemos usando.

Crear un modelo en Django.

Los modelos están definidos, normalmente, en el archivo **models.py** de la aplicación. Son implementados como subclases de `django.db.models.Model`, y pueden incluir campos, métodos y metadatos. El fragmento de código más abajo muestra un modelo "típico", llamado `MyModelName`:

```
from django.db import models

class MyModelName(models.Model):
    """
    Una clase típica definiendo un modelo, derivado desde
la clase Model.
    """
```

```
    # Campos
    my_field_name = models.CharField(max_length=20,
help_text="Enter field documentation")
    ...

    # Metadata
    class Meta:
        ordering = ["-my_field_name"]

    # Métodos
    def get_absolute_url(self):
         """
         Devuelve la url para acceder a una instancia
particular de MyModelName.
         """
         return reverse('model-detail-view',
args=[str(self.id)])

    def__str__(self):
        """
        Cadena para representar el objeto MyModelName (en
el sitio de Admin, etc.)
        """
        return self.field_name
```

En las secciones de abajo exploraremos cada una de las características interiores de un modelo en detalle:

Campos

Un modelo puede tener un número arbitrario de campos, de cualquier tipo. Cada uno representa una columna de datos que queremos guardar en nuestras tablas de la base de datos. Cada registro de la base de datos (fila) consistirá en uno de cada posible valor del campo. Echemos un vistazo al ejemplo visto arriba:

```
my_field_name = models.CharField(max_length=20, help_text="Enter field
documentation")
```

Migrar el Modelo a la base del Panel de Administración.

Por defecto, Django utiliza SQLite 3 para el almacenamiento de datos; pero permite cambiar a otros motores fácilmente.

Cada aplicación incluida en el núcleo de Django (sesiones, permisos, autenticación y otras cosas propias del framework) necesita una base de datos de igual manera, y la generación de las mismas se hace a través de migraciones.

Comencemos viendo el archivo settings.py que muestra las apps instaladas y las configuraciones de las bases de datos:

```
#----------

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
#----------
# Database
# https://docs.djangoproject.com/en/2.2/ref/settings/#databases

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

Por defecto se usa sqlite3 y el nombre es db.sqlite3

También fíjate en el apartado `INSTALLED_APPS`, las mismas vienen incluidas en el núcleo de Django para facilitarnos algunas cosas.

Si se quisiera modificar el motor, simplemente hay que modificar el apartado de `DATABASES` y agregar un usuario y contraseña en caso de ser necesario, pero eso lo veremos en otro post.

Más adelante vamos a modificar `INSTALLED_APPS` para agregar nuestra app de gastos y de esta manera hacer que se generen las migraciones automáticas.

Migíaciones en Django

Las migraciones en Django son una manera de modificar la estructura de las bases de datos. En ocasiones se crean tablas, en otras se remueven, pero siempre se hacen en una sola migración.

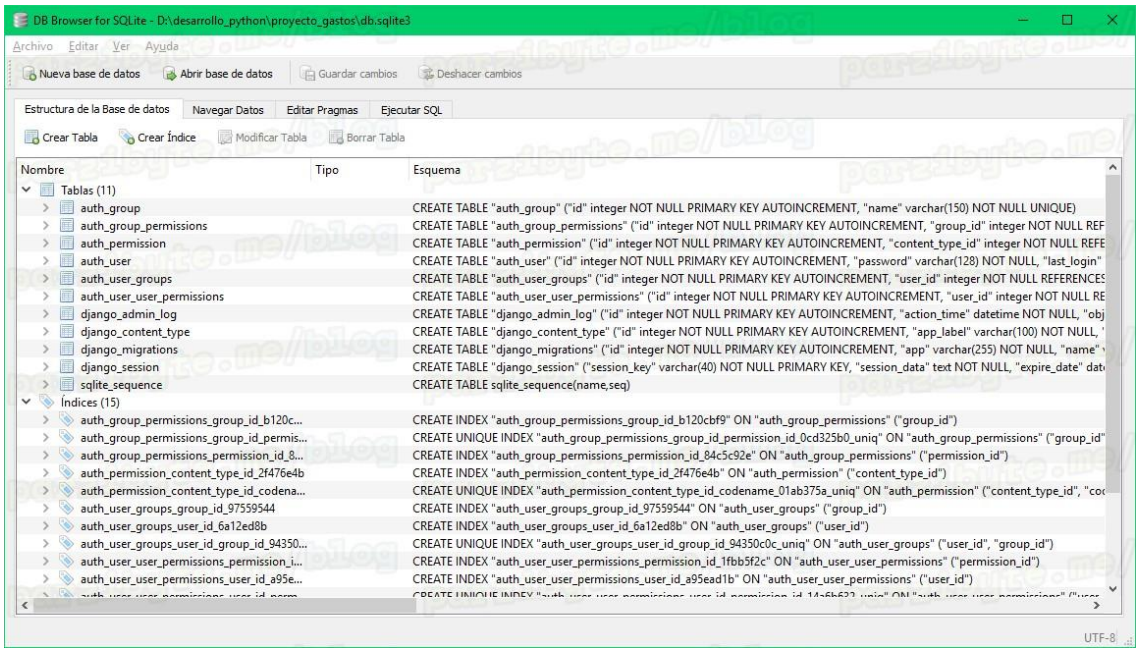
Por ejemplo, se podría tener una migración para crear la tabla de ingresos monetarios, y si después se agrega una tabla con una relación, se podría crear otra migración.

Vamos a comenzar haciendo la primer migración, que va a crear tablas que el framework utiliza.

Para ello, ejecuta:

```
python manage.py migrate
```

Si analizamos la base de datos, veremos que se han creado tablas que no nos pertenecen pero son necesarias para Django:



Tablas creadas por migración de Django

Todavía no hemos creado tablas para nuestros modelos, eso lo haremos en un momento.

Integrar el Modelo al Panel de Administración.

Para cambiar la forma en la que un modelo se despliega en la interfaz de administración debe definir una clase ModelAdmin (que describe el diseño) y registrarla con el modelo.

Comenzamos con el modelo Author. Abre admin.py en la aplicación catalog (/locallibrary/catalog/admin.py). Comenta tu registro original para el modelo `Author`(colocando el prefijo # en la línea):

```
# admin.site.register(Author)
```

Copiar al portapapeles

Ahora agrega una nueva clase `AuthorAdmin`y registre como se muestra abajo.

```
# Define the admin class
class AuthorAdmin(admin.ModelAdmin):
    pass

# Register the admin class with the associated model
admin.site.register(Author, AuthorAdmin)
```

Copiar al portapapeles

Ahora añadiremos clases `ModelAdmin` para `Book`, y `BookInstance`. De nuevo, debemos comentar nuestros registros originales:

```
#admin.site.register(Book)
#admin.site.register(BookInstance)
```

Copiar al portapapeles

Ahora, para crear y registrar los nuevos modelos usaremos, para propósitos de esta demostración, la expresión `@register` para registrar los modelos (hace exactamente lo mismo que `admin.site.register()`):

```
# Register the Admin classes for Book using the decorator

@admin.register(Book)
```

```
class BookAdmin(admin.ModelAdmin):
    pass


# Register the Admin classes for BookInstance using the decorator


@admin.register(BookInstance)
class BookInstanceAdmin(admin.ModelAdmin):
    pass
```

Ingresar información al modelo por el Panel de Administración.

En el archivo *admin.py* dentro de nuestra aplicación de Django vamos a heredar de la clase admin.ModelAdmin.

Podemos dotar a este modelo de la propiedad *list_display*, para decirle al admin que campos queremos listar en el administrador.

La propiedad *search_field* nos permite especificar sobre que campos va a efectuarse la búsqueda.

```
 # videogameStore/admin.py
from django.contrib import admin
from .models import Videogame


class VideogameAdmin(admin.ModelAdmin):
      list_display = ('name', 'created') #Ahora la interfaz mostrará
nombre, apellido y email de cada autor.
      search_fields = ('name', 'description')


admin.site.register(Videogame, VideogameAdmin)
```

Mira el campo created y la barra de búsqueda

Modificar el orden de los campos al editar

Si queremos modificar el orden de los campos agregamos una propiedad a nuestra clase

```
# videogameStore/admin.py
```

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

class VideogameAdmin(admin.ModelAdmin):
      list_display = ('name', 'created') #Ahora la interfaz mostrará
nombre, apellido y email de cada autor.
      search_fields = ('name', 'description')
      fields = ('description', 'name', 'genre', 'rating')
admin.site.register(Videogame, VideogameAdmin)
```

Como puedes apreciar, el orden en que aparecen los campos se ha modificado. En orden Description, Name, Genre y Rating, justo como lo especificamos.

Observa como el orden ha cambio al especificado en la propiedad fields

Ordenando los objetos por un campo

Ordering especifica el campo que se usará para ordenar los modelos.

```
# videogameStore/admin.py
from django.contrib import admin
from .models import Videogame

class VideogameAdmin(admin.ModelAdmin):
      list_display = ('name', 'created') #Ahora la interfaz mostrará
nombre, apellido y email de cada autor.
      search_fields = ('name', 'description')
      fields = ('description', 'name', 'genre', 'rating')
      ordering = ('-name',)
admin.site.register(Videogame, VideogameAdmin)
```

Aquí le hemos dicho que los ordene por su nombre, de manera descendente, usando el símbolo “-”.

Nota el acomodo de los modelos en orden descendente

Realizar la consulta de todo lo ingresado en el modelo desde el views.

Un Formulario HTML es un conjunto de uno o más campos/widgets en una página web, que pueden ser usados para recolectar información de los usuarios para el envío a un servidor. Los formularios son un mecanismo flexible para recolectar datos de entrada porque son widgets adecuados para ingresar diferentes tipos de datos, incluyendo campos de texto, checkboxes, radio buttons, selector de fechas, etc. Los formularios son también una forma relativamente segura de compartir datos con el servidor, ya que permiten enviar información en peticiones `POST` con protección de falsificación de solicitud entre sitios.

Si bien nosotros aún no hemos creado ningún formulario en este tutorial todavia, ya lo hemos encontrado en el sitio de administración de Django; por ejemplo, la captura de pantalla de abajo muestra un formulario para editar uno de nuestros modelos de Libro, compuesto de un número de listas de selección y editores de texto.

¡Trabajar con formularios puede ser complicado! Los desarrolladores deben de escribir código HTML para el formulario, validar y adecuadamente limpiar los datos ingresados en el servidor (y posiblemente también en el browser o navegador), volver a publicar el formulario con mensajes de error para informar a los usuarios de cualquier campo invalido, manejar los datos cuando hayan sido enviados exitosamente y finalmente, responder al usuario de alguna manera, para indicar el éxito de la operación. Django Forms elimina mucho del trabajo de todos estos pasos, al proporcionar un marco de trabajo que le permite definir formularios y sus campos a travéz de programación y luego, utilizar estos objetos para generar el código HTML del formulario y manejar gran parte de la validación y la interacción del usuario.

En este tutorial vamos a mostrarle algunas de las formas de crear y trabajar con formularios y en particular, cómo las vistas genéricas de edición de formularios pueden significativamente reducir la cantidad del trabajo necesario para crear formularios para manejar sus modelos. En el camino iremos extendiendo nuestra aplicación *LocalLibrary* por agregar un formulario para permitir a los bibliotecarios renovar libros de la biblioteca y crearemos páginas para crear, editar o eliminar libros y autores (reproduciendo una versión básica del formulario mostrado arriba para editar libros).

Mostrar los datos guardados en el modelo al html hijo.

Abre el archivo de vistas (**locallibrary/catalog/views.py**) y agrega el siguiente bloque de código al final:

```
from django.views.generic.edit import CreateView, UpdateView, DeleteView

from django.urls import reverse_lazy

from .models import Author


class AuthorCreate(CreateView):
    model = Author
    fields = '__all__'
    initial={'date_of_death':'05/01/2018',}


class AuthorUpdate(UpdateView):
    model = Author
    fields = ['first_name','last_name','date_of_birth','date_of_death']


class AuthorDelete(DeleteView):
    model = Author
    success_url = reverse_lazy('authors')
```

Copiar al portapapeles

Como puede ver, para crear las vistas de las que necesita derivar `CreateView`, `UpdateView`, y `DeleteView`(respectivamente) y luego definir el modelo asociado.

Para los casos de "crear" y "actualizar", también debe especificar los campos para mostrar en el formulario (utilizando la misma sintaxis que para`ModelForm`). En este caso, mostramos la sintaxis para mostrar "todos" los campos y cómo puede enumerarlos individualmente. También puede especificar valores iniciales para cada uno de los campos usando un diccionario de pares nombre_campo / valor (aquí establecemos arbitrariamente la fecha de finalización para multas de demostración; ¡es posible que desee eliminar eso!). Por defecto, estas vistas se redirigirán en caso de éxito a una página que muestre el elemento del modelo recién creado/editado, que en nuestro caso será la vista detallada del autor que creamos en un tutorial anterior. Puede especificar una ubicación alternativa de redireccionamiento declarando claramente el parámetro `success_url`(como hecho en la clase `AuthorDelete`).

La clase `AuthorDelete`no necesita mostrar ninguno de los campos, por lo que no es necesario especificarlos. Sin embargo, debe especificar el `success_url`, porque no hay un valor predeterminado obvio para que Django lo use. En este caso usamos la función

para redirigir a nuestra lista de autores después de que un `reverse_lazy()` autor ha sido eliminado —`reverse_lazy()` `reverse()`

Plantillas - Plantillas

Las vistas "create" y "update" utilizan la misma plantilla de forma predeterminada, que se nombrará después de su model: *model_name_form.html* (puedes cambiar el sufijo a algo diferente a **_form usando** el campo `template_name_suffix`en tu vista, ejemplo: `template_name_suffix = '_other_suffix'`)

Crea la siguiente plantilla **locallibrary/catalog/templates/catalog/author_form.html** y copia el siguiente texto:

```
{% extends "base_generic.html" %}


{% block content %}


<form action="" method="post">
    {% csrf_token %}
    <table>
    {{ form.as_table }}
    </table>
    <input type="submit" value="Submit" />


</form>
{% endblock %}
```

Copiar al portapapeles

Esto es similar a nuestros formularios anteriores y representa los campos usando una tabla. Tenga en cuenta también cómo declaramos nuevamente `{% csrf_token %}`para garantizar que nuestros formularios sean resistentes a los ataques CSRF.