AWAKELAB

Desarrollo Web Python con Django

Desarrollador de aplicaciones Full Stack

Python Trainee

6.3.- Contenido 3: Implementar una aplicación web Django utilizando templates para el despliegue de páginas con contenido dinámico que dan solución a un requerimiento.

Objetivo de la jornada

1. Implementa un proyecto Django para servir contenido estático dando solución a los requerimientos.
2. Utiliza templates para la renderización de contenido dinámico en un proyecto Django para dar solución a un requerimiento.
3. Utiliza herencia de plantillas en un proyecto Django para dar solución a un requerimiento.
4. Utiliza instrucciones de control en plantillas de un proyecto Django para dar solución a un requerimiento.

6.3.1.- Renderización dinámica y templates

Para continuar vamos a reorganizar un poco nuestro proyecto, hasta ahora solo hemos creado un proyecto no una aplicación, que es lo que haremos ahora:

```
(env) .../sitio$ python manage.py startapp app
```

Y cambiaremos el siguiente código desde **sitiodeprueba/views.py** a **sitio/app/views.py**

```
from django.http import HttpResponse
from django.template import loader

def index(request):
    template = loader.get_template('app/index.html')
    return HttpResponse(template.render({}, request))
```

También necesitamos mover nuestro archivo parte del contenido de **sitio/sitiodeprueba/urls.py** a un archivo nuevo que necesitamos crear **sitio/app/urls.py**

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

urlpatterns = [
```

```
    path('', views.index, name='index'),
]
```

También necesitamos decirle a Django que tenemos una nueva application, esto hacemos en **sitio/sitiodeprueba/settings.py**

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

La ruta **admin** la dejamos en el archivo original, pero también necesitamos agregar una referencia a **sitio/app/urls.py**

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

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

Como hicimos un cambio hacia nuestra aplicación (app) ahora ya no tenemos nada en la raíz del proyecto, sino que lo movimos hacia la URL correspondiente a nuestra **app** http://127.0.0.1:8000/app/

No se olvide que un proyecto Django se puede componer de múltiples aplicaciones bajo el techo común que es el proyecto.

Esta es nuestra nueva estructura de directorios y archivos:

```
└── sitio
    ├── app
    │   ├── __init__.py
    │   ├── admin.py
    │   ├── apps.py
    │   ├── migrations
    │   │   └── __init__.py
    │   ├── models.py
    │   ├── templates
    │   │   └── app
    │   │       └── index.html
    │   ├── tests.py
    │   ├── urls.py
    │   └── views.py
    ├── db.sqlite3
    ├── manage.py
    └── sitiodeprueba
        ├── __init__.py
        ├── asgi.py
        ├── settings.py
        ├── urls.py
        ├── views.py
        └── wsgi.py
```

6.3.1.1. Cómo usar páginas parciales

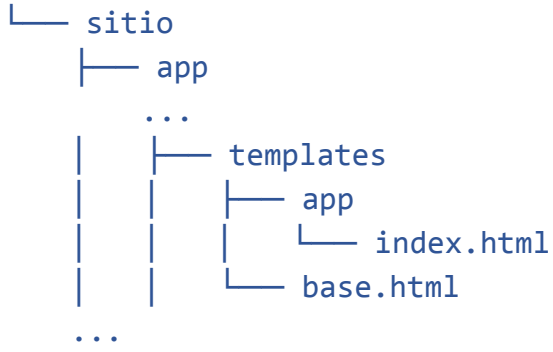
Vamos a re-estructurar nuestra página para aprovechar las capacidades de Django en relación a templates. Para estos propósitos vamos a refactorizar nuestra template **index.html**.

Primero creamos **base.html** (dentro de **sitio/app/templates**)

```
<!DOCTYPE html>
<html lang="es">
<head>
    <title>{% block title %}{% endblock %}</title>
    <meta charset="utf-8"/>
    <style>
        #imagen {
            float: left;
            width: 60%;
            text-align: center;
        }
        #formulario-beneficios {
            float: right;
            width: 40%;
        }
        header, footer {
            clear: both;
            text-align: center;
        } </style>
</head>
<body>
<header>
    {% block header %}{% endblock %}
</header>
<section id="imagen">
    {% block imagen %}{% endblock %}
</section>
<section id="formulario-beneficios">
    {% block formulario %}{% endblock %}
</section>
<footer>
    {% block footer %}{% endblock %}
</footer>
</body>
</html>
```

La plantilla **base.html** es mucho más pequeña que la anterior (index.html), y la diferencia es que contiene "bloques" (**blocks**) que serán "llenados posteriormente".

Nuestra nueva estructura de archivos relevante para las plantillas es ahora:

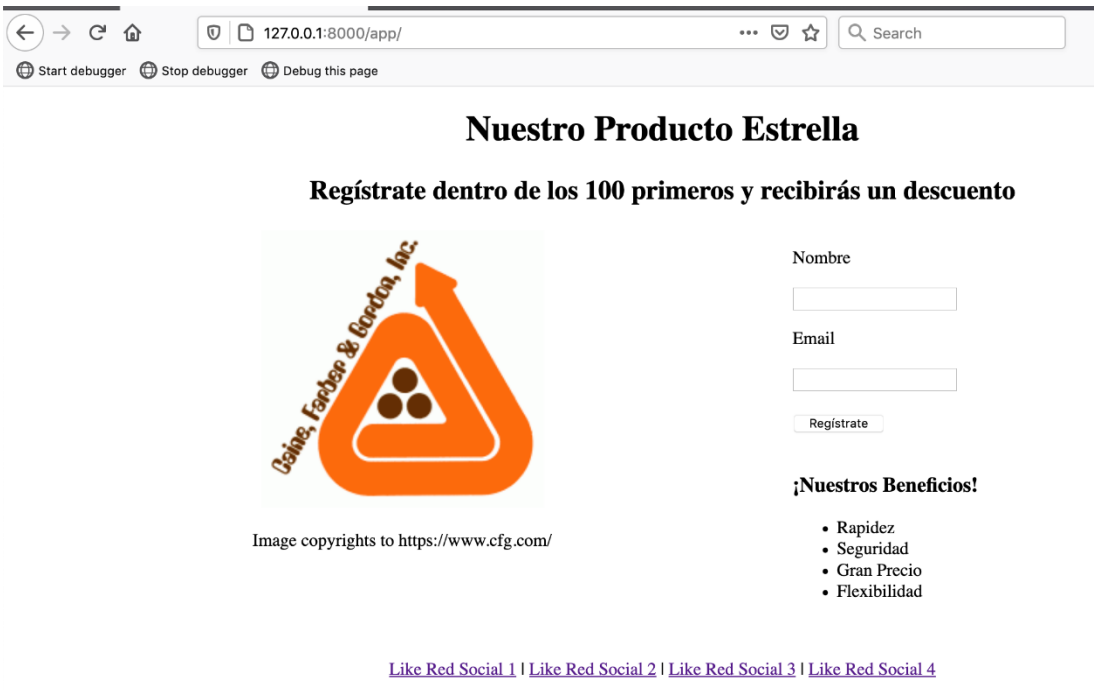


Dado que ya tenemos la plantilla "base" ahora vamos a llenar lo que nos falta para reconstruir **index.html**. La palabra clave es **extends** con ella estamos declarando que **index.html** es una extensión de **base.html**

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

{% block title %}Landing Page{% endblock %}

{% block header %}
      <h1>Nuestro Producto Estrella</h1> <h2>Regi´strate dentro de los 100
primeros    y recibira´s un descuento</h2>
{% endblock %}

{% block imagen %}
<img src="https://www.cfg.com/images/cfg1a_new.gif"
     height="250px">
    <p>Image copyrights to https://www.cfg.com/</p>
{% endblock %}

{% block formulario %}
    <form>
        <p>Nombre</p>
        <input type="text">
        <p>Email</p>
        <input type="email"/>
        <br>
        <br>
        <button type="submit">Regi´strate</button>
    </form>
    <br>
    <h3>¡Nuestros Beneficios!</h3>
    <ul>
```

```
        <li>Rapidez</li>
        <li>Seguridad</li>
        <li>Gran Precio</li>
        <li>Flexibilidad</li>
    </ul>
{% endblock %}

{% block footer %}
    <br>
    <p>
        <a href="">Like Red Social 1</a> |
        <a href="">Like Red Social 2</a> |
        <a href="">Like Red Social 3</a> |
        <a href="">Like Red Social 4</a>
    </p>
    <br>    <p>Copyright &copy; 2019-2020 -
        Desarrollos de Landing Pages Inc.</p>
{% endblock %}
```

El resultado final es exactamente el mismo, pero nuestro código está mejor organizado y los archivos son más pequeños (algo siempre deseable)



6.3.1.2. Vistas en Django

Hasta el momento hemos visto `views` basadas en funciones. No obstante, Django nos ofrece también vistas basadas en clases. Vamos a cambiar nuestra vista ara usar esta característica. Vamos a usar la clase **TemplateView**. Todas las vistas (**views**) heredan de la clase **View**, que se encarga de vincular la vista a las URL, el envío del método HTTP y otras características comunes. **TemplateView** extiende la clase base para que también represente una plantilla.

Cambiamos nuestro archivo **sitio/app/urls.py**

```
from django.urls import path
from django.views.generic import TemplateView

urlpatterns = [
    path('', TemplateView.as_view(template_name="app/index.html"))
]
```

Este cambio produce el mismo efecto, pero ahora estamos usando la clase **TemplateView** que heredó de **View**.

6.3.1.3. Herencia de vistas

Podemos ir un poco más allá, vamos a intervenir nuestro archivo **sitio/app/views.py**

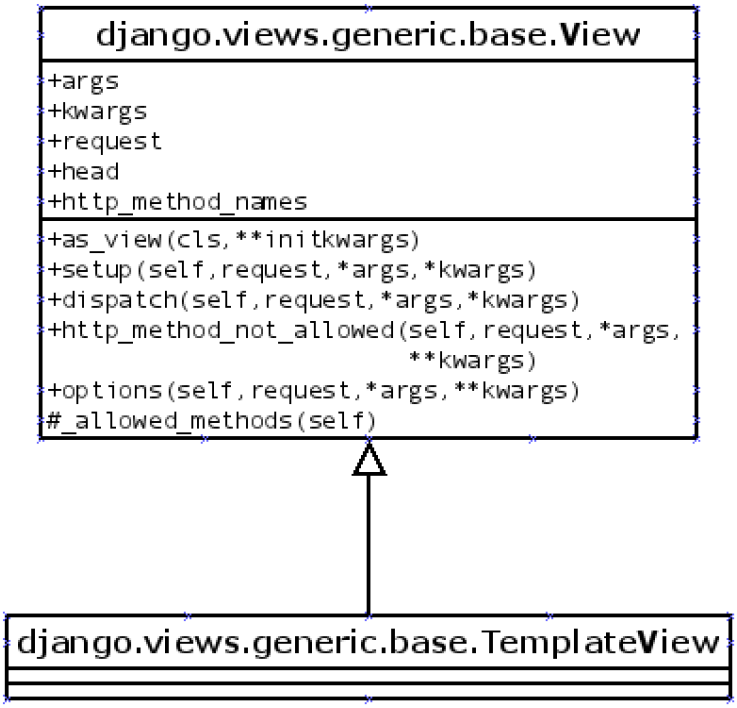
```
from django.views.generic import TemplateView

class IndexView(TemplateView):
    template_name = "app/index.html"
```

y también modificaremos **sitio/app/urls.py**

```
from django.urls import path
from .views import IndexView

urlpatterns = [
    path('', IndexView.as_view())
]
```

y nuestro resultado es el mismo si observamos http://127.0.0.1:8000/app/

Ahora bien ¿qué ventaja tiene hacer estos cambios? observemos la figura de abajo



Como **TemplateView** hereda de **View**, contamos con todos los atributos y métodos de esta última clase. Nota: no se muestran a todos los métodos disponibles para **TemplateView**, ya que aparte de heredar de **View** también implementa métodos de otros archivos (que son **Mixins** pero tal tópico está fuera del alcance de nuestros estudios por ahora).

6.3.1.4. Templates en Django

Para esta sección vamos a crear una página básica solo para mostrar conceptos relacionados con templates, la nueva plantilla es **sitio/app/templates/app/foo.html** también haremos que extienda **base.html.**

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

{% block title %}Foo Page{% endblock %}

{% block header %}
    <h1>Este es el header</h1>
{% endblock %}

{% block imagen %}
    <h2>Acá prodriamos poner una imagen</h2>
{% endblock %}
```

```
{% block formulario %}
    <h2>Acá prodriamos poner un formulario</h2>
{% endblock %}

{% block footer %}
    <h1>Este es el footer</h1>
{% endblock %}
```

6.3.1.5. Inclusión de Templates

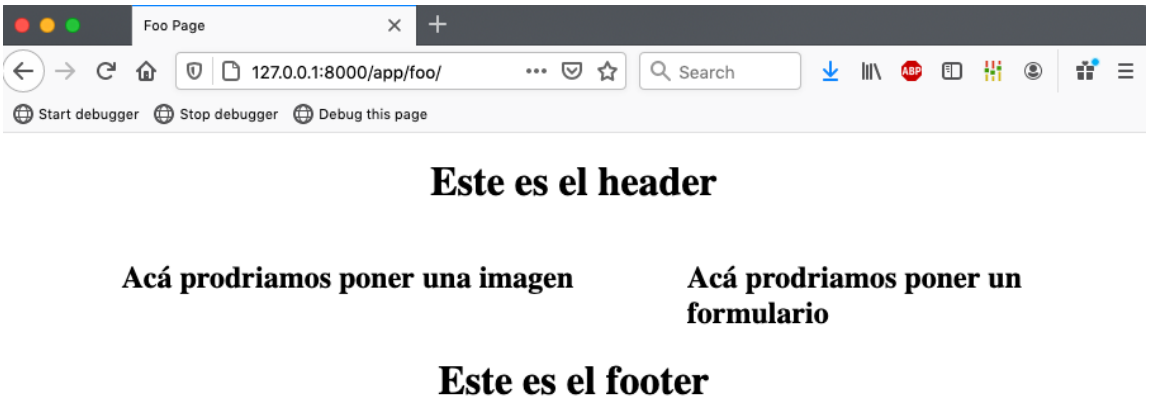
Para incluir esta nueva plantilla seguiremos los pasos que hemos estado viendo en **sitio/app/urls.py:**

```
from django.urls import pathfrom .views import IndexViewfrom .views
import FooView
urlpatterns = [
    path('', IndexView.as_view()),
    path('foo/', FooView.as_view())
]
```

El archivo **sitio/app/views.py** contiene

```
from django.views.generic import TemplateView
class IndexView(TemplateView):
    template_name = "app/index.html"
class FooView(TemplateView):
    template_name = "app/foo.html"
```

Si nos dirigimos a http://127.0.0.1:8000/app/foo/ deberíamos ver:



6.3.1.6. Variables en Plantillas

Hasta el momento nuestros templates y páginas no son realmente funcionales ya que no pueden desplegar variables.

Despliegue

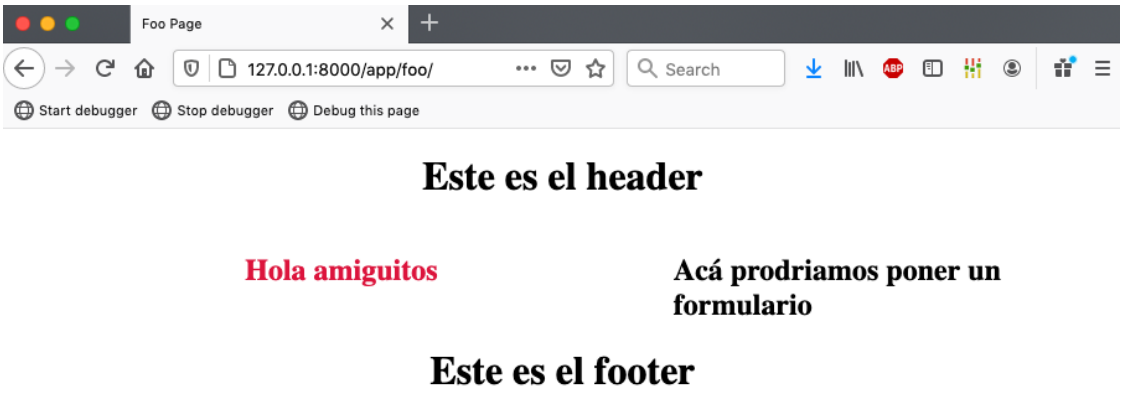
Una vez más modificaremos nuestros archivos, en **sitio/app/views.py** vamos a sobreescribir el método **get** de la clase **TemplateView** (estamos heredando de esta clase) y usamos la función **render**

```
from django.views.generic import TemplateViewfrom django.shortcuts
import render
...
class FooView(TemplateView):
    template_name = "app/foo.html"

    def get(self, request, *args, **kwargs):
        context = {'mensaje': 'Hola amiguitos'}
        return render(request, "app/foo.html", context=context)
```

Nuestra idea es pasar la variable **mensaje** (su contenido) a la plantilla **foo.html**, vamos a modificar el bloque **imagen** solo por elegir algún bloque al azar.

```
{% extends "base.html" %}...
{% block imagen %}
    <h2 style="color:crimson">{{ mensaje }}</h2>
{% endblock %}
...
```

La variable (envuelta en corchetes dobles **{{ variable }}**) debería ahora ser reconocida por nuestra plantilla, y así es:



Iteradores

Ya sabemos que podemos pasar variables desde la vista hacia la plantilla, ahora veremos algunas de las flexibilidades que nos permiten las plantillas en Django.

For

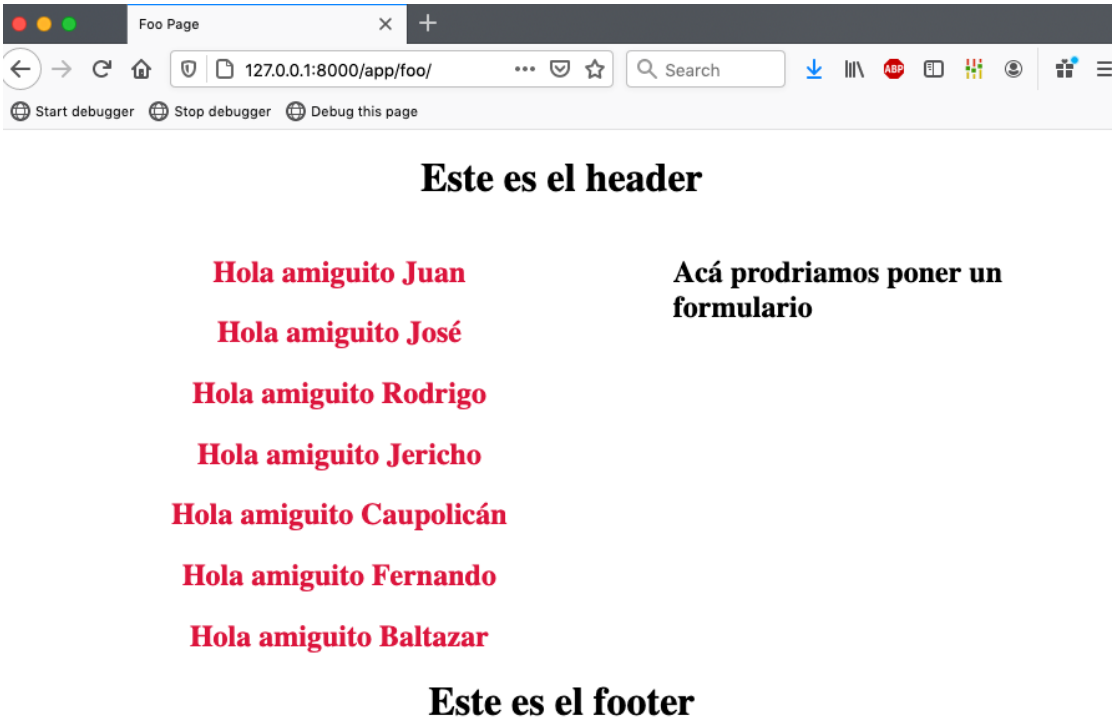
Una vez más alteramos nuestra vista **sitio/app/views.py** ahora queremos pasar una "lista" de nombres de algunos amigos como variable

```
...
class FooView(TemplateView):
    template_name = "app/foo.html"

    def get(self, request, *args, **kwargs):
        context = {'amigos': ['Juan', 'José', 'Rodrigo', 'Jericho',
                              'Caupolicán', 'Fernando', 'Baltazar']}
        return render(request, "app/foo.html", context=context)
```

Y consecuentemente alteraremos nuestra plantilla foo.html esta vez iterando sobre nuestra variable

```
...
{% block imagen %}
    {% for amigo in amigos %}
        <h2 style="color:crimson">
            Hola amiguito {{ amigo }}
        </h2>
    {% endfor %}{% endblock %}...
```

Si vamos a http://127.0.0.1:8000/app/foo/ veremos



Condiciones

If/elif/else

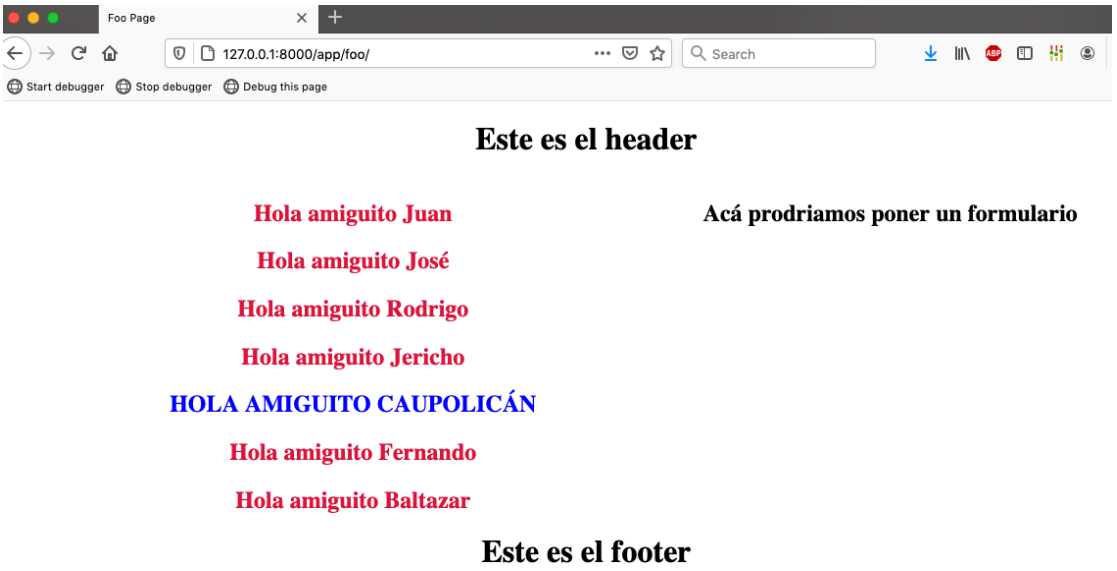
Estas condiciones son muy similares a lo que hemos visto en otras partes solo que esta vez están aplicadas a plantillas.

Haremos algo similar al ejercicio anterior, esta vez si encontramos a nuestro amiguito Caupolicán haremos que se vea en azul. Solo modificamos nuestra plantilla **foo.html**

```
{% extends "base.html" %}...
{% block imagen %}
    {% for amigo in amigos %}
        {% if amigo == 'Caupolicán' %}
            <h2 style="color:blue">
                Hola amiguito {{ amigo }}
            </h2>
        {% else %}
            <h2 style="color:crimson">
                Hola amiguito {{ amigo }}
            </h2>
        {% endif %}
    {% endfor %}{% endblock %}...
```

Aplicación de filtros

Apliquemos otra simple modificación en nuestra plantilla, cuando se trate de 'Caupolicán' hagamos que la línea se muestra en mayúsculas.



Como se puede inferir las formas de manipular las plantillas son muchas para un detalle de las funciones disponibles puede ver https://docs.djangoproject.com/en/3.1/ref/templates/builtins/

6.3.1.7. Contenido estático

STATIC_URL

Para manejar archivos como imágenes, archivos javascript, videos incrustados, CSS, etc. Django nos ayuda con la app **django.contrib.staticfiles**. Todas esta clase de archivos son conocidos como "static files"

Lo primero que necesitamos es "instalar" la app **django.contrib.staticfiles** en nuestro proyecto si es que no lo está, eso lo hacemos en **sitio/sitiodeprueba/settings.py** variable **INSTALLED_APPS**

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

```
]...
```

Al final del mismo archivo debería encontrar la variable **STATIC_URL** como solo estamos trabajando con nuestra aplicación **app** vamos a configurar esa ruta

```
...
STATIC_URL = 'app/static/'
```

Ahora necesitamos crear un directorio **static**; seguimos la misma convención que usamos en templates: **sitio/app/static/app/**. Dentro de esta carpeta pondremos una imagen de la controvertida escultura de Caupolicán hecha por Nicanor Plaza https://es.wikipedia.org/wiki/Nicanor_Plaza#Controversia_por_Caupolic%C3%A1n

En **sitio/app/templates/app/foo.html** hacemos unos pequeños cambios

```
{% extends "base.html" %}...{% block formulario %}
    <h2>Acá prodriamos poner un formulario</h2>
    <h3>Pero mejor usamos una imagen</h3>
{% load static %}
<img src="{% static 'app/Caupolican.jpg' %}" alt="Caupolicán"
width="300" height="500">
{% endblock %}
...
```

Naturalmente usted puede usar la imagen que desee. Al visitar http://127.0.0.1:8000/app/foo/

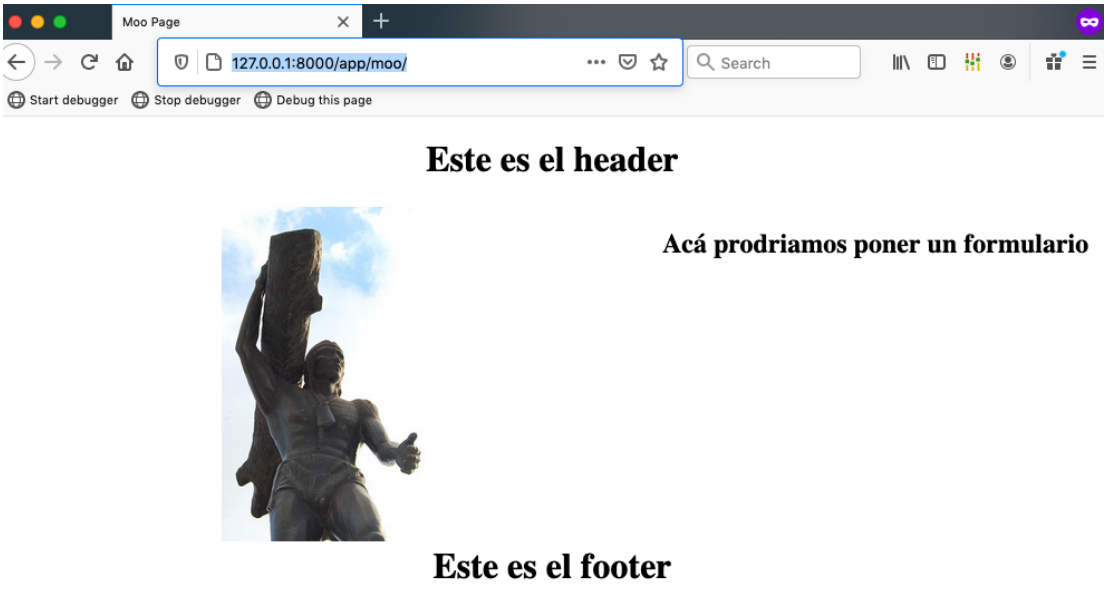
Hasta acá hemos usado algunas variables dela configuración de Django, se puede ver la documentacion oficial para mayor información https://docs.djangoproject.com/en/3.1/ref/settings/#static-files

6.3.1.8. Etiquetas URL en plantillas y redireccionamiento

Vamos a crear otra página con todo lo que esto implica a partir de la plantilla **base.html**

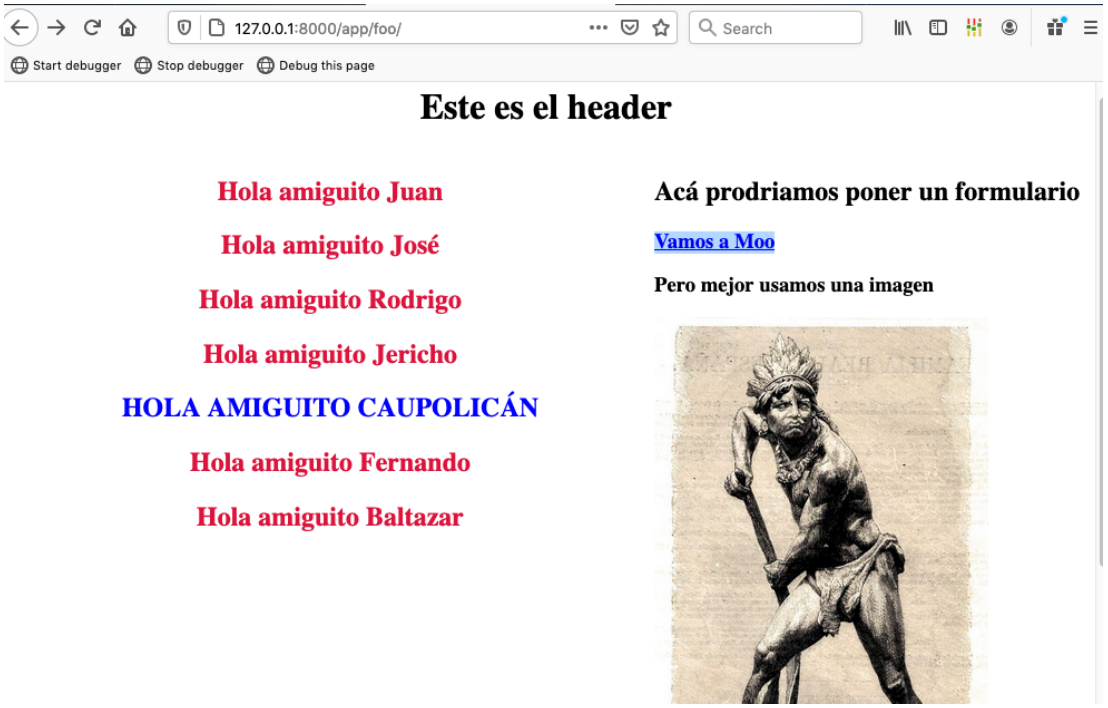
Vamos a omitir los detalles de la nueva página **moo.html**, ya que, son análogos, a la página **foo.html**

Ahora tenemos una nueva ruta http://127.0.0.1:8000/app/moo/



Ahora bien, qué sucedería si estando en nuestra página **foo** quisiéramos poner un enlace que nos redireccionara a **moo**. Incluir la ruta completa sería bastante inconveniente, ya que, siempre tendríamos que incluir en nuestra plantilla un enlace a **http://127.0.0.1:8000/app/moo/**, si tenemos muchas plantillas esto sería tedioso y daría una mayor cabida a errores.

En nuestra planilla **foo** ahora tenemos un enlace (izquierda) que dice "Vamos a Moo"



Esto es bastante simple de hacer, en **sitio/app/urls.py** incluimos un alias para la plantilla:

```
...urlpatterns = [
        ...
     path('moo/', MooView.as_view(), name='moo'),
]
```

y en la planilla **foo.html** usamos la palabra clave **url** y nuestro alias **moo**

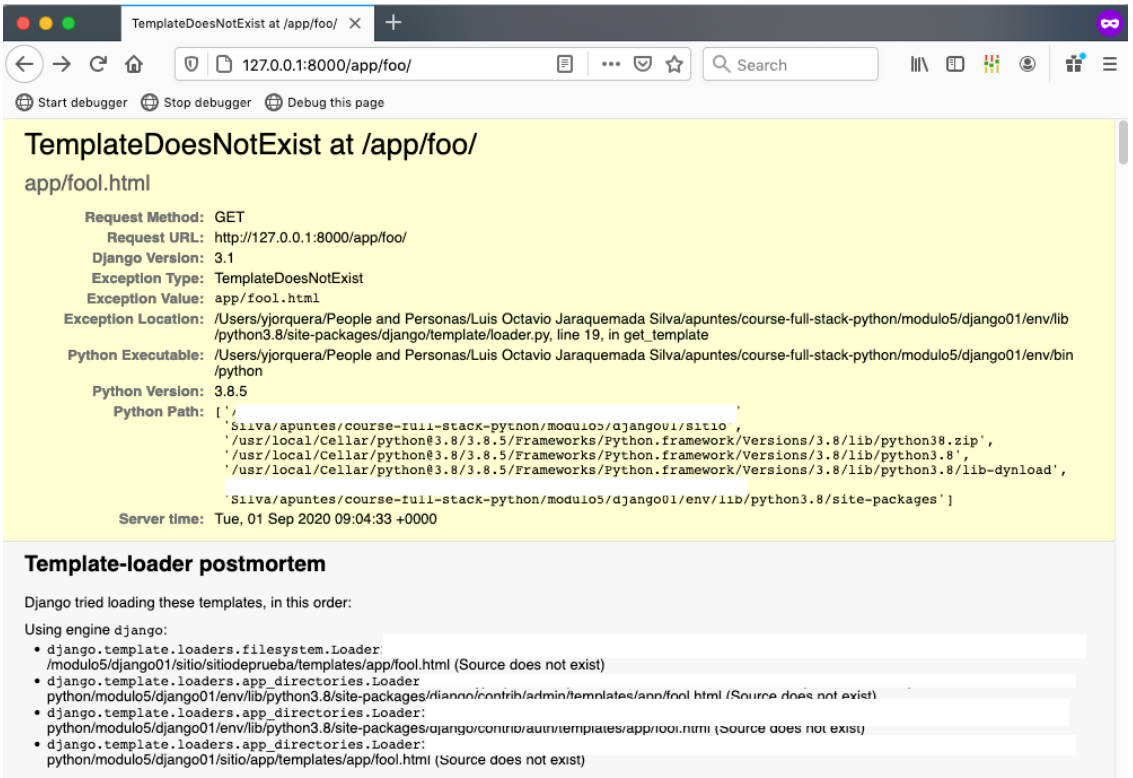
```
...{% block formulario %}
    <h2>Acá prodriamos poner un formulario</h2><h3><a href="{% url
'moo' %}">Vamos a Moo</a></h3>
    <h3>Pero mejor usamos una imagen</h3>{% load static %}
<img src="{% static 'app/Caupolican.jpg' %}" alt="Caupolicán"
    width="300" height="500">{% endblock %}...
```

6.3.1.9. Manejo de errores

Vamos a simular un error para que veamos las páginas que muestra django en tales casos. Supongamos que en **sitio/app/views.py** cometimos un error/type con el nombre de la template y en lugar de escribir **foo** escribimos **fool**

```
class FooView(TemplateView):
    def get(self, request, *args, **kwargs):
        ...
        return render(request, "app/fool.html", context=context)
```

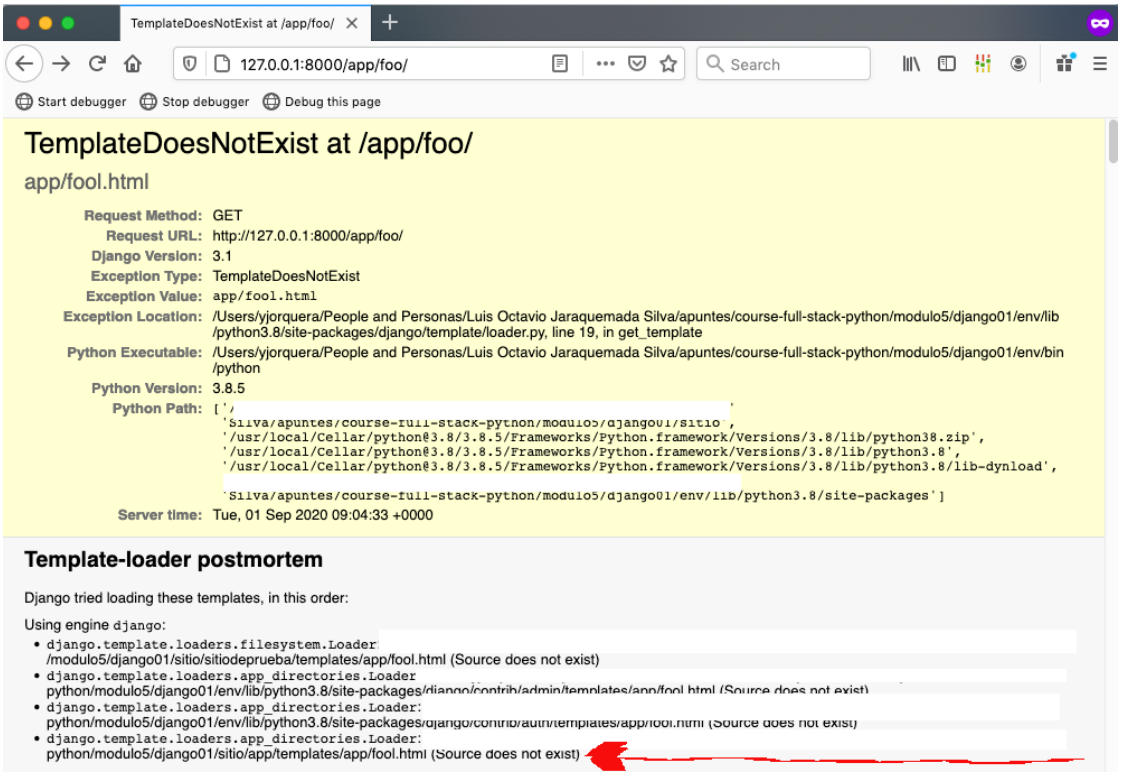
Veamos que Django nos informa del error con bastantes detalles para que sepamos donde está



Claramente el problema radica en lo que ve en la parte de arriba de la página de error:

```
TemplateDoesNotExist at /app/foo/app/fool.html
```

También tenemos muchas más pistas en esta página de error, por ejemplo, en la parte roja nos indica que **sitio/app/templates/app/fool.html (Source does not exist).** Usualmente si uno no está seguro de cuál es el problema, las páginas de error contienen suficiente información como para deducir donde está en problema.

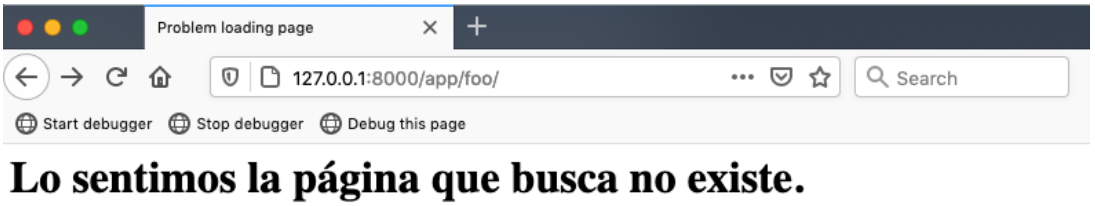


Naturalmente esto o es algo que queremos mostrar al usuario, este tipo de problemas también es evitable cuando estamos en un ambiente de producción de una forma distinta y más segura, pero ahora como estamos ilustrando problemas típicos lo hacemos en el entorno development.

Supongamos que no logramos dar con el problema, después de todo no es algo muy extraño cuando estamos desarrollando que un caracter sobrante o uno faltante nos arruine todo. Entonces qué podemos hacer, simplemente podemos captar la excepción por mientras. En **sitio/app/views.py** vamos a incluir lo siguiente:

```
from django.template import TemplateDoesNotExist...class
FooView(TemplateView):
    template_name = "app/foo.html"

    def get(self, request, *args, **kwargs):
        try:
            context = {'amigos': ['Juan', 'José', 'Rodrigo',
                    'Jericho', 'Caupolicán', 'Fernando', 'Baltazar']}
            return render(request, "app/fool.html", context=context)
        except TemplateDoesNotExist:
            return HttpResponse(
                    "<h1>Lo sentimos la página que busca no existe.</h1>")
```

y cuando el usuario va a http://127.0.0.1:8000/app/foo/ obtendrá:



Naturalmente acá estamos viendo ejemplos didácticos, muchísimas más cosas suceden cuando trabajamos en proyecto grande y en producción, pero la idea esencial es la misma.

6.3.1.10. Manejo de Raise

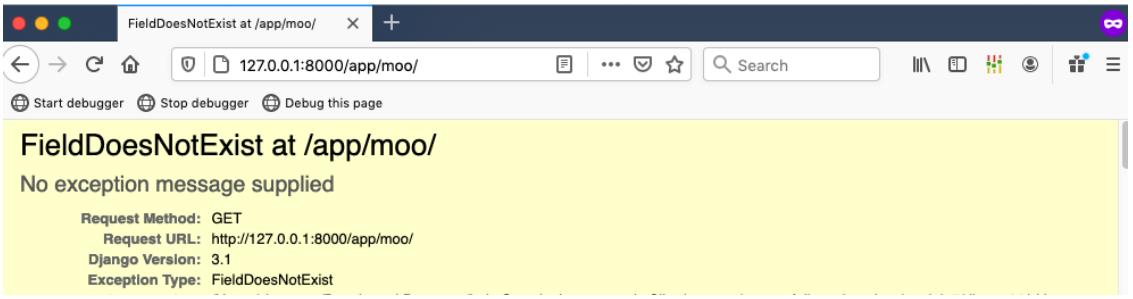
Acá tenemos una lista de excepciones generadas por Django (también contamos las de Python por definición). Atención estas son excepciones generales del core de Django **django.core.exceptions**. en el ejemplo que vimos anteriormente usamos excepciones Django que tienen que ver con templates **.../django/template/exceptions.py** que nos ofrece:

```
...# Public exceptions
from .base import VariableDoesNotExist                              # NOQA
isort:skip
from .context import Context, ContextPopException, RequestContext  # NOQA
isort:skip
from .exceptions import TemplateDoesNotExist, TemplateSyntaxError  # NOQA
isort:skip
```

Continuemos con las **core** exceptions, supongamos que necesitamos hacer ciertas operaciones en **MooView** en **sitio/app/views.py**; esas operaciones nos tienen que entregar un resultado que se debe guardar en la variable **result**, lo cual es crítico para nuestra aplicación, no obstante esa lista regresa vacía, eso lo emulamos acá:

```
class MooView(TemplateView):
    template_name = "app/moo.html"
    def get(self, request, *args, **kwargs):
        # lógica que debe llenar de valores `result`
        result = []
        if not result:
            raise EmptyResultSet
        context = {}
             return render(request, "app/moo.html", context=context)
```

Como **result** está vacío levantamos la excepción **EmptyResultSet**



Esto se hace por lo general cuando tenemos partes críticas de código y no podemos permitir que algo suceda mal. Entonces hacemos explícito el error, naturalmente solo visible para la fase desarrollo, si tal situación acontece seguramente el usuario verá un Error 500 o algo similar.

6.3.2.- Referencias

[1] Django Documentation
https://docs.djangoproject.com/en/3.1/

[2] D. Feldroy & A. Feldroy , Two Scoops of Django 3.x, Best Practices for the Django Web Frame-work, 2020.

[3] William S. Vincent, Django for Beginners Build websites with Python & Django, 2020.

[4] Nigel George, Build a Website with Django 3, 2019.

[5] Django Tutorial
https://www.geeksforgeeks.org/django-tutorial/