

# Curso de Django

## Módulo 3: Vistas y Plantillas.

Lcdo. Diego Medardo Saavedra García. Mgtr.

2023-08-01

### Tabla de contenidos

<b>1</b>	<b>Módulo 3: Vistas y Plantillas.</b>	<b>2</b>
1.1	Creación de Vistas en Django . . . . .	2
1.2	Otra forma de generar un Hola Mundo en Django. . . . .	3
1.3	Mostrar Publicaciones y Comentarios . . . . .	5
1.4	Crea una vista para mostrar las publicaciones: . . . . .	5
1.5	Sistema de Plantillas de Django (Jinja2) . . . . .	7
1.6	Conceptos Principales del Sistema de Plantillas de Django: . . . . .	7
1.7	Pasos para Utilizar el Sistema de Plantillas de Django en el Proyecto “blog”: . . . . .	8
<b>2</b>	<b>CRUD de Publicaciones</b>	<b>10</b>
2.1	Crear Publicaciones . . . . .	10
2.2	Leer Publicaciones . . . . .	11
2.3	Actualizar Publicaciones . . . . .	12
2.4	Eliminar Publicaciones . . . . .	14
2.5	CRUD de Comentarios . . . . .	15
2.6	Leer Comentarios . . . . .	17
2.7	Actualizar Comentarios . . . . .	18
2.8	Eliminar Comentarios . . . . .	20
2.9	Migrar y Ejecutar el Servidor . . . . .	21
<b>3</b>	<b>Correcciones</b>	<b>22</b>
3.1	Integración de Botones. . . . .	22
3.2	Para las Publicaciones: . . . . .	22
3.3	Para los Comentarios: . . . . .	25
3.4	Corregir el Modelo . . . . .	29
3.5	Backend: Construcción de una API con Django Rest Framework . . . . .	30
3.6	Documentación de las API . . . . .	32

<b>4</b>	<b>FRONTEND: Herencia de Plantillas con Django.</b>	<b>34</b>
4.1	Herencia de Plantillas . . . . .	34
4.2	Modificación de Views. . . . .	36
4.3	Bootstrap en base.html . . . . .	38
4.4	Agregando bootstrap a los templates. . . . .	39
<b>5</b>	<b>Reactjs</b>	<b>41</b>
5.1	Frontend con Reactjs . . . . .	41
5.1.1	1. index.css . . . . .	49
5.1.2	2. App.css . . . . .	49
5.1.3	3. Importar los archivos CSS en App.js . . . . .	50
5.2	Inconvenientes . . . . .	52

## 1 Módulo 3: Vistas y Plantillas.

### 1.1 Creación de Vistas en Django

Las vistas en Django son funciones que procesan las solicitudes del usuario y devuelven una respuesta HTTP. Cada vista debe recibir una solicitud como argumento y devolver una respuesta.

Ejemplo de una vista que muestra un mensaje de bienvenida:

**Paso 1:** En el archivo “views.py” de la aplicación “publicaciones”, agrega el código de la vista de bienvenida:

```
# publicaciones/views.py

from django.http import HttpResponse

def vista_bienvenida(request):
    return HttpResponse("¡Bienvenido al blog!")
```

**Paso 3:** Configura la URL para la vista en el archivo “urls.py” de la aplicación “publicaciones”:

```
# publicaciones/urls.py

from django.urls import path
from . import views

urlpatterns = [
```

```

    path('bienvenida/', views.vista_bienvenida, name='vista_bienvenida'),
]

```

**Paso 4:** Configura la URL de la aplicación **publicaciones** en el archivo **urls.py** del proyecto:

```

# blog/urls.py

from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('publicaciones/', include('publicaciones.urls')), # Agrega esta línea para incluir
]

```

**Paso 5:** Ahora, ejecuta el servidor de desarrollo con el siguiente comando:

```
python manage.py runserver
```

**Paso 6:** Abre tu navegador web e ingresa a la siguiente dirección:

<http://127.0.0.1:8000/publicaciones/bienvenida/>

Deberías ver el mensaje “¡Bienvenido al blog!” en el navegador.

## 1.2 Otra forma de generar un Hola Mundo en Django.

**Paso 1:** Creación de la vista en views.py:

En esta etapa, se crea una vista llamada “HolaMundoView” utilizando la clase `TemplateView`. Esta vista simplemente renderiza la plantilla “hola\_mundo.html” que muestra un mensaje “Hola Mundo!”.

```

# views.py

from django.shortcuts import render
from django.views.generic import TemplateView

class HolaMundoView(TemplateView):
    template_name = 'hola_mundo.html'

```

**Paso 2:** Configuración de las URLs en urls.py:

En el archivo `urls.py` de la aplicación “publicaciones”, se define la URL para la vista “HolaMundoView”. También se incluyen las URLs de la aplicación en las URLs globales del proyecto.

```
# urls.py de publicaciones

from django.urls import path
from .views import HolaMundoView

urlpatterns = [
    path('hola_mundo/', HolaMundoView.as_view(), name='hola_mundo'), ]
```

### **Paso 3** Creación de la plantilla `hola_mundo.html`:

La plantilla “`hola_mundo.html`” es un archivo HTML simple que muestra el mensaje “Hola Mundo!” en un encabezado `h1`.

```
<!DOCTYPE html>
<html>
<head>
<title>Hola Mundo</title>
</head>
<body>
<h1>Hola Mundo!</h1>
</body>
</html>
```

### **Paso 4:** Configuración de las URLs globales del proyecto en `urls.py`:

En el archivo `urls.py` del proyecto principal, se incluye la URL de la aplicación “publicaciones” utilizando la función “`include`”. Esto permitirá acceder a las URLs de la aplicación a través de la URL base “`publicaciones/`”.

```
# urls.py del proyecto
from django.contrib import admin
from django.urls import path, include

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

Con estos pasos, hemos creado una aplicación simple que muestra el mensaje “Hola Mundo!” en la página cuando accedemos a la URL “publicaciones/hola\_mundo/”. Además, hemos configurado la conexión entre las URLs de la aplicación y las URLs globales del proyecto. A partir de aquí, podemos agregar más funcionalidades y vistas a nuestra aplicación utilizando los modelos “Publicacion” y “Comentario”.

### 1.3 Mostrar Publicaciones y Comentarios

Para mostrar las publicaciones y comentarios agregados en los modelos, primero, asegúrate de que hayas definido correctamente los modelos “Publicacion” y “Comentario” en el archivo `models.py` de la aplicación “publicaciones” como se mostró en ejemplos anteriores.

### 1.4 Crea una vista para mostrar las publicaciones:

En el archivo `views.py` de la aplicación “publicaciones”, crea una vista llamada “ListaPublicacionesView” para mostrar todas las publicaciones:

```
# publicaciones/views.py

from django.views.generic import ListView
from .models import Publicacion

class ListaPublicacionesView(ListView):
    model = Publicacion
    template_name = 'lista_publicaciones.html'
    context_object_name = 'publicaciones'
```

Crea una plantilla para mostrar la lista de publicaciones:

Crea un archivo llamado “lista\_publicaciones.html” dentro de la carpeta “templates” de la aplicación “publicaciones”:

```
<!-- publicaciones/templates/lista_publicaciones.html -->

<!DOCTYPE html>
<html>
<head>
    <title>Lista de Publicaciones</title>
</head>
<body>
    <h1>Lista de Publicaciones</h1>
```

```

<ul>
    {% for publicacion in publicaciones %}
        <li>{{ publicacion.titulo }}</li>
        <ul>
            {% for comentario in publicacion.comentarios.all %}
                <li>{{ comentario.texto }}</li>
            {% endfor %}
        </ul>
    {% endfor %}
</ul>
</body>
</html>

```

En este ejemplo, estamos utilizando una estructura de bucles for en la plantilla para mostrar las publicaciones y sus comentarios asociados.

Configura las URLs para mostrar la lista de publicaciones:

En el archivo `urls.py` de la aplicación “publicaciones”, agrega la configuración para mostrar la lista de publicaciones:

```

# publicaciones/urls.py

from django.urls import path
from .views import ListaPublicacionesView

urlpatterns = [
    path('publicaciones/', ListaPublicacionesView.as_view(), name='lista_publicaciones'),
]

```

Actualiza las URLs del proyecto:

En el archivo `urls.py` del proyecto “blog”, actualiza las URLs de la aplicación “publicaciones” para que se muestren en la ruta principal:

```

# blog/urls.py

from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('publicaciones.urls')),
]

```

]

Ejecuta el servidor de desarrollo:

Ahora, ejecuta el servidor de desarrollo nuevamente con el siguiente comando:

```
python manage.py runserver
```

Accede a la URL <http://localhost:8000/publicaciones/> en tu navegador y deberías ver la lista de publicaciones y sus comentarios asociados.

Si no has agregado publicaciones o comentarios en la base de datos, es posible que no veas datos en la lista.

¡Listo! Ahora has configurado el proyecto “blog” para mostrar las publicaciones y comentarios agregados en los modelos “Publicacion” y “Comentario” utilizando el modelo Template View de Django.

## 1.5 Sistema de Plantillas de Django (Jinja2)

- Django utiliza el motor de plantillas Jinja2 para gestionar la presentación de los datos en las vistas.
- Jinja2 es un poderoso motor de plantillas que permite incrustar código Python y generar HTML de forma dinámica.
- En el contexto del proyecto “blog” que hemos estado desarrollando, el sistema de plantillas de Django se encargará de renderizar las vistas y mostrar las publicaciones con sus comentarios en la plantilla “lista\_publicaciones.html”.

## 1.6 Conceptos Principales del Sistema de Plantillas de Django:

**Templates:** Los templates son archivos HTML que contienen código Python que define cómo se mostrarán los datos en la interfaz de usuario.

En nuestro caso, el archivo “lista\_publicaciones.html” será un template donde mostraremos la lista de publicaciones y sus comentarios.

**Contexto:** El contexto es un diccionario de Python que contiene los datos que se van a renderizar en el template.

En este contexto, proporcionaremos la lista de publicaciones y sus comentarios para que sean mostrados en la plantilla.

**Variables de Plantilla:** En los templates de Django, podemos utilizar variables de plantilla para acceder a los datos proporcionados en el contexto.

Por ejemplo, podemos utilizar la variable publicaciones para acceder a la lista de publicaciones y sus comentarios.

**Directivas de Control:** Jinja2 permite utilizar directivas de control, como bucles y condicionales, en los templates para generar contenido de forma dinámica.

Esto nos permite iterar sobre la lista de publicaciones y mostrar cada una de ellas con sus comentarios.

## 1.7 Pasos para Utilizar el Sistema de Plantillas de Django en el Proyecto “blog”:

Pasos para Utilizar el Sistema de Plantillas de Django en el Proyecto “blog”:

**Paso 1:** Crear el archivo “lista\_publicaciones.html”

En el directorio “templates” de la aplicación “publicaciones”, creamos el archivo “lista\_publicaciones.html” donde definiremos la estructura HTML y utilizaremos las variables de plantilla para mostrar los datos.

```
<!-- lista_publicaciones.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Lista de Publicaciones</title>
</head>
<body>
    <h1>Lista de Publicaciones</h1>
    <ul>
        {% for publicacion in publicaciones %}
        <li>
            <h2>{{ publicacion.titulo }}</h2>
            <p>{{ publicacion.contenido }}</p>
            <h3>Comentarios:</h3>
            <ul>
                {% for comentario in publicacion.comentarios.all %}
                <li>{{ comentario.texto }}</li>
                {% endfor %}
            </ul>
        </li>
        {% endfor %}
    </ul>
```



```
</body>
</html>
```

## Paso 2: Definir las Vistas

En el archivo “views.py” de la aplicación “publicaciones”, definimos las vistas que serán responsables de obtener los datos de la base de datos (en este caso, las publicaciones y sus comentarios) y pasarlos al template.

```
# views.py
from django.shortcuts import render
from .models import Publicacion

def lista_publicaciones(request):
    publicaciones = Publicacion.objects.all()
    context = {'publicaciones': publicaciones}
    return render(request, 'lista_publicaciones.html', context)
```

## Paso 3: Utilizar el Contexto

En las vistas, creamos un contexto que contiene los datos que queremos mostrar en el template. En nuestro caso, el contexto contendrá la lista de publicaciones y sus comentarios.

## Paso 4: Renderizar el Template

Finalmente, en las vistas, utilizamos el método render() para renderizar el template “lista\_publicaciones.html” con el contexto que creamos. Esto generará el contenido HTML dinámico que mostrará las publicaciones y sus comentarios.

## Paso 5: Configurar las URLs

En el archivo “urls.py” de la aplicación “publicaciones”, definimos la URL que se utilizará para acceder a la vista que renderiza el template “lista\_publicaciones.html”.

```
# urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('lista/', views.lista_publicaciones, name='lista_publicaciones'),
]
```

Con estos pasos, habremos integrado el sistema de plantillas de Django (Jinja2) en nuestro proyecto “blog” y podremos mostrar de forma dinámica las publicaciones y sus comentarios

en la plantilla “lista\_publicaciones.html”. Al acceder a la URL “/lista/”, se mostrará la lista de publicaciones con sus comentarios.

## 2 CRUD de Publicaciones

### 2.1 Crear Publicaciones

**Paso 1:** Crear el formulario de Publicación

En el archivo “forms.py” de la aplicación “publicaciones”, creamos un formulario para la creación de publicaciones.

```
# forms.py
from django import forms
from .models import Publicacion

class PublicacionForm(forms.ModelForm):
    class Meta:
        model = Publicacion
        fields = ['titulo', 'contenido', 'autor']
```

**Paso 2:** Actualizar el archivo “views.py”

En el archivo “views.py” de la aplicación “publicaciones”, definimos una vista para crear una nueva publicación y renderizar el formulario.

```
# views.py
from django.shortcuts import render, redirect
from .forms import PublicacionForm

def crear_publicacion(request):
    if request.method == 'POST':
        form = PublicacionForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect('lista_publicaciones')
    else:
        form = PublicacionForm()
    return render(request, 'crear_publicacion.html', {'form': form})
```

**Paso 3:** Crear la plantilla para el formulario de creación

En el directorio “templates” de la aplicación “publicaciones”, creamos el archivo “crear\_publicacion.html” que contendrá el formulario de creación de publicaciones.

```
<!-- crear_publicacion.html -->
<!DOCTYPE html>
<html>
<head>
  <title>Crear Publicación</title>
</head>
<body>
  <h1>Crear Nueva Publicación</h1>
  <form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">Crear</button>
  </form>
</body>
</html>
```

#### Paso 4: Configurar las URLs

En el archivo “urls.py” de la aplicación “publicaciones”, agregamos una URL para la vista de creación de publicaciones.

```
# urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('lista/', views.lista_publicaciones, name='lista_publicaciones'),
    path('crear/', views.crear_publicacion, name='crear_publicacion'),
]
```

## 2.2 Leer Publicaciones

#### Paso 1: Actualizar el archivo “views.py”

En el archivo “views.py” de la aplicación “publicaciones”, definimos una vista para mostrar la lista de publicaciones.

```
# views.py
from django.shortcuts import render
```

```

from .models import Publicacion

def lista_publicaciones(request):
    publicaciones = Publicacion.objects.all()
    return render(request, 'lista_publicaciones.html', {'publicaciones': publicaciones})

```

**Paso 2:** Actualizar el archivo “lista\_publicaciones.html”

En la plantilla “lista\_publicaciones.html”, podemos acceder a las publicaciones y mostrarlas en una lista.

```

<!-- lista_publicaciones.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Lista de Publicaciones</title>
</head>
<body>
    <h1>Lista de Publicaciones</h1>
    <ul>
        {% for publicacion in publicaciones %}
        <li>
            <h2>{{ publicacion.titulo }}</h2>
            <p>{{ publicacion.contenido }}</p>
            <h3>Comentarios:</h3>
            <ul>
                {% for comentario in publicacion.comentarios.all %}
                <li>{{ comentario.texto }}</li>
                {% endfor %}
            </ul>
        </li>
        {% endfor %}
    </ul>
</body>
</html>

```

## 2.3 Actualizar Publicaciones

**Paso 1:** Actualizar el archivo “views.py”

En el archivo “views.py” de la aplicación “publicaciones”, definimos una vista para actualizar una publicación existente.

```

# views.py
from django.shortcuts import render, get_object_or_404, redirect
from .forms import PublicacionForm
from .models import Publicacion

def actualizar_publicacion(request, pk):
    publicacion = get_object_or_404(Publicacion, pk=pk)
    if request.method == 'POST':
        form = PublicacionForm(request.POST, instance=publicacion)
        if form.is_valid():
            form.save()
            return redirect('lista_publicaciones')
    else:
        form = PublicacionForm(instance=publicacion)
    return render(request, 'actualizar_publicacion.html', {'form': form})

```

### Paso 2: Crear la plantilla para el formulario de actualización

En el directorio “templates” de la aplicación “publicaciones”, creamos el archivo “actualizar\_publicacion.html” que contendrá el formulario de actualización de publicaciones.

```

<!-- actualizar_publicacion.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Actualizar Publicación</title>
</head>
<body>
    <h1>Actualizar Publicación</h1>
    <form method="post">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Guardar Cambios</button>
    </form>
</body>
</html>

```

### Paso 3: Configurar las URLs

En el archivo “urls.py” de la aplicación “publicaciones”, agregamos una URL para la vista de actualización de publicaciones.

```
# urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('lista/', views.lista_publicaciones, name='lista_publicaciones'),
    path('crear/', views.crear_publicacion, name='crear_publicacion'),
    path('actualizar/<int:pk>/', views.actualizar_publicacion, name='actualizar_publicacion')
]
```

## 2.4 Eliminar Publicaciones

**Paso 1:** Actualizar el archivo “views.py”

En el archivo “views.py” de la aplicación “publicaciones”, definimos una vista para eliminar una publicación existente.

```
# views.py
from django.shortcuts import get_object_or_404, redirect
from .models import Publicacion

def eliminar_publicacion(request, pk):
    publicacion = get_object_or_404(Publicacion, pk=pk)
    if request.method == 'POST':
        publicacion.delete()
        return redirect('lista_publicaciones')
    return render(request, 'eliminar_publicacion.html', {'publicacion': publicacion})
```

**Paso 2:** Crear la plantilla para confirmar la eliminación

En el directorio “templates” de la aplicación “publicaciones”, creamos el archivo “eliminar\_publicacion.html” que contendrá la confirmación para eliminar la publicación.

```
<!-- eliminar_publicacion.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Eliminar Publicación</title>
</head>
<body>
    <h1>Eliminar Publicación</h1>
    <p>¿Estás seguro de que deseas eliminar la publicación "{ publicacion.titulo }"?</p>
```

```

    <form method="post">
        {% csrf_token %}
        <button type="submit">Eliminar</button>
    </form>
</body>
</html>

```

### Paso 3: Configurar las URLs

En el archivo “urls.py” de la aplicación “publicaciones”, agregamos una URL para la vista de eliminación de publicaciones.

```

# urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('lista/', views.lista_publicaciones, name='lista_publicaciones'),
    path('crear/', views.crear_publicacion, name='crear_publicacion'),
    path('actualizar/<int:pk>/', views.actualizar_publicacion, name='actualizar_publicacion'),
    path('eliminar/<int:pk>/', views.eliminar_publicacion, name='eliminar_publicacion'),
]

```

## 2.5 CRUD de Comentarios

El CRUD de comentarios sigue un proceso similar al CRUD de publicaciones. A continuación, se describen los pasos para cada operación: Crear Comentarios

### Paso 1: Crear el formulario de Comentario

En el archivo “forms.py” de la aplicación “publicaciones”, creamos un formulario para la creación de comentarios.

```

# forms.py
from django import forms
from .models import Comentario

class ComentarioForm(forms.ModelForm):
    class Meta:
        model = Comentario
        fields = ['publicacion', 'autor', 'contenido']

```

### Paso 2: Actualizar el archivo “views.py”

En el archivo “views.py” de la aplicación “publicaciones”, definimos una vista para crear un nuevo comentario y renderizar el formulario.

```
# views.py
from django.shortcuts import render, redirect
from .forms import ComentarioForm

def crear_comentario(request):
    if request.method == 'POST':
        form = ComentarioForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect('lista_publicaciones')
    else:
        form = ComentarioForm()
    return render(request, 'crear_comentario.html', {'form': form})
```

### Paso 3: Crear la plantilla para el formulario de creación

En el directorio “templates” de la aplicación “publicaciones”, creamos el archivo “crear\_comentario.html” que contendrá el formulario de creación de comentarios.

```
<!-- crear_comentario.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Crear Comentario</title>
</head>
<body>
    <h1>Crear Nuevo Comentario</h1>
    <form method="post">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Crear</button>
    </form>
</body>
</html>
```

### Paso 4: Configurar las URLs

En el archivo “urls.py” de la aplicación “publicaciones”, agregamos una URL para la vista de creación de comentarios.



```
# urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('lista/', views.lista_publicaciones, name='lista_publicaciones'),
    path('crear/', views.crear_publicacion, name='crear_publicacion'),
    path('actualizar/<int:pk>/', views.actualizar_publicacion, name='actualizar_publicacion'),
    path('eliminar/<int:pk>/', views.eliminar_publicacion, name='eliminar_publicacion'),
    path('crear_comentario/', views.crear_comentario, name='crear_comentario'),
]
```

## 2.6 Leer Comentarios

Paso 1: Actualizar el archivo “views.py”

En el archivo “views.py” de la aplicación “publicaciones”, definimos una vista para mostrar la lista de comentarios.

```
# views.py
from django.shortcuts import render
from .models import Comentario

def lista_comentarios(request):
    comentarios = Comentario.objects.all()
    return render(request, 'lista_comentarios.html', {'comentarios': comentarios})
```

Paso 2: Actualizar el archivo “lista\_comentarios.html”

En la plantilla “lista\_comentarios.html”, podemos acceder a los comentarios y mostrarlos en una lista.

```
<!-- lista_comentarios.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Lista de Comentarios</title>
</head>
<body>
    <h1>Lista de Comentarios</h1>
    <ul>
        {% for comentario in comentarios %}
```

```

        <li>
            <p>Comentario de {{ comentario.autor }} en {{ comentario.publicacion.titulo }}</p>
            <p>{{ comentario.contenido }}</p>
        </li>
    {% endfor %}
</ul>
</body>
</html>

```

### Paso 3: Configurar las URLs

En el archivo “urls.py” de la aplicación “publicaciones”, agregamos una URL para la vista de lista de comentarios.

```

# urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('lista/', views.lista_publicaciones, name='lista_publicaciones'),
    path('crear/', views.crear_publicacion, name='crear_publicacion'),
    path('actualizar/<int:pk>/', views.actualizar_publicacion, name='actualizar_publicacion'),
    path('eliminar/<int:pk>/', views.eliminar_publicacion, name='eliminar_publicacion'),
    path('crear_comentario/', views.crear_comentario, name='crear_comentario'),
    path('lista_comentarios/', views.lista_comentarios, name='lista_comentarios'),
]

```

## 2.7 Actualizar Comentarios

### Paso 1: Actualizar el archivo “views.py”

En el archivo “views.py” de la aplicación “publicaciones”, definimos una vista para actualizar un comentario existente.

```

# views.py
from django.shortcuts import render, get_object_or_404, redirect
from .forms import ComentarioForm
from .models import Comentario

def actualizar_comentario(request, pk):
    comentario = get_object_or_404(Comentario, pk=pk)
    if request.method == 'POST':

```

```

        form = ComentarioForm(request.POST, instance=comentario)
        if form.is_valid():
            form.save()
            return redirect('lista_comentarios')
    else:
        form = ComentarioForm(instance=comentario)
    return render(request, 'actualizar_comentario.html', {'form': form})

```

## Paso 2: Crear la plantilla para el formulario de actualización

En el directorio “templates” de la aplicación “publicaciones”, creamos el archivo “actualizar\_comentario.html” que contendrá el formulario de actualización de comentarios.

```

<!-- actualizar_comentario.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Actualizar Comentario</title>
</head>
<body>
    <h1>Actualizar Comentario</h1>
    <form method="post">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Guardar Cambios</button>
    </form>
</body>
</html>

```

## Paso 3: Configurar las URLs

En el archivo “urls.py” de la aplicación “publicaciones”, agregamos una URL para la vista de actualización de comentarios.

```

# urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('lista/', views.lista_publicaciones, name='lista_publicaciones'),
    path('crear/', views.crear_publicacion, name='crear_publicacion'),
    path('actualizar/<int:pk>/', views.actualizar_publicacion, name='actualizar_publicacion'),
    path('eliminar/<int:pk>/', views.eliminar_publicacion, name='eliminar_publicacion'),

```

```

    path('crear_comentario/', views.crear_comentario, name='crear_comentario'),
    path('lista_comentarios/', views.lista_comentarios, name='lista_comentarios'),
    path('actualizar_comentario/<int:pk>/', views.actualizar_comentario, name='actualizar_
]

```

## 2.8 Eliminar Comentarios

**Paso 1:** Actualizar el archivo “views.py”

En el archivo “views.py” de la aplicación “publicaciones”, definimos una vista para eliminar un comentario existente.

```

# views.py
from django.shortcuts import get_object_or_404, redirect
from .models import Comentario

def eliminar_comentario(request, pk):
    comentario = get_object_or_404(Comentario, pk=pk)
    if request.method == 'POST':
        comentario.delete()
        return redirect('lista_comentarios')
    return render(request, 'eliminar_comentario.html', {'comentario': comentario})

```

**Paso 2:** Crear la plantilla para confirmar la eliminación

En el directorio “templates” de la aplicación “publicaciones”, creamos el archivo “eliminar\_comentario.html” que contendrá la confirmación para eliminar el comentario.

```

<!-- eliminar_comentario.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Eliminar Comentario</title>
</head>
<body>
    <h1>Eliminar Comentario</h1>
    <p>¿Estás seguro de que deseas eliminar el comentario de "{ { comentario.autor }}" en "
    <form method="post">
        {% csrf_token %}
        <button type="submit">Eliminar</button>
    </form>

```

```
</body>
</html>
```

### Paso 3: Configurar las URLs

En el archivo “urls.py” de la aplicación “publicaciones”, agregamos una URL para la vista de eliminación de comentarios.

```
# urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('lista/', views.lista_publicaciones, name='lista_publicaciones'),
    path('crear/', views.crear_publicacion, name='crear_publicacion'),
    path('actualizar/<int:pk>/', views.actualizar_publicacion, name='actualizar_publicacion'),
    path('eliminar/<int:pk>/', views.eliminar_publicacion, name='eliminar_publicacion'),
    path('crear_comentario/', views.crear_comentario, name='crear_comentario'),
    path('lista_comentarios/', views.lista_comentarios, name='lista_comentarios'),
    path('actualizar_comentario/<int:pk>/', views.actualizar_comentario, name='actualizar_comentario'),
    path('eliminar_comentario/<int:pk>/', views.eliminar_comentario, name='eliminar_comentario')
]
```

## 2.9 Migrar y Ejecutar el Servidor

### Paso 1: Aplicar las migraciones

Después de agregar los modelos y las vistas, es necesario aplicar las migraciones para crear las tablas correspondientes en la base de datos.

Ejecutamos el siguiente comando:

```
python manage.py makemigrations
python manage.py migrate
```

### Paso 2: Ejecutar el servidor

Finalmente, para ver nuestro proyecto en funcionamiento, ejecutamos el servidor de desarrollo de Django.

Ejecutamos el siguiente comando:

```
python manage.py runserver
```

Con esto, podemos acceder a nuestro sistema CRUD de publicaciones y comentarios en el navegador, utilizando las URLs configuradas en las vistas y templates.

Por ejemplo,

1. Para ver la lista de publicaciones, accedemos a “/lista/”
2. Para crear una nueva publicación, accedemos a “/crear/”.
3. Para ver la lista de comentarios, accedemos a “/lista\_comentarios/”
4. Para crear un nuevo comentario, accedemos a “/crear\_comentario/”.

## 3 Correcciones

### 3.1 Integración de Botones.

Para integrar los botones de actualizar y eliminar en la lista de publicaciones y comentarios, necesitamos realizar algunos cambios en las plantillas y en las vistas.

A continuación, describo los pasos necesarios para cada uno:

### 3.2 Para las Publicaciones:

**Paso 1:** Actualizar “lista\_publicaciones.html” en el directorio “templates” de la aplicación “publicaciones”. Agregar los enlaces para actualizar y eliminar cada publicación.

```
<!-- lista_publicaciones.html -->
<!DOCTYPE html>
<html>
<head>
  <title>Lista de Publicaciones</title>
</head>
<body>
  <h1>Lista de Publicaciones</h1>
  <ul>
    {% for publicacion in publicaciones %}
    <li>
      <h2>{{ publicacion.titulo }}</h2>
      <p>{{ publicacion.contenido }}</p>
      <a href="{% url 'detalle_publicacion' pk=publicacion.pk %}">Ver detalles</a>
      <a href="{% url 'actualizar_publicacion' pk=publicacion.pk %}">Actualizar</a>
      <a href="{% url 'eliminar_publicacion' pk=publicacion.pk %}">Eliminar</a>
    </li>
  </ul>
</body>
</html>
```

```

        <ul>
            {% for comentario in publicacion.comentarios %}
            <li>{{ comentario.texto }}</li>
            <a href="{% url 'actualizar_comentario' pk=comentario.pk %}">Actualizar</a>
            <a href="{% url 'eliminar_comentario' pk=comentario.pk %}">Eliminar</a>
            {% endfor %}
        </ul>
    </li>
    {% endfor %}
</ul>
</body>
</html>

```

Paso 2: Actualizar “views.py” en la aplicación “publicaciones”. Agregar las vistas para actualizar y eliminar las publicaciones.

```

# views.py
from django.shortcuts import render, get_object_or_404, redirect
from .forms import PublicacionForm, ComentarioForm
from .models import Publicacion

from django.shortcuts import render, redirect, get_object_or_404
from .forms import PublicacionForm, ComentarioForm
from .models import Publicacion, Comentario

def lista_publicaciones(request):
    publicaciones = Publicacion.objects.all()
    return render(request, 'lista_publicaciones.html', {'publicaciones': publicaciones})

def crear_publicacion(request):
    if request.method == 'POST':
        form = PublicacionForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect('lista_publicaciones')
    else:
        form = PublicacionForm()
    return render(request, 'crear_publicacion.html', {'form': form})

def detalle_publicacion(request, pk):
    publicacion = get_object_or_404(Publicacion, pk=pk)
    comentarios = Comentario.objects.filter(publicacion=publicacion)

```

```

if request.method == 'POST':
    comentario_form = ComentarioForm(request.POST)
    if comentario_form.is_valid():
        comentario = comentario_form.save(commit=False)
        comentario.publicacion = publicacion
        comentario.save()
        return redirect('detalle_publicacion', pk=pk)
    else:
        comentario_form = ComentarioForm()

context = {
    'publicacion': publicacion,
    'comentarios': comentarios,
    'comentario_form': comentario_form,
}
return render(request, 'detalle_publicacion.html', context)

def actualizar_publicacion(request, pk):
    publicacion = get_object_or_404(Publicacion, pk=pk)
    if request.method == 'POST':
        form = PublicacionForm(request.POST, instance=publicacion)
        if form.is_valid():
            form.save()
            return redirect('lista_publicaciones')
    else:
        form = PublicacionForm(instance=publicacion)
    return render(request, 'crear_publicacion.html', {'form': form})

def eliminar_publicacion(request, pk):
    publicacion = get_object_or_404(Publicacion, pk=pk)
    if request.method == 'POST':
        publicacion.delete()
        return redirect('lista_publicaciones')
    return render(request, 'eliminar_publicacion.html', {'publicacion': publicacion})

```

**Paso 3:** Actualizar “urls.py” en la aplicación “publicaciones”. Agregar las URLs correspondientes para las vistas de actualizar y eliminar publicaciones.

```

# urls.py
from django.urls import path
from . import views

```



```
urlpatterns = [
    # Rutas de Publicaciones
    path('lista/', views.lista_publicaciones, name='lista_publicaciones'),
    path('crear/', views.crear_publicacion, name='crear_publicacion'),
    path('detalle/<int:pk>/', views.detalle_publicacion, name='detalle_publicacion'),
    path('actualizar/<int:pk>/', views.actualizar_publicacion, name='actualizar_publicacion'),
    path('eliminar/<int:pk>/', views.eliminar_publicacion, name='eliminar_publicacion'),

    # Rutas de Comentarios
    path('<int:pk>/actualizar_comentario/', views.actualizar_comentario, name='actualizar_comentario'),
    path('<int:pk>/eliminar_comentario/', views.eliminar_comentario, name='eliminar_comentario'),
]
```

### 3.3 Para los Comentarios:

El proceso es similar al de las publicaciones, solo que debemos aplicarlo para los comentarios.

**Paso 1:** Actualizar “lista\_publicaciones.html” en el directorio “templates” de la aplicación “publicaciones”. Agregar los enlaces para actualizar y eliminar cada comentario.

```
<!-- lista_publicaciones.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Lista de Publicaciones</title>
</head>
<body>
    <h1>Lista de Publicaciones</h1>
    <ul>
        {% for publicacion in publicaciones %}
        <li>
            <h2>{{ publicacion.titulo }}</h2>
            <p>{{ publicacion.contenido }}</p>
            <a href="{% url 'detalle_publicacion' pk=publicacion.pk %}">Ver detalles</a>
            <a href="{% url 'actualizar_publicacion' pk=publicacion.pk %}">Actualizar</a>
            <a href="{% url 'eliminar_publicacion' pk=publicacion.pk %}">Eliminar</a>
        </li>
        {% for comentario in publicacion.comentarios %}
        <li>{{ comentario.texto }}</li>
        <a href="{% url 'actualizar_comentario' pk=comentario.pk %}">Actualizar</a>
        </li>
        </ul>
    </ul>
</body>
</html>
```

```

        <a href="{% url 'eliminar_comentario' pk=comentario.pk %}">Eliminar</a>
    {% endfor %}
</ul>
</li>
{% endfor %}
</ul>
</body>
</html>

```

También es necesario la creación de un nuevo template llamado `detalle_publicacion.html` en el directorio “templates” de la aplicación “publicaciones”. Agregar los enlaces para actualizar y eliminar cada comentario.

```

<!-- detalle_publicacion.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Detalle de Publicación</title>
</head>
<body>
    <h1>{{ publicacion.titulo }}</h1>
    <p>{{ publicacion.contenido }}</p>
    <h3>Comentarios:</h3>
    <ul>
        {% for comentario in comentarios %}
        <li>{{ comentario.contenido }}</li>
        <a href="{% url 'actualizar_comentario' comentario.pk %}">Actualizar Comentario</a>
        <a href="{% url 'eliminar_comentario' comentario.pk %}">Eliminar Comentario</a>
        {% endfor %}
    </ul>

    <a href="{% url 'actualizar_publicacion' publicacion.pk %}">Actualizar Publicación</a>
    <a href="{% url 'eliminar_publicacion' publicacion.pk %}">Eliminar Publicación</a>

    <!-- Agregar formulario para agregar comentario -->
    <h3>Agregar Comentario:</h3>
    <form method="post">
        {% csrf_token %}
        {{ comentario_form.as_p }}
        <button type="submit">Enviar Comentario</button>
    </form>

```

```
</body>
</html>
```

Y finalmente actualizar el archivo `actualizar_comentario.html` en el directorio “templates” de la aplicación “publicaciones”. Agregar los enlaces para actualizar y eliminar cada comentario.

```
<!-- actualizar_comentario.html -->
<!DOCTYPE html>
<head>
    <title>Actualizar Comentario</title>
</head>
<body>
    <h1>Actualizar Comentario</h1>
    <form method="post">
        {% csrf_token %}
        {{ comentario_form.as_p }}
        <button type="submit">Guardar Cambios</button>
    </form>
</body>
</html>
```

**Paso 2:** Actualizar “views.py” en la aplicación “publicaciones”. Agregar las vistas para actualizar y eliminar los comentarios.

```
# views.py
from django.shortcuts import render, get_object_or_404, redirect
from .forms import PublicacionForm, ComentarioForm
from .models import Publicacion, Comentario

def lista_publicaciones(request):
    publicaciones = Publicacion.objects.all()
    return render(request, 'lista_publicaciones.html', {'publicaciones': publicaciones})

def crear_publicacion(request):
    # Código existente

def detalle_publicacion(request, pk):
    # Código existente

def actualizar_publicacion(request, pk):
    # Código existente
```

```

def eliminar_publicacion(request, pk):
    # Código existente

def crear_comentario(request):
    if request.method == 'POST':
        form = ComentarioForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect('lista_publicaciones')
    else:
        form = ComentarioForm()
    return render(request, 'crear_comentario.html', {'form': form})

def lista_comentarios(request):
    comentarios = Comentario.objects.all()
    return render(request, 'lista_comentarios.html', {'comentarios': comentarios})

def actualizar_comentario(request, pk):
    comentario = get_object_or_404(Comentario, pk=pk)

    if request.method == 'POST':
        comentario_form = ComentarioForm(request.POST, instance=comentario)
        if comentario_form.is_valid():
            comentario_form.save()
            return redirect('detalle_publicacion', pk=comentario.publicacion.pk)
    else:
        comentario_form = ComentarioForm(instance=comentario)

    context = {
        'comentario_form': comentario_form,
    }
    return render(request, 'actualizar_comentario.html', context)

def eliminar_comentario(request, pk):
    comentario = get_object_or_404(Comentario, pk=pk)
    publicacion_pk = comentario.publicacion.pk
    comentario.delete()
    return redirect('detalle_publicacion', pk=publicacion_pk)

```

**Paso 3:** Actualizar “urls.py” en la aplicación “publicaciones”. Agregar las URLs correspondientes para las vistas de actualizar y eliminar comentarios.

```
# urls.py
from django.urls import path
from . import views

urlpatterns = [
    # Rutas de Comentarios
    path('<int:pk>/actualizar_comentario/', views.actualizar_comentario, name='actualizar_
    path('<int:pk>/eliminar_comentario/', views.eliminar_comentario, name='eliminar_coment
]
```

### 3.4 Corregir el Modelo

Finalmente para corregir el modelo y hacer que el campo autor herede los datos de **User** de Django, podemos usar un campo **ForeignKey** que apunte al modelo User.

Esto nos permitirá asociar cada publicación y comentario a un usuario específico. Aquí está el código corregido:

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

class Publicacion(models.Model):
    titulo = models.CharField(max_length=200)
    contenido = models.TextField()
    fecha_publicacion = models.DateTimeField(auto_now_add=True)
    autor = models.ForeignKey(User, on_delete=models.CASCADE)

    def __str__(self):
        return self.titulo

class Comentario(models.Model):
    publicacion = models.ForeignKey(Publicacion, on_delete=models.CASCADE)
    autor = models.ForeignKey(User, on_delete=models.CASCADE)
    contenido = models.TextField()
    fecha_publicacion = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return f"Comentario de {self.autor.username} en {self.publicacion.titulo}"
```

En este código, hemos modificado el campo autor en ambos modelos para que sea un **ForeignKey** que apunta al modelo **User** de Django.

Con esto, cada publicación y comentario estará asociado a un usuario registrado en el sistema.

El argumento `on_delete=models.CASCADE` en el campo autor de Comentario asegura que si un usuario es eliminado, también se eliminarán todos sus comentarios relacionados, pero ten en cuenta que esto es opcional y depende de la lógica de negocio que desees implementar.

### 3.5 Backend: Construcción de una API con Django Rest Framework

En esta sección, dividiremos todo lo que hemos realizado hasta ahora en una API utilizando Django Rest Framework (DRF). Esto nos permitirá exponer nuestros modelos (Publicacion y Comentario) como puntos finales (endpoints) para realizar operaciones CRUD (Crear, Leer, Actualizar y Eliminar) a través de peticiones HTTP.

#### Paso 1: Instalar Django Rest Framework

Primero, debemos instalar Django Rest Framework en nuestro entorno virtual. Ejecuta el siguiente comando:

```
pip install djangorestframework
```

#### Paso 2: Configurar Django Rest Framework en el Proyecto

En el archivo `settings.py` del proyecto, agrega `'rest_framework'` a la lista de aplicaciones instaladas:

```
INSTALLED_APPS = [  
    # Otras aplicaciones...  
    'rest_framework',  
]
```

#### Paso 3: Serializadores

En DRF, los serializadores se utilizan para convertir nuestros modelos de Django en formatos JSON y viceversa. Vamos a crear los serializadores para los modelos Publicacion y Comentario en un archivo `serializers.py` dentro de la aplicación publicaciones.

```
# serializers.py  
from rest_framework import serializers  
from .models import Publicacion, Comentario  
  
class ComentarioSerializer(serializers.ModelSerializer):  
    class Meta:  
        model = Comentario
```

```

        fields = '__all__'

class PublicacionSerializer(serializers.ModelSerializer):
    comentarios = ComentarioSerializer(many=True, read_only=True)

    class Meta:
        model = Publicacion
        fields = '__all__'

```

En este código, creamos dos serializadores, `ComentarioSerializer` y `PublicacionSerializer`, que utilizan el modelo correspondiente y definen los campos que queremos exponer en nuestra API. En el caso de la publicación, utilizamos `comentarios` para mostrar los comentarios relacionados.

#### Paso 4: Vistas

Vamos a modificar nuestras vistas para utilizar los serializadores y convertir nuestros datos en formato JSON. En el archivo `views.py` de la aplicación `publicaciones`, actualiza el contenido de las vistas de la siguiente manera:

```

# views.py
from rest_framework import generics
from .models import Publicacion, Comentario
from .serializers import PublicacionSerializer, ComentarioSerializer

class ListaPublicaciones(generics.ListCreateAPIView):
    queryset = Publicacion.objects.all()
    serializer_class = PublicacionSerializer

class DetallePublicacion(generics.RetrieveUpdateDestroyAPIView):
    queryset = Publicacion.objects.all()
    serializer_class = PublicacionSerializer

class ListaComentarios(generics.ListCreateAPIView):
    queryset = Comentario.objects.all()
    serializer_class = ComentarioSerializer

class DetalleComentario(generics.RetrieveUpdateDestroyAPIView):
    queryset = Comentario.objects.all()
    serializer_class = ComentarioSerializer

```

#### Paso 5: URLs

Ahora, vamos a configurar las URLs de nuestra API en el archivo `urls.py` de la aplicación publicaciones.

```
# urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('publicaciones/', views.ListaPublicaciones.as_view(), name='lista_publicaciones'),
    path('publicaciones/<int:pk>/', views.DetallePublicacion.as_view(), name='detalle_publicacion'),
    path('comentarios/', views.ListaComentarios.as_view(), name='lista_comentarios'),
    path('comentarios/<int:pk>/', views.DetalleComentario.as_view(), name='detalle_comentario'),
]
```

En este código, configuramos las URLs de nuestras vistas utilizando las vistas basadas en clases proporcionadas por DRF. Creamos puntos finales (endpoints) para listar, crear, ver, actualizar y eliminar publicaciones y comentarios.

¡Hemos construido una API básica para nuestro proyecto “Blog” utilizando Django Rest Framework!

Ahora podemos realizar operaciones CRUD a través de las peticiones HTTP en nuestros modelos de Publicacion y Comentario.

### 3.6 Documentación de las API

Para documentar las API, podemos utilizar la herramienta **drf-yasg**. La información la vamos a obtener de la documentación oficial en el siguiente link

[Documentación de drf-yasg](#)

**Paso 1:** Instalar drf-yasg

Primero, debemos instalar drf-yasg en nuestro entorno virtual. Ejecuta el siguiente comando:

```
pip install -U drf-yasg
```

**Paso 2:** Configurar drf-yasg en el Proyecto

En el archivo `settings.py` del proyecto, agrega ‘`drf_yasg`’ a la lista de aplicaciones instaladas:

```
INSTALLED_APPS = [
    # Otras aplicaciones...
    'drf_yasg',
]
```



```
]
```

### Paso 3: Configurar drf-yasg en el archivo urls.py

En el archivo urls.py del proyecto, agrega las siguientes líneas de código:

```
# urls.py
...
from django.urls import re_path
from rest_framework import permissions
from drf_yasg.views import get_schema_view
from drf_yasg import openapi

...

schema_view = get_schema_view(
    openapi.Info(
        title="Snippets API",
        default_version='v1',
        description="Test description",
        terms_of_service="https://www.google.com/policies/terms/",
        contact=openapi.Contact(email="contact@snippets.local"),
        license=openapi.License(name="BSD License"),
    ),
    public=True,
    permission_classes=(permissions.AllowAny,),
)

urlpatterns = [
    path('swagger<format>/', schema_view.without_ui(cache_timeout=0), name='schema-json'),
    path('swagger/', schema_view.with_ui('swagger', cache_timeout=0), name='schema-swagger'),
    path('redoc/', schema_view.with_ui('redoc', cache_timeout=0), name='schema-redoc'),
    ...
]
```

### Paso 4: Agregar la variable authentication\_classes.

En el archivo url.py del proyecto agregar la variable authentication\_classes para que no nos pida autenticación para poder ver la documentación de las API.

```
# urls.py
...
authentication_classes = []
```

...

### Paso 5: Ejecutar el servidor

Ejecuta el servidor y en el navegador ingresa a <http://localhost:8000/swagger> o <http://localhost:8000/redoc>

```
python manage.py runserver
```

También puedes probar los plugins de VSCode RappidApi Client o Thunder Client para poder ver la documentación de las API desde el editor de código.

En la siguiente sección, construiremos el frontend para consumir esta API.

## 4 FRONTEND: Herencia de Plantillas con Django.

### 4.1 Herencia de Plantillas

Para iniciar con el tema de herencia de plantillas de django, es necesario generar un archivo `base.html` que será nuestra plantilla base para la generación de las demás plantillas.

En esta plantilla base incluiremos una barra de navegación con accesos a archivos estáticos de **Home**, **Contacto**, **Quienes somos** y **Crear Publicación**.

- Abre el archivo `base.html` en el directorio `templates` de tu proyecto.
- Agrega el siguiente código para definir la barra de navegación:

```
<!DOCTYPE html>
<html>
<head>
    <title>{% block title %}Mi Blog{% endblock %}</title>
    <!-- Agregar enlaces a Bootstrap CSS -->
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/css/boots
</head>
<body>
    <nav class="navbar navbar-expand navbar-dark bg-dark">
        <div class="container">
            <a class="navbar-brand" href="{% url 'home' %}">Mi Blog</a>
            <ul class="navbar-nav ml-auto">
                <li class="nav-item">
                    <a class="nav-link" href="{% url 'home' %}">Home</a>
                </li>
                <li class="nav-item">
```

```

        <a class="nav-link" href="{% url 'home' %}">Creditos</a>
    </li>
    <li class="nav-item">
        <a class="nav-link" href="{% url 'contacto' %}">Contacto</a>
    </li>
    <li class="nav-item">
        <a class="nav-link" href="{% url 'quienes_somos' %}">Quienes Somos</a>
    </li>
    <li class="nav-item">
        <a class="nav-link" href="{% url 'crear_publicacion' %}">Crear</a>
    </li>
</ul>
</div>
</nav>

<div class="container mt-4">
    {% block content %}
    {% endblock %}
</div>

<!-- Agregar enlaces a Bootstrap JS y jQuery -->
<script src="https://code.jquery.com/jquery-3.5.1.slim.min.js"></script>
<script src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.9.1/dist/umd/popper.min.js">
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"></script>
</body>
</html>

```

En este código, hemos definido una etiqueta

que contiene enlaces para “Home”, “Contacto”, “Quienes Somos” y “Crear”. Los enlaces utilizan las etiquetas `{% url %}` de Django para generar las URLs correspondientes a cada vista.

- En cada plantilla que extienda de `base.html`, asegúrate de agregar bloques de contenido para el título y el contenido específico de cada página:

Por ejemplo, en la plantilla `lista_publicaciones.html`:

```

{% extends 'base.html' %}

{% block title %}Lista de Publicaciones{% endblock %}

{% block content %}

```

```

<h1>Lista de Publicaciones</h1>
<!-- Contenido de la lista de publicaciones -->
{% endblock %}

```

En la plantilla detalle\_publicacion.html:

```

{% extends 'base.html' %}

{% block title %}Detalle de Publicación{% endblock %}

{% block content %}
<h1>Detalle de Publicación</h1>
<!-- Contenido del detalle de la publicación -->
{% endblock %}

```

Y así sucesivamente para las demás plantillas que extiendan de base.html.

## 4.2 Modificación de Views.

Vista y URL para la página “Home”:

```

# views.py
from django.shortcuts import render

def home_view(request):
    return render(request, 'home.html')

# urls.py del proyecto
from django.urls import path
from . import views

urlpatterns = [
    path('home/', views.home_view, name='home'),
    # otras URLs de tu proyecto
]

```

- Vista y URL para la página “Contacto”:

```

# views.py
from django.shortcuts import render

```

```

def contacto_view(request):
    return render(request, 'contacto.html')

# urls.py del proyecto
from django.urls import path
from . import views

urlpatterns = [
    # otras URLs de tu proyecto
    path('contacto/', views.contacto_view, name='contacto'),
]

```

Vista y URL para la página “Quienes Somos”:

```

# views.py
from django.shortcuts import render

def quienes_somos_view(request):
    return render(request, 'quienes_somos.html')

# urls.py del proyecto
from django.urls import path
from . import views

urlpatterns = [
    # otras URLs de tu proyecto
    path('quienes_somos/', views.quienes_somos_view, name='quienes_somos'),
]

```

- Vista y URL para la página “Crear”:

```

# views.py
from django.shortcuts import render

def crear_view(request):
    return render(request, 'crear.html')

# urls.py del proyecto
from django.urls import path
from . import views

urlpatterns = [

```

```

    # otras URLs de tu proyecto
    path('crear/', views.crear_view, name='crear_publicacion'),
]

```

### 4.3 Bootstrap en base.html

Para incluir Bootstrap en el archivo base.html, debes agregar las etiquetas y

```

<script src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.0.7/dist/umd/popper.min.js"></script>
<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"></script>

```

Este código incluye las hojas de estilos y scripts de Bootstrap desde sus respectivos CDNs (

Si prefieres tener los archivos locales en tu proyecto, simplemente descárgalos desde la página

Con esta modificación, tu archivo base.html utilizará Bootstrap y tendrás una barra de naveg

## Inclusión de Herencia de Plantillas en las demás plantillas.

1. Template de home.html.

```

<<< html
{% extends 'base.html' %}

{% block title %}Home - Mi Blog{% endblock %}

{% block content %}
<h1>Bienvenido a Mi Blog</h1>
<p>Este es el contenido de la página de inicio.</p>
{% endblock %}

```

2. Template de contacto.html.

```

{% extends 'base.html' %}

{% block title %}Contacto - Mi Blog{% endblock %}

{% block content %}
<h1>Contacto</h1>
<p>Información de contacto y formulario de contacto.</p>
{% endblock %}

```

### 3. Template de quienes\_somos.html.

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

{% block title %}Quienes Somos - Mi Blog{% endblock %}

{% block content %}
<h1>Quienes Somos</h1>
<p>Información sobre nosotros y nuestra historia.</p>
{% endblock %}
```

En cada uno de estos templates, utilizamos la etiqueta `{% extends 'base.html' %}` para indicar que estamos heredando el contenido y la estructura del archivo `base.html`. Luego, en el bloque `{% block content %}`, agregamos el contenido específico de cada página.

Cada template tiene su título personalizado en el bloque `{% block title %}`.

Cuando se acceda a las URLs correspondientes a cada template, se mostrará el contenido específico de cada página dentro del contenedor proporcionado por el archivo `base.html`, y la barra de navegación con los enlaces a Home, Contacto, Quienes Somos y Crear se mantendrá visible en todas las páginas gracias a la herencia.

## 4.4 Agregando bootstrap a los templates.

Para agregar Bootstrap a las plantillas anteriores, necesitaremos incluir los enlaces a los archivos CSS y JS de Bootstrap en el archivo `base.html`. Además, podemos utilizar las clases de Bootstrap para mejorar la apariencia de los elementos en cada página.

A continuación, se muestran las plantillas actualizadas con Bootstrap:

### 1. Template de base.html

```
<!DOCTYPE html>
<html>
<head>
  <title>{% block title %}Mi Blog{% endblock %}</title>
  <!-- Agregar enlaces a Bootstrap CSS -->
  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/css/boots
</head>
<body>
  <nav class="navbar navbar-expand navbar-dark bg-dark">
    <div class="container">
      <a class="navbar-brand" href="{% url 'home' %}">Mi Blog</a>
```

```

        <ul class="navbar-nav ml-auto">
            <li class="nav-item">
                <a class="nav-link" href="{% url 'home' %}">Home</a>
            </li>
            <li class="nav-item">
                <a class="nav-link" href="{% url 'contacto' %}">Contacto</a>
            </li>
            <li class="nav-item">
                <a class="nav-link" href="{% url 'quienes_somos' %}">Quienes Somos</a>
            </li>
            <li class="nav-item">
                <a class="nav-link" href="{% url 'crear_publicacion' %}">Crear</a>
            </li>
        </ul>
    </div>
</nav>

<div class="container mt-4">
    {% block content %}
    {% endblock %}
</div>

<!-- Agregar enlaces a Bootstrap JS y jQuery -->
<script src="https://code.jquery.com/jquery-3.5.1.slim.min.js"></script>
<script src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.9.1/dist/umd/popper.min.js">
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"></script>
</body>
</html>

```

2. Template de home.html.

```

{% extends 'base.html' %}

{% block title %}Home - Mi Blog{% endblock %}

{% block content %}
<div class="jumbotron">
    <h1 class="display-4">Bienvenido a Mi Blog</h1>
    <p class="lead">Este es el contenido de la página de inicio.</p>
</div>
{% endblock %}

```



### 3. Template de contacto.html.

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

{% block title %}Contacto - Mi Blog{% endblock %}

{% block content %}
<div class="row">
  <div class="col-md-6 mx-auto">
    <h1>Contacto</h1>
    <p>Información de contacto y formulario de contacto.</p>
  </div>
</div>
{% endblock %}
```

### 4. Template de quienes\_somos.html

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

{% block title %}Quienes Somos - Mi Blog{% endblock %}

{% block content %}
<div class="container">
  <h1>Quienes Somos</h1>
  <p>Información sobre nosotros y nuestra historia.</p>
</div>
{% endblock %}
```

En estas plantillas, hemos utilizado las clases de Bootstrap para dar estilo a los elementos. Por ejemplo, en el archivo base.html, hemos utilizado la clase “navbar” para crear la barra de navegación y la clase “jumbotron” para resaltar el contenido de la página de inicio. Además, en los templates de contacto.html y quienes\_somos.html, hemos utilizado la clase “container” para centrar el contenido y la clase “row” para crear una fila que contenga el contenido.

## 5 Reactjs

### 5.1 Frontend con Reactjs

Para crear un proyecto desde cero en ReactJS que consuma los datos de la API creada con Django Rest Framework, sigue estos pasos:

**Paso 1:** Verificar versiones de Node.js y npm.

Antes de comenzar, asegúrate de tener Node.js y npm instalados en tu sistema. Abre la terminal y ejecuta los siguientes comandos para verificar las versiones:

```
node -v
npm -v
```

### Paso 2: Crear un nuevo proyecto de React

Crea un nuevo proyecto de React utilizando create-react-app. Ejecuta el siguiente comando en la terminal:

```
npx create-react-app frontend
```

Esto creará una nueva carpeta llamada “frontend” con una estructura de proyecto de React preconfigurada.

### Paso 3: Instalar axios

En la carpeta “frontend”, instala la librería axios para realizar peticiones HTTP a la API de Django. Ejecuta el siguiente comando en la terminal:

```
cd frontend
npm install axios
```

### Paso 4: Crear el componente ListaPublicaciones.js

Crea el componente **ListaPublicaciones.js** en la carpeta “src/components” con el siguiente contenido:

```
import React, { useState, useEffect } from 'react';
import axios from 'axios';

const ListaPublicaciones = () => {
  const [publicaciones, setPublicaciones] = useState([]);

  useEffect(() => {
    const fetchPublicaciones = async () => {
      try {
        const response = await axios.get('http://localhost:8000/publicaciones/');
        setPublicaciones(response.data.results);
      } catch (error) {
        console.error('Error al obtener las publicaciones:', error);
      }
    };
  });
};
```

```

    fetchPublicaciones();
  }, []);

  return (
    <div>
      <h1>Lista de Publicaciones</h1>
      <ul>
        {publicaciones.map((publicacion) => (
          <li key={publicacion.id}>
            <h2>{publicacion.titulo}</h2>
            <p>{publicacion.contenido}</p>
          </li>
        ))}
      </ul>
    </div>
  );
};

export default ListaPublicaciones;

```

### Paso 5: Crear el componente CrearPublicacion.js

Crea el componente CrearPublicacion.js en la carpeta “src/components” con el siguiente contenido:

```

import React from 'react';
import axios from 'axios';

const CrearPublicacion = () => {
  const [titulo, setTitulo] = React.useState('');
  const [contenido, setContenido] = React.useState('');

  // Replace "1" with the ID of the author you want to associate with the publicación
  const autorId = 1;

  const handleCrear = async () => {
    try {
      await axios.post('http://localhost:8000/publicaciones/', {
        titulo,
        contenido,
        autor: autorId,
      });
    }
  };

```

```

        alert('Publicación creada correctamente.');
```

```

        setTitulo('');
        setContenido('');
    } catch (error) {
        console.error('Error al crear la publicación:', error);
    }
};

return (
    <div>
        <h1>Crear Publicación</h1>
        <label>
            Título:
            <input type="text" value={titulo} onChange={(e) => setTitulo(e.target.value)} />
        </label>
        <br />
        <label>
            Contenido:
            <textarea value={contenido} onChange={(e) => setContenido(e.target.value)} />
        </label>
        <br />
        <button onClick={handleCrear}>Crear</button>
    </div>
);
};

export default CrearPublicacion;
```

### Paso 6: Crear el componente ActualizarPublicacion.js

Crea el componente ActualizarPublicacion.js en la carpeta “src/components” con el siguiente contenido:

```

import React, { useState, useEffect } from 'react';
import axios from 'axios';

const ActualizarPublicacion = () => {
    const [titulo, setTitulo] = useState('');
    const [contenido, setContenido] = useState('');
    const [autor, setAutor] = useState('');

    // Replace "3" with the ID of the publicación you want to update
```

```

const publicacionId = 3;

useEffect(() => {
  const fetchPublicacion = async () => {
    try {
      const response = await axios.get(`http://localhost:8000/publicaciones/${publicacionId}`);
      const { titulo, contenido, autor } = response.data;
      setTitulo(titulo);
      setContenido(contenido);
      setAutor(autor);
    } catch (error) {
      console.error('Error al obtener la publicación:', error);
    }
  };

  fetchPublicacion();
}, [publicacionId]);

const handleActualizar = async () => {
  try {
    await axios.put(`http://localhost:8000/publicaciones/${publicacionId}/`, {
      titulo,
      contenido,
      autor,
    });
    alert('Publicación actualizada correctamente.');
```

```

  } catch (error) {
    console.error('Error al actualizar la publicación:', error);
  }
};

return (
  <div>
    <h1>Actualizar Publicación</h1>
    <label>
      Título:
      <input type="text" value={titulo} onChange={(e) => setTitulo(e.target.value)} />
    </label>
    <br />
    <label>
      Contenido:

```

```

        <textarea value={contenido} onChange={(e) => setContenido(e.target.value)} />
      </label>
      <br />
      <label>
        Autor:
        <input type="number" value={autor} onChange={(e) => setAutor(Number(e.target.value))} />
      </label>
      <br />
      <button onClick={handleActualizar}>Actualizar</button>
    </div>
  );
};

export default ActualizarPublicacion;

```

### Paso 7: Crear el componente EliminarPublicacion.js

Crea el componente EliminarPublicacion.js en la carpeta “src/components” con el siguiente contenido:

```

import React, { useEffect } from 'react';
import axios from 'axios';

const EliminarPublicacion = () => {
  // Replace "2" with the ID of the publicación you want to delete
  const publicacionId = 2;

  useEffect(() => {
    const fetchPublicacion = async () => {
      try {
        const response = await axios.get(`http://localhost:8000/publicaciones/${publicacionId}`);
        const { titulo, contenido, autor } = response.data;
        console.log('Publicación a eliminar:', { titulo, contenido, autor });
      } catch (error) {
        console.error('Error al obtener la publicación:', error);
      }
    };

    fetchPublicacion();
  }, [publicacionId]);

  const handleEliminar = async () => {

```

```

    try {
      await axios.delete(`http://localhost:8000/publicaciones/${publicacionId}/`);
      alert('Publicación eliminada correctamente.');
```

```

    } catch (error) {
      console.error('Error al eliminar la publicación:', error);
    }
  };

  return (
    <div>
      <h1>Eliminar Publicación</h1>
      <p>¿Estás seguro de que deseas eliminar esta publicación?</p>
      <button onClick={handleEliminar}>Eliminar</button>
    </div>
  );
};

export default EliminarPublicacion;
```

#### Paso 8: Crear el componente Home.js

Crea el componente Home.js en la carpeta “src/components” con el siguiente contenido:

```

import React from 'react';

const Home = () => {
  return (
    <div>
      <h1>Bienvenido a la página de inicio</h1>
    </div>
  );
};

export default Home;
```

#### Paso 9: Crear el componente NavBar.js

Crea el componente NavBar.js en la carpeta “src/components” con el siguiente contenido:

```

import React from 'react';
import { Link } from 'react-router-dom';

const NavBar = () => {
```

```

return (
  <nav className="navbar navbar-expand navbar-dark bg-dark">
    <div className="container">
      <Link className="navbar-brand" to="/">Mi Blog</Link>
      <ul className="navbar-nav ml-auto">
        <li className="nav-item">
          <Link className="nav-link" to="/">Home</Link>
        </li>
        <li className="nav-item">
          <Link className="nav-link" to="/crear">Crear</Link>
        </li>
      </ul>
    </div>
  </nav>
);
};

export default NavBar;

```

### Paso 10: Actualizar el archivo App.js

Actualiza el archivo App.js en la carpeta “src” con el siguiente contenido:

```

import React from 'react';
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';
import NavBar from './components/NavBar';
import ListaPublicaciones from './components/ListaPublicaciones';
import CrearPublicacion from './components/CrearPublicacion';
import DetallePublicacion from './components/DetallePublicacion';
import ActualizarPublicacion from './components/ActualizarPublicacion';
import EliminarPublicacion from './components/EliminarPublicacion';

const App = () => {
  return (
    <Router>
      <div>
        <NavBar />
        <Routes>
          <Route path="/" element={<ListaPublicaciones />} />
          <Route path="/crear" element={<CrearPublicacion />} />
          <Route path="/publicaciones/:id" element={<DetallePublicacion />} />
          <Route path="/publicaciones/:id/actualizar" element={<ActualizarPublicacion />} />

```



```

        <Route path="/publicaciones/:id/eliminar" element={<EliminarPublicacion />} />
      </Routes>
    </div>
  </Router>
);
};

export default App;

```

### Paso 11: Estilos CSS con los archivos index.css y App.css

En una aplicación React, los estilos CSS se aplican a los componentes para mejorar la apariencia y el diseño de la interfaz de usuario. Por defecto, React crea dos archivos para estilos: index.css y App.css.

**index.css:** Este archivo contiene estilos globales que se aplican a toda la aplicación.

**App.css:** Este archivo contiene estilos específicos para el componente principal de la aplicación (en este caso, el componente App.js).

A continuación, se muestra cómo puedes usar estos archivos CSS en tu aplicación React:

#### 5.1.1 1. index.css

En el directorio src, crea un archivo llamado index.css y agrega los estilos globales que deseas aplicar a toda la aplicación. Por ejemplo:

```

/* index.css */
body {
  font-family: Arial, sans-serif;
  margin: 0;
  padding: 0;
}

.container {
  max-width: 1200px;
  margin: 0 auto;
}

```

#### 5.1.2 2. App.css

En el directorio src, ya deberías tener un archivo llamado App.css. Puedes agregar estilos específicos para el componente App.js aquí. Por ejemplo:

```

/* App.css */
.header {
  background-color: #007bff;
  color: #ffffff;
  padding: 1rem;
  text-align: center;
}

.nav-link {
  color: #ffffff;
  text-decoration: none;
  margin-right: 1rem;
}

.nav-link:hover {
  color: #0056b3;
}

```

### 5.1.3 3. Importar los archivos CSS en App.js

En el componente App.js, importa los archivos CSS para que los estilos se apliquen a la aplicación. Debes importar index.css al principio del archivo src/index.js, y App.css en App.js:

```

// src/index.js
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
import './index.css'; // Importar index.css aquí

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);

// src/App.js
import React from 'react';
import './App.css'; // Importar App.css aquí

```

```
function App() {
  // Código de la aplicación
  return (
    <div className="container">
      <header className="header">
        <nav>
          <a className="nav-link" href="/">Home</a>
          <a className="nav-link" href="/contacto">Contacto</a>
          <a className="nav-link" href="/quienes_somos">Quienes Somos</a>
          <a className="nav-link" href="/crear_publicacion">Crear</a>
        </nav>
      </header>
      {/* Resto del contenido de la aplicación */}
    </div>
  );
}

export default App;
```

Con esto, tus estilos CSS definidos en `index.css` se aplicarán a toda la aplicación, y los estilos definidos en `App.css` se aplicarán específicamente al componente `App.js`. Recuerda adaptar los estilos a tus necesidades y preferencias.

## Paso 12: Importar estilos CSS

En el archivo **index.js** en la carpeta “**src**”, importa el archivo **styles.css** para aplicar los estilos personalizados:

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);

reportWebVitals();
```

Con esto, has creado un proyecto de React independiente que consume los datos de la API creada con Django Rest Framework. Puedes ejecutar el proyecto de React con el siguiente comando:

```
npm start
```

Luego, podrás ver la aplicación en el navegador accediendo a <http://localhost:3000/>.

Los componentes ListaPublicaciones, CrearPublicacion, ActualizarPublicacion y EliminarPublicacion estarán disponibles en sus respectivas rutas y consumirán datos de la API de Django.

## 5.2 Inconvenientes

En caso de tener inconvenientes con las peticiones por parte de la aplicación de React, se debe agregar la siguiente línea en el archivo settings.py de la carpeta “backend”:

```
CORS_ORIGIN_ALLOW_ALL = True
```