Constanza Vásquez
constanza.vasquez@biomedica.udec.cl

Tutorial Django

¿Qué es?

Django es un web framework (open source) de alto nivel escrito en Python, permite el desarrollo de aplicaciones web de forma rápida y segura.

Django puede ser utilizado para desarrollar casi cualquier tipo de aplicación web, desde sistemas de gestión de contenidos hasta páginas de noticias o redes sociales. Puede trabajar de forma complementaria con otros frameworks y también entregar contenido en casi cualquier formato (incluyendo HTML, RSS feed, JSON, XML, entre otros).

Django ayuda a los desarrolladores a evitar errores comunes de seguridad, puesto que está diseñado para proteger la aplicación automáticamente. Por ejemplo, entrega métodos seguros para administrar usuarios y contraseñas evitando el problema de poner información de la sesión en las cookies donde es más vulnerable. Las aplicaciones desarrolladas con Django cuentan con protección contra inyecciones SQL, cross-site scripting, coss-site forgery y clickjacking.

Django es escalable puesto que su arquitectura independiente permite modificar algunas secciones sin afectar a las otras, permitiendo hacer mejoras cuando la demanda aumenta o se requieren añadir funcionalidades.

Estructura

Django trabaja utilizando el **modelo MVT –Model, View, Template–**. El **modelo** (*model*) hace referencia a la base de datos, la **vista** (*view*) es la parte del sistema que selecciona qué mostrar y cómo mostrarlo y finalmente, la **plantilla** (*template*) corresponde a lo que se muestra en el navegador.

En la Fig.1 se muestra el esquema general del funcionamiento de una aplicación web desarrollada en Django.

Desde el navegador se realiza una petición HTTP, Django la recoge y revisa mediante expresiones regulares alguna coincidencia con las URLs registradas.

Cuando encuentra una coincidencia se dirige a las vistas, aquí se definen funciones que reciben las peticiones y generan una respuesta HTTP. Para satisfacer las peticiones se pueden recuperar datos de los modelos (base de datos) y darles el formato necesario para la respuesta HTTP que se ve reflejada en la plantilla.

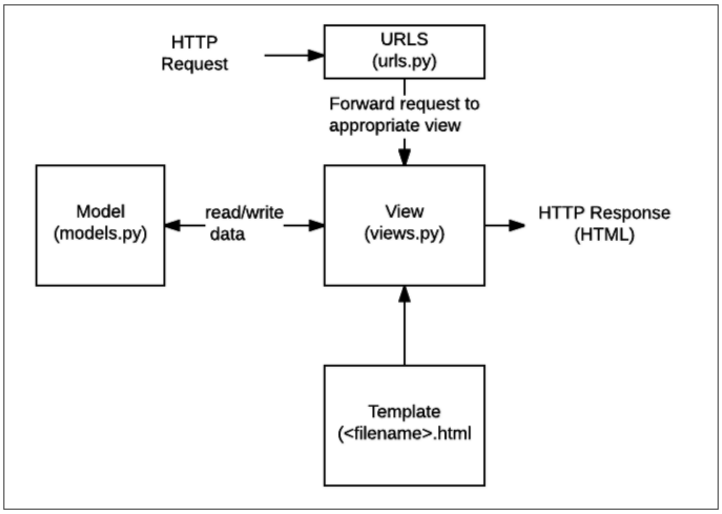


Fig. 1 Estructura funcionamiento Django.

Requerimientos

- Lenguajes de programación: Python, HTML y CSS.
- Sistema Operativo: Windows, Mac o Linux.
- Editor de Texto: A mi me gusta utilizar Sublime Text porque permite editar tanto Python como html o css. (https://www.sublimetext.com/)
- Tener instalada la última versión de Python 3. (https://www.python.org/downloads/)
- La documentación de Django la puedes encontrar aquí: https://docs.djangoproject.com/en/1.11/

Descarga e Instalación

1. Para asegurar la portabilidad de nuestro proyecto debemos trabajar con un entorno virtual, éste nos permitirá tener en un sólo lugar todo lo que necesitamos para nuestra aplicación web. En la consola o símbolo del sistema de nuestro computador, escribimos:

```
>> pip install virtualenv
```

2. Para crear un nuevo entorno virtual se debe crear una carpeta en la ubicación que quieras y luego, abrir la consola en la carpeta para ingresar:

```
>> python –m venv
```

3. Una vez creado el entorno debemos activarlo, como sigue:

```
>> cd Scripts
>>   activate
```

Al activar verás algo como esto en la consola:

```
(myvirtenv) C:\Users\Adm\Desktop\myvirtenv
```

4. Para descargar e instalar Django en el entorno virtual, escribimos:

```
(myvirtenv) C:\Users\Adm\Desktop\myvirtenv> pip install django
```

Tutorial

Para aprender a utilizar Django vamos a desarrollar una aplicación muy sencilla que nos permita ingresar y leer post como en un blog. Vamos a ir paso a paso.

Crear proyecto

El primer paso para desarrollar una aplicación web con Django es activar el entorno virtual como fue realizado en el punto 3 de la sección anterior.

Luego, podemos crear el proyecto (“mysite”) como sigue:

```
>> django-admin startproject mysite
```

Se generarán las siguientes carpetas:

```
mysite/
    manage.py
    mysite/
        __init__.py
        settings.py
        urls.py
        wsgi.py
```

Los que corresponden a:

- **mysite/** (exterior) es la carpeta que contiene tu proyecto. Ésta carpeta no le interesa a Django por lo que si le cambias el nombre no afecta al proyecto.
- **manage.py**: script que te permite interactuar con el proyecto Django.
- **mysite/** (interior) este directorio es el paquete Python para tu proyecto. El nombre es el que utilizarás para importar cualquier elemento que necesites de él.
- **mysite/__init__.py**: archivo vacío que le dice a Python que este directorio debe ser considerado como un paquete Python.
- **mysite/settings.py**: configuraciones para el proyecto.
- **mysite/urls.py**: declaración de las urls que van a ser parte de tu proyecto.
- **mysite/wsgi.py**: se utiliza en producción. Es un punto de entrada para WSGI compatible con servidores web para alojar tu proyecto.

Si estás en la carpeta del entorno virtual, debes ingresar a la carpeta que contiene los componentes de tu proyecto:

```
>> cd mysite
```

Luego, podemos verificar si el proyecto se creó correctamente:

```
>> python manage.py runserver
```

Verás algo así, al ingresar a http://127.0.0.1:8000/ :

Crear carpetas: templates y static

Ahora que nuestro proyecto está funcionando correctamente, vamos a crear en `mysite` dos carpetas: `templates` y `static`, éstas contendrán los archivos html y los archivos necesarios para la personalización, respectivamente. El directorio se verá así:

```
mysite/
    manage.py
```

```
 mysite/
     __init__.py
     settings.py
     urls.py
     wsgi.py
templates/
static/
```

Modificar configuración

Antes de crear la aplicación vamos a hacer algunos cambios en `mysite/settings.py`.

Primero vamos a configurar la zona horaria, para eso encuentra la línea que dice `TIME_ZONE`, la vamos a cambiar por nuestra zona horaria, así:

```
TIME_ZONE = 'America/Santiago'
```

Luego, vamos a definir la ruta para nuestras plantillas. Buscamos `TEMPLATES` y modificamos `'DIRS'` como sigue:

```
'DIRS': [os.path.join(BASE_DIR, 'templates')],
```

Finalmente, agregamos la ruta para los archivos estáticos. Vamos hasta el final del archivo y justo debajo de `STATIC_URL` agregamos:

```
STATICFILES_DIRS = (os.path.join(BASE_DIR, 'static'),)
  STATIC_ROOT = os.path.join(BASE_DIR, 'staticfiles')
```

Configurar Base de Datos

Por defecto el proyecto viene configurado para utilizar Sqlite3, sin embargo, existen múltiples posibilidades dependiendo de lo que quieras realizar. (https://docs.djangoproject.com/en/1.11/ref/databases/)

Las modificaciones deben ser realizadas, aquí:

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

Crear Aplicación

Para crear nuestra aplicación (myapp) debemos ingresar en la consola:

```
>> python manage.py startapp myapp
```

Y se creará un directorio, como el siguiente:

```
myapp/
    __init__.py
    admin.py
```

```
        apps.py
        migrations/
        models.py
        tests.py
        views.py
```

Ahora vamos a `mysite/settings.py` y agregamos la aplicación en INSTALLED_APPS:

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

Crear Plantilla

En la carpeta `templates/` creamos nuestra primera plantilla. Ésta debe ser un archivo *.html* y la llamarémos *index.html.* En tu editor de texto favorito, escribimos lo siguiente:

index.html

```
<!doctype html>
<html>
    <head>
        <title>My App</title>
    </head>
    <body>
        <h1>Hello World!</h1>
        <p>I’m learning Django<p>
    </body>
</html>
```

Si abrimos http://127.0.0.1:8000/ vamos a seguir viendo la misma ventana que antes, esto es porque no le hemos dicho a Python de dónde obtener este archivo.

Añadir URL a `urls.py`

En primer lugar, debemos importar las vistas de nuestra aplicación, de esta forma las urls podrán ser asociadas a ellas.

```
from myapp import views
```

Luego, agregamos la siguiente url en `urlpatterns`:

```
url(r'^$', views.index, name='home'),
```

La expresión anterior se describe como:

- **url():** encierra la entrada para indicar que es una URL
- **r'^$':** es el comienzo de un patrón URL. Se pueden definir patrones diferentes dependiendo de la URL. Se utilizan expresiones regulares.
- **views.index:** indica que se utilizará la función index que se encuentra en `views.py` de la aplicación.
- **name='home':** es opcional. Permite asignar un nombre a esta URL para su posterior utilización.

`urls.py` queda definido como sigue:

```
from django.conf.urls import url
from django.contrib import admin

from myapp import views

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^$', views.index, name='home'),
]
```

Crear la vista en `views.py`

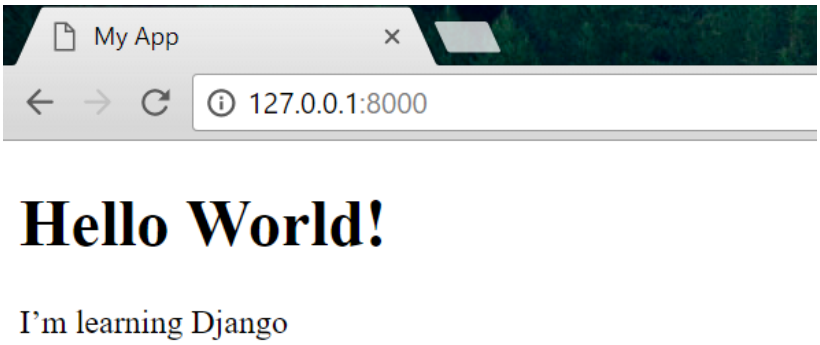
Ahora que tenemos nuestra plantilla y la url definida, necesitamos unirlas de forma que la url muestre la plantilla.

Hay distintas formas de realizar la respuesta http desde las vistas. En esta ocasión utilizaremos la función `render`.

El script `views.py` queda definido como sigue:

```
from django.shortcuts import render

# Create your views here.
def index(request):

# Aquí se realizan los procesamientos de la información para enviar a la plantilla

      return render(request, 'index.html')
```

Ahora, cuando cargamos nuevamente nuestra página obtenemos:



Añadir archivos estáticos

En la carpeta `static/` creamos una nueva carpeta llamada `css.` En ésta última, creamos un archivo `sytle.css.`

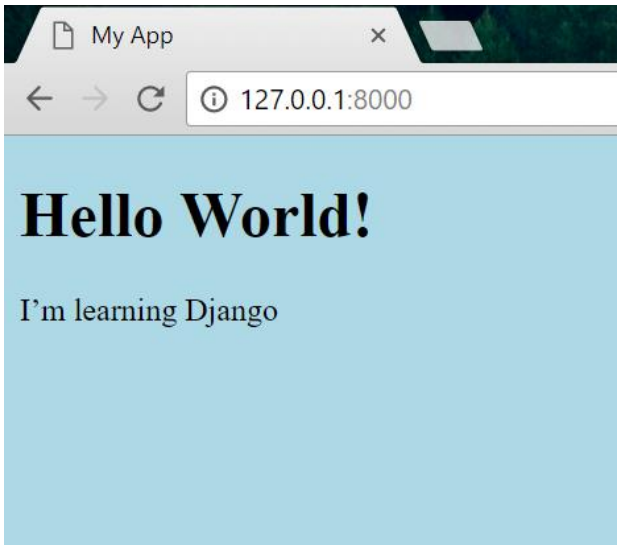
style.css

```
body {
    background-color: lightblue;
}
```

Luego, en index.html, agregamos:

```
{% load staticfiles %}
<!doctype html>
<html>
    <head>
        <title>My App</title>
        <link rel="stylesheet" href="{% static 'css/style.css' %}" />
    </head>
    <body>
        <h1>Hello World!</h1>
        <p>I'm learning Django<p>
    </body>
</html>
```

Y obtenemos, lo siguiente:



Modelos

Para almacenar la información necesaria para nuestra aplicación, creamos un modelo. Los modelos funcionan en base a Objetos.

¿Qué es un objeto? Es un conjunto de propiedades y acciones.

Por ejemplo: si queremos crear un Post para un Blog, entonces vamos a tener que el objeto Post va a tener características como: título, texto y autor.

Vamos a crear el modelo Post. Para esto vamos al archivo `myapp/models.py` y definimos la clase que se referirá al objeto.

```
from django.db import models

class Post(models.Model):
        author = models.CharField(max_length=200)
        title = models.CharField(max_length=200)
        text = models.TextField()
         slug = models.SlugField(max_length=50)
```

Para crear la tabla en la base de datos debemos escribir lo siguiente en la consola:

```
>> python manage.py makemigrations myapp
```

En la consola aparecerá lo siguiente:

```
Migrations for 'myapp':
  myapp\migrations\0001_initial.py
    - Create model Post
```

Luego de crear el archivo, efectuamos la migración:

```
>> python manage.py migrate
```

En la consola se mostrará:

```
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, myapp, sessions
Running migrations:
  Applying myapp.0001_initial... OK
```

Administrador de Django

Para agregar, editar y borrar los elementos de los modelos creados, se puede utilizar el administrador de Django.
En primer lugar, debemos crear un usuario administrador, como sigue:

```
>> python manage.py createsuperuser
```

Te pedirán ingresar un nombre de usuario, correo electrónico y contraseña.

```
Username: admin
Email address: admin@gmail.com
Password:
Password (again):
Superuser created successfully.
```

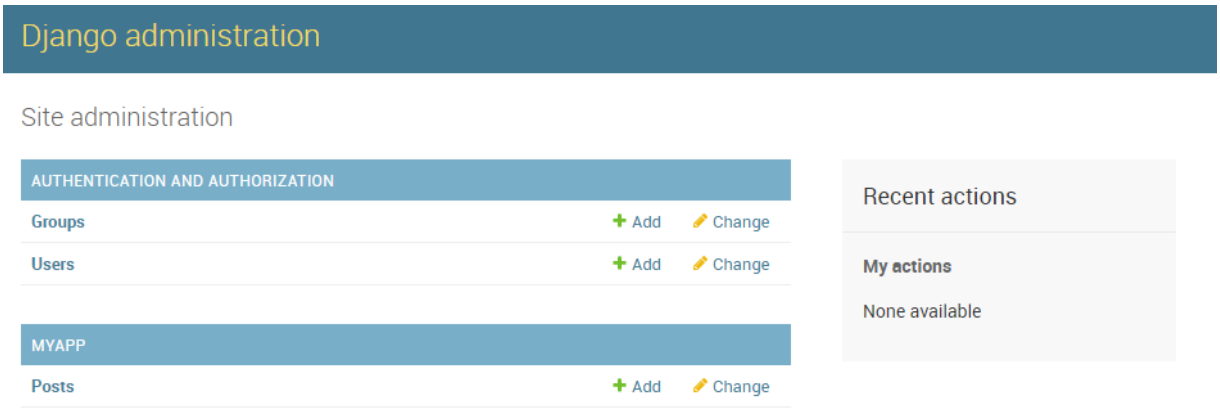
Luego, abrimos el archivo `myapp/admin.py` y reemplazamos su contenido por esto:

admin.py

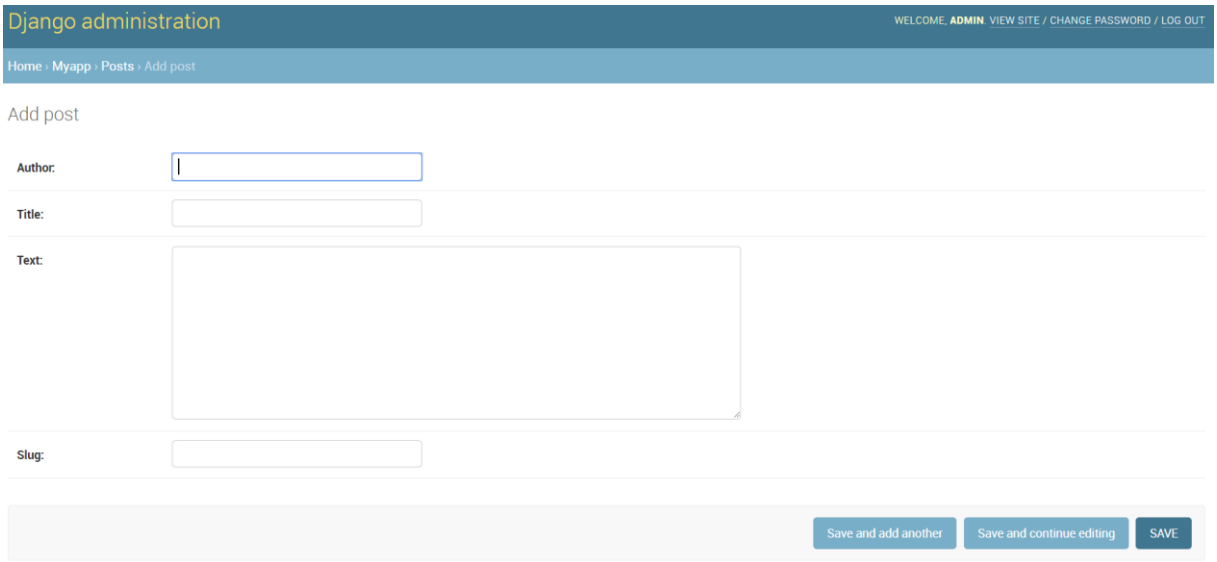
```
from django.contrib import admin

from .models import Post

# Register your models here.
class PostAdmin(admin.ModelAdmin):
      list_display = ('author', 'title', 'text', 'slug',)
      search_fields = ('author', 'title',)
      prepopulated_fields = {'slug': ('title',)}
      class Meta:
          model = Post

admin.site.register(Post, PostAdmin)
```

Corremos nuestra aplicación: `>> python manage.py runserver`

Y verificamos la página de administrador en: http://127.0.0.1:8000/admin. Iniciamos sesión con el usuario creado anteriormente y nos encontramos con lo siguiente:



Si ingresamos a `Posts` y luego presionamos `ADD POST+` en la esquina superior derecha, podremos guardar información en nuestro modelo.



Formularios

Para guardar datos en nuestro modelo desde la aplicación web utilizamos formularios. En la carpeta `myapp/` creamos el archivo `forms.py`. Definimos una clase que va a relacionar el formulario con el modelo recién creado, excluimos el campo slug porque lo vamos a llenar desde la vista y asignamos las etiquetas que se mostrarán para cada campo.

`forms.py`

```
from django import forms
from django.forms import ModelForm
from .models import Post

class PostForm(ModelForm):
    class Meta:
      model = Post
```

```
        exclude = ('slug',)
        labels = {
                    'author': ('Autor'),
                    'title': ('Título'),
                    'text': ('Texto'),
        }
```

Ahora que tenemos nuestro formulario definido debemos generar una plantilla para poder mostrarlo, para esto creamos un archivo que llamaremos `add_post.html` en la carpeta `templates/`.

add_post.html

```
<!DOCTYPE html>
<html>
<head>
        <title>Añadir Post</title>
</head>
<body>
        <div class="row">
                <div class="col-md-8">
                        <div class="page-header">
                                <h1>Añadir Post</h1>
                        </div>
                        <form role="form" method="post">
                                {% csrf_token %}
                                {{ form.as_p }}
                                <input type="submit" name="save" value="Guardar"
                                 class="btn btn-success"/>
                        </form>
                </div>
        </div>
</body>
</html>
```

Ahora que tenemos nuestro formulario y plantilla creados debemos añadir la url en `mysite/urls.py`.

```
             url(r'^add_post/$', views.add_post, name='add_post'),
```

Y definimos la vista en `myapp/views.py`:

```
from django.shortcuts import render, redirect
from .forms import PostForm
from django.utils.text import slugify

# Create your views here.
def index(request):
        # Aquí se realizan los procesamientos de la información para enviar a la
plantilla

        return render(request, 'index.html')

def add_post(request):
```

```
form_class = PostForm

if request.method =='POST':

	form = form_class(request.POST)

	if form.is_valid():
		save_1 = form.save(commit=False)
		save_1.slug = slugify(request.POST['title'])

		save_1.save()

		return redirect("home")
else: form = form_class()

context_data = {
	"form":form,
}
return render(request, "add_post.html", context_data)
```

En `myapp/views.py` importamos el formulario y las funciones que utilizaremos. Luego, en la función `add_post` definimos el formulario que vamos a utilizar y guardamos la información. En el caso del campo `slug` no lo mostramos en el formulario puesto que vamos a llenarlo nosotros haciendo un slugify del título ingresado, como el slug es único lo utilizaremos posteriormente para generar las url para cada elemento. Una vez que el usuario guarda el objeto se redirecciona a la página de inicio.

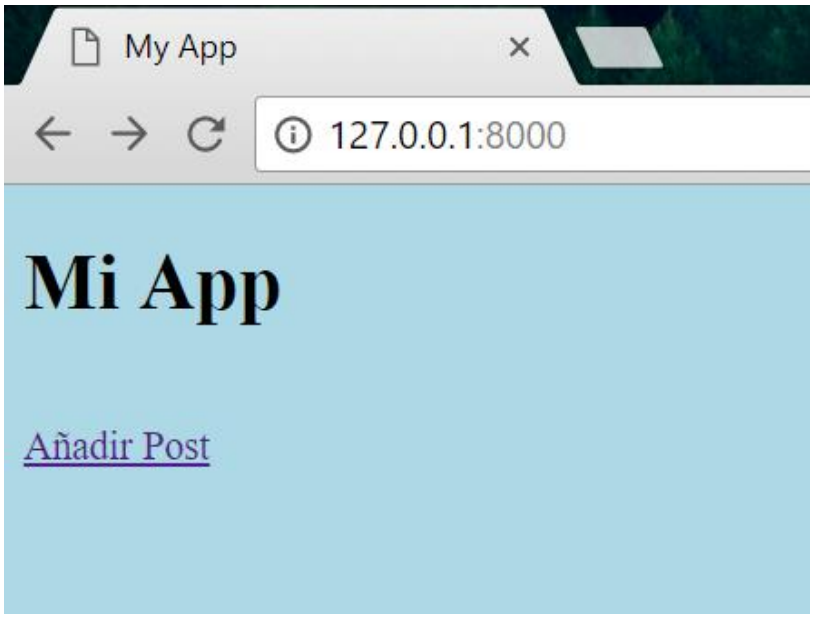
Finalmente, añadimos un hipervínculo en el inicio que nos lleve a `add_post.html`.

```
<a href="{% url 'add_post' %}">Añadir Post</a>
```

Luego, la plantilla nos resulta así:

add_post.html

```
{% load staticfiles %}

<!doctype html>
<html>
    <head>
        <title>My App</title>
        <link rel="stylesheet" href="{% static 'css/style.css' %}" />
    </head>
    <body>
        <h1>Mi App</h1>
       </br>
        <a href="{% url 'add_post' %}">Añadir Post</a>
    </body>
</html>
```



Mostrar Posts en index.html

Ahora que tenemos ingresados objetos en nuestro modelo podemos mostrarlos en nuestra vista principal, para esto debemos modificar nuestra función index en `myapp/views.py`.

```
from .models import Post

def index(request):
      posts = Post.objects.all()

      context_data = {
            "object_list": posts,
      }

      return render(request, 'index.html', context_data)
```

Importamos nuestro modelo y recuperamos todos los objetos que posee. Luego, en `context_data` enviamos la información a la plantilla definida como sigue:

index.html

```
{% load staticfiles %}
<!doctype html>
<html>
    <head>
        <title>My App</title>
        <link rel="stylesheet" href="{% static 'css/style.css' %}" />
    </head>
    <body>
        <h1>Mi App</h1>
        </br>
        <a href="{% url 'add_post' %}">Añadir Post</a>
        </br>
        <div class="col-sm-24">
            <div class="row">
                {% for obj in object_list %}
                    <div class="col-sm-12">
                        <div>
                            <h2>{{obj.title}}</h2>
                        </div>
                    </div>
                {% endfor %}
            </div>
        </div>
    </body>
</html>
```

Para mostrar los post recorremos mediante un ciclo los objetos contenidos en el modelo, luego accedemos al detalle de cada uno según el nombre de cada campo. Así si queremos acceder al título se escribe `obj.title`, si queremos acceder al autor `obj.author` y si queremos acceder al texto `obj.text`.

Leer cada Post

Si queremos leer el detalle de lo que contiene cada post, debemos en primer lugar definir la URL que nos va a llevar a la plantilla que lo contiene. Como cada post es diferente, utilizaremos el slug que definimos anteriormente de la siguiente forma:

```
url(r'^(?P<slug>[-\w]+)/$', views.post_detail, name='post_detail'),
```

Luego, definimos la vista `post_detail,` en donde además de recibir request vamos a recibir el slug que nos va a permitir encontrar el post seleccionado.

```
def post_detail(request, slug):
    post = Post.objects.get(slug=slug)

    context_data = {
        "title":post.title,
        "author":post.author,
        "text":post.text,
    }
    return render(request, "post_detail.html", context_data)
```

Escribimos el template para esta vista:

post_detail.html

```
<!doctype html>
<html>
    <head>
        <title>{{ title }}</title>
    </head>
    <body>
        <h1>{{title}}</h1>
        </br>
        <small>{{ author }}</small>
        </br>
        </br>
        <p>{{ text }}</p>
    </body>
</html>
```

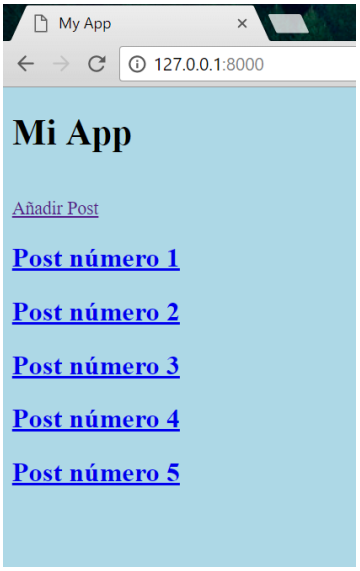
Y finalmente, agregamos los vínculos para ingresar a cada post desde el inicio, así:

```
        <a href="{% url 'post_detail' slug=obj.slug %}">{{obj.title}}</a>
```

Cabe destacar, que desde el hipervínculo debemos enviar el slug del post para que la vista lo reciba y pueda encontrar el post seleccionado.

index.html

```
{% load staticfiles %}
<!doctype html>
<html>
    <head>
        <title>My App</title>
        <link rel="stylesheet" href="{% static 'css/style.css' %}" />
    </head>
    <body>
        <h1>Mi App</h1>
        </br>
        <a href="{% url 'add_post' %}">Añadir Post</a>
        </br>
       <div class="col-sm-24">
              <div class="row">
                     {% for obj in object_list %}
                     <div class="col-sm-12">
                            <div>
                                <h2><a href="{% url 'post_detail' slug=obj.slug%}"
                                  >{{obj.title}}</a>
                                </h2>
                            </div>
                     </div>
                     {% endfor %}
              </div>
       </div>
    </body>
</html>
```



Complementos que se pueden utilizar

Bootstrap: https://getbootstrap.com/docs/3.3/components/

Bootstrap es uno de los frameworks HTML y CSS más populares para desarrollar webs con mejor apariencia.

Para instalarlo debes añadir las siguientes líneas a tu html:

```
<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/boo
tstrap.min.css">
<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/boo
tstrap-theme.min.css">
```

Y luego, puedes incorporar cualquiera de los componentes que posee en tu plantilla.

Chart.js: http://www.chartjs.org/docs/latest/

Permite incorporar a tu aplicación gráficos javascript a tu aplicación web. Para utilizarlo tienes que agregar lo siguiente a tu plantilla:

```
<script src="//cdnjs.cloudflare.com/ajax/libs/Chart.js/0.2.0/Chart.min.js"
type="text/javascript"></script>
<script src="//cdnjs.cloudflare.com/ajax/libs/jquery/2.1.1/jquery.min.js"
type="text/javascript"></script>


<canvas id="myChart" width="400" height="400"></canvas>

<script type="text/javascript">
      $(document).ready(function(){
var ctx = document.getElementById("myChart").getContext('2d');
var myChart = new Chart(ctx, {
    type: 'bar',
    data: {
        labels: ["Red", "Blue", "Yellow", "Green", "Purple", "Orange"],
```

```
        datasets: [{
            label: '# of Votes',
            data: [12, 19, 3, 5, 2, 3],
            backgroundColor: [
                'rgba(255, 99, 132, 0.2)',
                'rgba(54, 162, 235, 0.2)',
                'rgba(255, 206, 86, 0.2)',
                'rgba(75, 192, 192, 0.2)',
                'rgba(153, 102, 255, 0.2)',
                'rgba(255, 159, 64, 0.2)'
            ],
            borderColor: [
                'rgba(255,99,132,1)',
                'rgba(54, 162, 235, 1)',
                'rgba(255, 206, 86, 1)',
                'rgba(75, 192, 192, 1)',
                'rgba(153, 102, 255, 1)',
                'rgba(255, 159, 64, 1)'
            ],
            borderWidth: 1
        }]
    },
    options: {
        scales: {
            yAxes: [{
                ticks: {
                    beginAtZero:true
                }
            }]
        }
    }
});
})
</script>
```

Puedes encontrar más ejemplos, configuraciones y tipos de gráficos en la página de chart js.