HINTS

CONSEJOS CONCEPTUALES

- Cargando (`views.py`) - Consiste en encontrar la plantilla para un determinado identificado y preprocesar.

- Representación (`views.py`) - Inyecta el contexto (datos) en la plantilla.

- Contexto (`views.py`) - Diccionario como claves de asignación de objetos a valores.

- Variables (`template.html`) - Genera un valor del contexto, y está rodeado por corchetes dobles.
    - Por ejemplo: esta plantilla la cadena '`Hola {{name}}.`', rendiría 'Hola, Juan' del `contexto{'name': 'Juan'}`

- Etiquetas - Lógica que se utilizará en el proceso de renderizado. Es una definición intencionalmente amplia, ya que el texto se puede usar como estructura de control, o contenido de salida. Consulte los documentos de Django para obtener más información al respecto.

ESTRUCTURA DE CARPETAS DE PLANTILLAS

Las carpetas de plantillas se pueden organizar de dos maneras. Nivel de aplicación:

```
├── my_proj
|   ├── config # Optional keeps config separate from apps
|   ├── apps. # Optional all apps in one folder
|   |   └── my_app
|   |       ├── __init__.py
|   |       ├── models.py
|   |       └── views.py
|   |       ├── templates
|   |           ├── my_app
|   |               ├── home.html
└── manage.py
```

- Las plantillas se guardan dentro de cada aplicación.
- Se usó el nombre de la aplicación '`my_app`' en la carpeta para evitar colisiones de espacios de nombres.
- Este enfoque es adecuado para proyectos más pequeños.

Nivel de proyecto:

```
├── my_proj
|   └── config # Optional keeps config separate from apps
|       ├── __init__.py
|       ├── settings.py
|       ├── urls.py
|       ├── asgi.py
|       └── wsgi.py
|   └── apps # Optional all apps in one folder
|       ├── my_app
|       ├── another_app
├── templates
|   ├── base.html
|   └── my_app
|       ├── detail.html
|   └── another_app
|       ├── detail.html
└── manage.py
```

Aquí podemos ver la jerarquía del orden de llamadas de las plantillas, ya que podemos ver que las Plantillas se guardan en una carpeta principal y las Plantillas adicionales para cada aplicación se guardan en una carpeta separada.

- Adecuado para proyectos más grandes. A medida que crece la cantidad de aplicaciones, se vuelve más engorroso buscar plantillas de varias aplicaciones.
- Notará que se prefiere mantener la configuración y las aplicaciones en carpetas separadas. Esta es una desviación de la aplicación de ejemplo que se muestra en los documentos de Django. Es la forma recomendada de estructurar grandes proyectos.

HERENCIA DE PLANTILLAS: PLANTILLAS GENÉRICAS Y SECUNDARIAS

Las plantillas de Django contendrán una gran cantidad de código que es común en la mayoría de sus vistas. Una característica poderosa del lenguaje de plantillas de Django es la herencia, que nos permite construir una plantilla de "esqueleto" base que contiene elementos comunes y bloques que las plantillas secundarias pueden anular.

DEFINICIÓN DE UNA PLANTILLA GENÉRICA

Se considera una buena práctica definir una plantilla genérica de la que se puedan heredar otras plantillas. Esto mantiene nuestras plantillas HTML DRY, mantiene el código en un solo lugar, y facilita su modificación.

```
<!--base.html-->
{% load static some_other_tag %} {# >>1 - Load tags#}
<!DOCTYPE html>
<html>
<head>
    {# Use inbuilt block tag for variables that we can override from
other templates #}
    <title>{% block title %}Home{% endblock %}</title>

    {# include <meta> tags including tags to make your site SEO friendly
#}

    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1,
shrink-to-fit=no">

    {# include styling #}
    <link href="{% static "css/project.css" %}" rel="stylesheet">
    {% block head_css %}
      <link href="{% static "css/other.css" %}" rel="stylesheet">
    {% endblock %}
    {# include javascript #}
    <script                                                    defer
src="https://cdn.jsdelivr.net/gh/alpinejs/alpine@v3.0.1/dist/alpine.min
.js"></script>
    {% block head_js %}{% endblock %}
</head>
<body>
    {% block content %}{% endblock %}
    {% block footer_js %}{% endblock %}

</body>
</html>
```

- Incluye `<meta>` tags, scripts de página, y estilos que se pueden reutilizar en otras páginas.
- Utiliza el block tag i.e. `{% block x %} {% endblock %}` que se puede anular de otras plantillas.

BLOCK Y ENDBLOCK

La etiqueta de `block` tiene dos funciones:

- Es un marcador de posición para el contenido.

- Es el contenido que reemplazará al marcador de posición.

En las plantillas principales, la etiqueta `block` ***es un marcador*** de posición que será reemplazado por un bloque en una plantilla secundaria con el mismo nombre. En las plantillas secundarias, la etiqueta `block` ***es contenido*** que reemplazará el marcador de posición en la plantilla principal con el mismo nombre. La etiqueta `endblock` es solo el la etiqueta de cierre de `block`.

PLANTILLAS SECUNDARIAS

Una plantilla secundaria de ejemplo que amplía la plantilla base se vería así.

```
<!--base.html-->
{% load static some_other_tag %} {# >>1 - Load tags#}
<!DOCTYPE html>
<html>
<head>
    {# Use inbuilt block tag for variables that we can override from
other templates #}
    <title>{% block title %}Home{% endblock %}</title>

    {# include <meta> tags including tags to make your site SEO friendly
#}

    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1,
shrink-to-fit=no">

    {# include styling #}
    <link href="{% static "css/project.css" %}" rel="stylesheet">
    {% block head_css %}
      <link href="{% static "css/other.css" %}" rel="stylesheet">
    {% endblock %}

    {# include javascript #}
    <script                                                    defer
src="https://cdn.jsdelivr.net/gh/alpinejs/alpine@v3.0.1/dist/alpine.min
.js"></script>
    {% block head_js %}{% endblock %}
</head>
<body>
    {% block content %}{% endblock %}
    {% block footer_js %}{% endblock %}

</body>
</html>
```

- Utiliza el '`extends tag`' que le dice al motor de plantillas que extienda otra plantilla. En este caso, base.html.

- Anula las etiquetas de título y contenido para inyectar contenido específico de la página.
- Notará que la etiqueta `head_css` usa la variable `{{block.super}}`. Esto es útil para agregar componentes del bloque principal en lugar de anularlo por completo. En el ejemplo, el contenido de other.css también estará disponible en esta página.

GENERADORES EN PYTHON

Una función generadora se define como una función normal, pero siempre que necesita generar un valor, lo hace con la palabra clave yield en lugar de return.

La declaración de rendimiento suspende la ejecución de una función y devuelve un valor a la persona que llama, pero retiene suficiente estado para permitir que la función se reanude donde se quedó. Cuando la función se reanuda, continúa la ejecución inmediatamente después de la última ejecución de rendimiento, lo que permite que el código produzca una serie de valores a lo largo del tiempo.

Si el cuerpo de una definición contiene rendimiento, la función se convierte automáticamente en una función generadora. Las funciones de generador devuelven un objeto generador, que se usa llamando al siguiente método en el objeto generador o usando el objeto generador en un bucle "for in", como se muestra en el siguiente ejemplo.

```
def funcGenerador():
    yield 1
    yield 2
    yield 3

# código para evaluar el generador:
for valor in funcGenerador():
    print(valor)
```