TEXT CLASS REVIEW

TEMAS A TRATAR EN EL CUE

- Templates en Django.
- Variables en Plantillas.
- Aplicación de filtros Contenido estático.
- STATICFILES_DIR.
- Herencia de plantillas.
- Bloques (block – endblock).
- Extends de plantillas.
- Modificación de datos en plantillas hijas.
- Etiquetas URL en plantillas y redireccionamiento.
- Manejo de errores.
- Manejo de Raise.
- Creando modelos.

TEMPLATES EN DJANGO

Al ser un framework, Django necesita una forma conveniente de generar HTML dinámicamente. El enfoque más común se basa en plantillas. Una plantilla contiene las partes estáticas de la salida HTML deseada, así como alguna sintaxis especial que describe cómo se insertará el contenido dinámico.

Un proyecto de Django se puede configurar con uno o varios motores de plantillas (o incluso cero si no usa plantillas). Django incluye backends integrados para su propio sistema de plantillas, creativamente denominado lenguaje de plantillas de Django (DTL), y una popular alternativa Jinja2.

Django define una API estándar para cargar y renderizar plantillas independientemente del backend. La carga consiste en encontrar la plantilla para un identificador determinado y preprocesarla, generalmente compilándola en una representación en memoria. Representar significa interpolar la plantilla con datos de contexto, y devolver la cadena resultante.

Tanto el soporte genérico para motores de plantillas, como la implementación del lenguaje de plantillas Django, viven en el espacio de nombres `django.template`.

Una plantilla se representa con un contexto. La representación reemplaza las variables con sus valores, que se buscan en el contexto, y ejecuta etiquetas. Todo lo demás sale como está. La sintaxis del lenguaje de plantilla Django implica cuatro construcciones: variables (`variable`), etiquetas (`tags`), filtros (`filters`), comentarios (`comments`).

Una plantilla es un archivo de texto que determina la estructura o diseño de un archivo (como una página HTML), con marcadores usados para representar el contenido real. Django automáticamente buscará plantillas en un directorio llamado `'templates'` de su aplicación. Así por ejemplo, la función render() esperará poder encontrar el archivo `/locallibrary/catalog/Templates/index.html`, y entregará un error si el archivo no puede ser encontrado: ahora se entregará un mensaje de error bastante intuitivo "`TemplateDoesNotExist` at `/catalog/`".

INCLUSIÓN DE TEMPLATES

La etiqueta de inclusión le permite incluir una plantilla dentro de la plantilla actual. Esto es útil cuando tiene un bloque de contenido que es el mismo para muchas páginas. El nombre de la plantilla puede ser una variable o una cadena codificada (entre comillas), entre comillas simples o dobles.

Por ejemplo, el contenido de la plantilla "`foo/bar.html`":

```
{% include "foo/bar.html" %}
```

Normalmente, el nombre de la plantilla es relativo al directorio raíz del cargador de plantillas. Un argumento de cadena también puede ser una ruta relativa que comience con `./` o `../`, como se describe en la etiqueta `extends`. Este ejemplo incluye el contenido de la plantilla cuyo nombre está contenido en la variable `template_name`:

```
{% include template_name %}
```

La variable también puede ser cualquier objeto con un método `render()` que acepte un contexto. Esto le permite hacer referencia a un compilado. La variable también puede ser cualquier objeto con un método `render()` que acepte un contexto.

Además, la variable puede ser iterable de nombres de plantilla, en cuyo caso se utilizará el primero que se pueda cargar, según `select_template()`.

Una plantilla incluida se representa dentro del contexto de la plantilla que la incluye. Este ejemplo produce la salida: "¡Hola, Juan!".

El Contexto: la variable persona se establece en "Juan", y la variable saludo se establece en "Hola".

El Template:

```
{% include "fragmento_nombre.html" %}
```

La plantilla del `fragmento_nombre.html`:

```
{{ saludo }}, {{ persona|default:"amigo" }}!
```

Puede pasar un contexto adicional a la plantilla utilizando argumentos de palabras clave:

```
{% include "fragmento_nombre.html" with persona="Juan" saludo="Hola" %}
```

Si desea representar el contexto solo con las variables proporcionadas (o incluso sin ninguna variable), use la opción `only`. No hay otras variables disponibles para la plantilla incluida:

```
{% include "fragmento_nombre.html" with saludo="Hola" only %}
```

VARIABLES EN PLANTILLAS

El framework Django, llama al método `render()` del objeto Template con un conjunto de variables (o sea, el contexto). Este retorna una plantilla totalmente renderizada como una cadena de caracteres, con todas las variables y etiquetas de bloques evaluadas de acuerdo con el contexto.

Cuando el motor de plantillas encuentra una variable, la evalúa y la reemplaza con el resultado. Los nombres de variables consisten en cualquier combinación de caracteres alfanuméricos, y el guión bajo (`"_"`), pero no pueden comenzar con un guion bajo y no pueden ser un número. El punto (`"."`) también aparece en secciones variables, aunque tiene un significado especial, como se indica a continuación. Es importante destacar que no puede tener espacios o caracteres de puntuación en los nombres de las variables.

Utilice un punto (.) para acceder a los atributos de una variable. En las plantillas de Django, puede representar variables colocándolas entre corchetes `{{}}`, por ejemplo:

TEMPLATE.HTML:

```
<h1>Hola {{ nombre }}, bienvenido a este sitio Web</h1>
```

Crear variables en las vistas (view):

La variable nombre del ejemplo anterior se envió a la plantilla a través de una vista.

VIEWS.PY

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

def testing(request):
  template = loader.get_template('template.html')
  context = {
    'name': 'Juan',
  }
  return HttpResponse(template.render(context, request))
```

Como puede observar en la vista anterior, creamos un objeto llamado `context`, lo llenamos con datos, y lo enviamos como el primer parámetro en la función `template.render()`.

BUCLE O ITERADOR FOR

Un ciclo `for` se usa para iterar sobre una secuencia, como recorrer elementos en una matriz, una lista o un diccionario.

Por ejemplo:

TEMPLATE.HTML

```
<!DOCTYPE html>
<html>
<body>

{% for x in frutas %}
  <h1>{{ x }}</h1>
{% endfor %}

<p>En views.py puedes ver cómo se ve la variable de frutas.</p>

</body>
</html>
```

VIEWS.PY

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

def testing(request):
  template = loader.get_template('template.html')
  context = {

    'fruits': ['Manzana', 'Maduro', 'Naranja'],

  }
  return HttpResponse(template.render(context, request))
```

WHILE

A diferencia del ciclo `for`, el ciclo `while` no tiene una etiqueta de plantilla en Django, por lo que para implementar su funcionalidad en nuestros programas solo podemos hacerlo usando código Python en nuestro archivo views.py o cualquier otro archivo de Python. Para lo que sí podemos usar template tags son las estructuras condicionales `if`, `else` y `elif`.

CONDICIONES IF – ELIF - ELSE

La estructura `if - elif - else` es una forma común de controlar el flujo de un programa, lo que te permite ejecutar bloques de código específicos según el valor de algunos datos.

Una declaración if evalúa una variable, y ejecuta un bloque de código si el valor es verdadero.

Ejemplo:

TEMPLATE.HTML

```
<!DOCTYPE html>
<html>
<body>

{% if saludo == 1 %}
  <h1>Hola</h1>
{% endif %}

<p>En views.py se puede observar el valor de la variable saludo. </p>

</body>
</html>
```

VIEWS.PY

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

def testing(request):
  template = loader.get_template('template.html')
  context = {
    'saludo': 1,
  }
  return HttpResponse(template.render(context, request))
```

La palabra clave **elif** dice "si las condiciones anteriores no fueron ciertas, intente con esta condición".

Ejemplo:

TEMPLATE.HTML

```
<!DOCTYPE html>
<html>
<body>

{% if saludo == 1 %}
  <h1>Hola</h1>
{% elif saludo == 2 %}
  <h1>Bienvenido</h1>
{% endif %}

<p>En views.py se puede observar el valor de la variable saludo.</p>

</body>
</html>
```

```
15
```

VIEWS.PY

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

def testing(request):
  template = loader.get_template('template.html')
  context = {
    'saludo': 2,
  }
  return HttpResponse(template.render(context, request))
```

La palabra clave **else** captura cualquier cosa que no esté capturada por las condiciones anteriores.

Ejemplo:

TEMPLATE.HTML

```
<!DOCTYPE html>
<html>
<body>

{% if saludo == 1 %}
  <h1>Hola</h1>
{% elif saludo == 2 %}
  <h1>Bienvenido</h1>
{% else %}
  <h1>Hasta luego</h1>
{% endif %} }

<p>En views.py se puede observar el valor de la variable saludo.</p>

</body>
</html>
```

VIEWS.PY

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

def testing(request):
  template = loader.get_template('template.html')
  context = {
    'saludo': 3,
  }
  return HttpResponse(template.render(context, request))
```

APLICACIÓN DE FILTROS CONTENIDO ESTÁTICO

Puede modificar las variables para su visualización mediante el uso de filtros.

- Los filtros tienen este aspecto: `{{nombre|lower}}`. Esto muestra el valor de la variable nombre, después de filtrarla a través del filtro `lower`, que convierte el texto a minúsculas. Use una tubería `(|)` para aplicar un filtro.
- Los filtros se pueden "encadenar". La salida de un filtro se aplica al siguiente. `{{text|escape|linebreaks}}` es un modismo común para escapar de los contenidos de texto, y luego convertir los saltos de línea en etiquetas `<p>`.
- Algunos filtros aceptan argumentos. Un argumento de filtro tiene este aspecto: `{{bio|truncatewords:30}}`. Esto mostrará las primeras 30 palabras de la variable bio.
- Los argumentos de filtro que contienen espacios deben estar entrecomillados; por ejemplo, para unirse a una lista con comas y espacios, usaría `{{lista|unirse:", “}}`.

Django proporciona alrededor de sesenta filtros de plantilla integrados. Puede leer todo sobre ellos en la referencia de filtro integrada. Para darle una idea de lo que está disponible, estos son algunos de los filtros de plantilla más utilizados:

- default: si una variable es falsa o está vacía, use el valor predeterminado dado. De lo contrario, utilice el valor de la variable. Por ejemplo:

```
{{value|default:"nada"}}
```

Si no se proporciona el valor o está vacío, lo anterior mostrará "nada".

- Length: devuelve la longitud del valor. Esto funciona tanto para cadenas como para listas. Por ejemplo: template.html.

```
<!DOCTYPE html>
<html>
<body>
<h1>{{frutas|length}}</h1>
<p>En views.py se puede observar el contenido de la variable
frutas.</p>
</body>
</html>
```

VIEWS.PY

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

def testing(request):
  template = loader.get_template('template.html')
  context = {
    'fruits': ['Manzana', 'Maduro', 'Naranja'],
  }
  return HttpResponse(template.render(context, request))
```

MÚLTIPLES FILTROS

Puede agregar más de un filtro agregando tubería || caracteres seguidos de nombres de filtro.
Ejemplo:

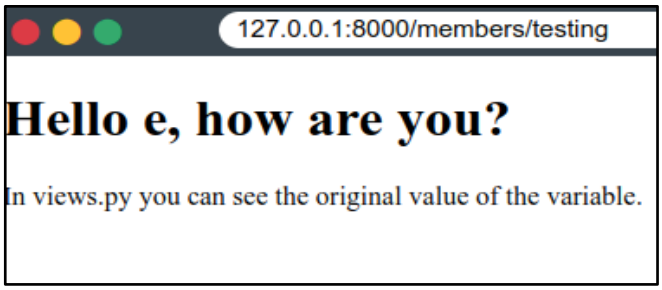
TEMPLATE.HTML

```
<!DOCTYPE html>
<html>
<body>
<h1>Hola {{ nombre|first|lower }}, bienvenido..!</h1>

<p>En views.py se puede observar el contenido de la variable
nombre.</p>

</body>
</html>
```

VIEWS.PY

```
from django.http import HttpResponse
from django.template import loader
def testing(request):
  template = loader.get_template('template.html')
  context = {
    'firstname': 'Juan',
  }
  return HttpResponse(template.render(context, request))
```

Se observa la salida:



LA ETIQUETA DEL FILTRO

Esta le permite ejecutar una sección completa de código a través de un filtro, y devolverlo de acuerdo con las palabras clave del filtro.

Por ejemplo, devolver el nombre de la variable con letras mayúsculas:

```
{% filter upper %}
  <h1>Hola a todos como están</h1>
{% endfilter %}
```

STATICFILES_DIR

Predeterminado: `[]` (Lista vacía)

Esta configuración define las ubicaciones adicionales que atravesará la aplicación `staticfiles` si el buscador `FileSystemFinder` está habilitado. Por ejemplo: si usa el comando de administración `collectstatic` o `findstatic`, o usa la vista de servicio de archivos estáticos. Esto debe establecerse en una lista de cadenas que contienen rutas completas a su(s) directorio(s) de archivos adicionales, por ejemplo:

```
STATICFILES_DIRS = [
    "/home/special.polls.com/polls/static",
    "/home/polls.com/polls/static",
    "/opt/webfiles/common",
]
```

Tenga en cuenta que estas rutas deben usar barras diagonales estilo Unix, incluso en Windows (por ejemplo, `"C:/Users/user/mysite/extra_static_content"`).

PREFIJOS (OPCIONAL)

En caso de que desee hacer referencia a archivos en una de las ubicaciones con un espacio de nombres adicional, puede proporcionar opcionalmente un prefijo como tuplas (prefijo, ruta). Por ejemplo:

```
STATICFILES_DIRS = [
    # ...
    ("downloads", "/opt/webfiles/stats"),
]
```

Suponiendo que tiene `STATIC_URL` establecido en `'static/'`, el comando de gestión `collectstatic` recopilará los archivos de "estadísticas" en un subdirectorio de 'descargas' de `STATIC_ROOT`. Esto le permitiría consultar el archivo local `'/opt/webfiles/stats/polls_20101022.tar.gz'` con `'/static/downloads/polls_20101022.tar.gz'` en sus plantillas. Ejemplo:

```
s<a href="{% static 'downloads/polls_20101022.tar.gz' %}">
```

HERENCIA DE PLANTILLAS

Es un patrón que se parece a las técnicas de programación orientada a objetos, en el que los bloques de contenido se insertan dentro de otras plantillas HTML. A través de las Herencias se puede crear una plantilla de base que contiene todos los elementos comunes de un sitio, y se define como “Ranuras” o “Bloques”.

La parte más poderosa, y por lo tanto la más compleja, del motor de plantillas de Django es la herencia de plantillas. Esta le permite crear una plantilla básica de "esqueleto" que contiene todos los elementos comunes de su sitio, y define bloques que las plantillas secundarias pueden anular.

Por ejemplo:

BASE.HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <link rel="stylesheet" href="style.css">
    <title>{% block title %}Mi sitio Web{% endblock %}</title>
</head>
<body>
    <div id="sidebar">
        {% block sidebar %}
```

```
        <ul>
            <li><a href="/">Inicio</a></li>
            <li><a href="/blog/">Blog</a></li>
        </ul>
        {% endblock %}
    </div>

    <div id="content">
        {% block content %}{% endblock %}
    </div>
</body>
</html>
```

Esta plantilla llamada base.html, define un documento de esqueleto HTML que puede usar para una página de dos columnas. Es el trabajo de las plantillas "secundarias" llenar los bloques vacíos con contenido.

En este ejemplo, la etiqueta de bloque define tres bloques que las plantillas secundarias pueden completar. Todo lo que hace la etiqueta de bloque es decirle al motor de plantillas que una plantilla secundaria puede anular esas partes de la plantilla.

Una plantilla secundaria podría verse así:

```
{% extends "base.htmMy amazing blogl" %}

{% block title %}Mi sitio de Blog{% endblock %}

{% block content %}
{% for entry in blog_entries %}
    <h2>{{ entry.title }}</h2>
    <p>{{ entry.body }}</p>
{% endfor %}
{% endblock %}
```

La etiqueta `extends` es la clave aquí. Le dice al motor de plantillas que esta plantilla "extiende" otra plantilla. Cuando el sistema de plantillas evalúa esta plantilla, primero localiza el padre, en este caso, "`base.html`".

En ese momento, el motor de plantillas notará las tres etiquetas de bloque en `base.html`, y reemplazará esos bloques con el contenido de la plantilla secundaria.

BLOQUES (BLOCK-ENDBLOCK)

La etiqueta de bloque tiene dos funciones:

- Es un marcador de posición para el contenido.
- Es el contenido que reemplazará al marcador de posición.

En las plantillas maestras, la etiqueta de bloque 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 de bloque es contenido que reemplazará el marcador de posición en la plantilla maestra con el mismo nombre.

La sintaxis es:

```
{% block name %}
  ...
{% endblock %}
```

Por ejemplo:

BASE.HTML

```
<!DOCTYPE html>
<html>
<body>
<h1>Bienvenidos</h1>
{% block userinfo %}
  <h2>Usuario no registrado</h2>
{% endblock %}
<p>Consulte las dos plantillas para ver cómo se ven y views.py para ver la referencia a la plantilla secundaria.</p>
</body>
</html>
```

VIEWS.PY

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

def testing(request):
  template = loader.get_template('secundariatemplate.html')
  return HttpResponse(template.render())

```

SECUNDARIATEMPLATE.HTML

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

{% block userinfo %}
  <h2>Juan De Jesus</h2>
  <p>Bienvenido a explorar el sitio Web</p>
{% endblock %}
```

En el ejemplo anterior, se observa que el contenido de una plantilla maestra, base.html, tiene un bloque llamado `userinfo`. Este bloque se reemplazará con un bloque llamado información de usuario en una plantilla secundaria.

EXTENDS DE PLANTILLAS

La etiqueta `extends` se usa para declarar una plantilla principal. Debe ser la primera etiqueta utilizada en una plantilla secundaria, y una plantilla secundaria solo puede extenderse hasta una plantilla principal.

MODIFICACIÓN DE DATOS EN PLANTILLAS HIJAS

Si la planilla hija no personaliza el bloque, heredará todo el contenido de la planilla padre por defecto (incluida la plantilla css, js). Si la placa hija necesita modificar el archivo css o js, agregue un bloque en el lugar correspondiente. Como se observó en secundariatemplate.html que extiende o hereda de `base.html`.

ETIQUETAS URL EN PLANTILLAS Y REDIRECCIONAMIENTO

Las URL devuelven una referencia de ruta absoluta (una URL sin el nombre de dominio) que coincide con una vista dada y parámetros opcionales. Cualquier carácter especial en la ruta resultante se codificará mediante `iri_to_uri()`.

Esta es una forma de generar enlaces sin violar el principio de DRY al tener que codificar las URL en las plantillas:

```
{% url 'some-url-name' v1 v2%}
```

El primer argumento es un nombre de patrón de URL. Puede ser un literal entrecomillado, o cualquier otra variable de contexto. Los argumentos adicionales son opcionales y deben ser valores separados por espacios que se usarán como argumentos en la URL. El ejemplo anterior muestra el paso de argumentos posicionales. Alternativamente, puede usar la sintaxis de palabras clave:

```
{% url 'some-url-name' arg1=v1 arg2=v2 %}
```

No mezcle la sintaxis posicional y de palabras clave en una sola llamada. Todos los argumentos requeridos por `URLconf` deben estar presentes.

Por ejemplo, suponga que tiene una vista, `app_views.client`, cuya `URLconf` toma una ID de cliente (aquí, `client()` es un método dentro del archivo de vistas app_views.py). La línea `URLconf` podría verse así:

APP_VIEWS.PY

```
path('client/<int:id>/', app_views.client, name='app-views-client')
```

Si la `URLconf` de esta aplicación se incluye en la `URLconf` del proyecto en una ruta como esta:

APP_VIEWS.PY

```
path('clients/', include('project_name.app_name.urls'))
```

Luego, en una plantilla, puede crear un enlace a esta vista como este:

```
{% url 'app-views-client' client.id %}
```

La etiqueta de la plantilla generará la cadena `/clients/client/123/`. Tenga en cuenta que, si la URL que está invirtiendo no existe, obtendrá una excepción `NoReverseMatch`, lo que hará que su sitio muestre una página de error. Si desea recuperar una URL sin mostrarla, puede usar una llamada ligeramente diferente. El ámbito de la variable creada por la sintaxis as var es el `{% block %}` en el que aparece la etiqueta `{% url %}`.

Esta sintaxis `{% url ... as var %}` no generará un error si falta la vista. En la práctica, usará esto para vincular vistas que son opcionales:

```
{% url 'some-url-name' as the_url %}
{% if the_url %}
  <a href="{{ the_url }}">Un enlace opcional</a>
{% endif %}
```

Si desea recuperar una URL con espacio de nombres, especifique el nombre completo:

```
{% url 'myapp:view-name' %}
```

MANEJO DE ERRORES

Las opiniones del marco REST manejan varias excepciones, y se ocupan de devolver las respuestas de error apropiadas.

Las excepciones manejadas son:

- Subclases de APIException generadas dentro del marco REST.
- Excepción Http404 de Django.
- PermissionDenied de Django Excepción rechazada.

En cada caso, el marco REST devolverá una respuesta con un código de estado, y un tipo de contenido apropiado. El cuerpo de la respuesta incluirá cualquier detalle adicional sobre la naturaleza del error.

MANEJO DE RAISE

Son una herramienta muy potente que la gran mayoría de lenguajes de programación modernos tienen. Se trata de una forma de controlar el comportamiento de un programa cuando se produce un error.

Esto es muy importante ya que salvo que tratemos este error, el programa se parará, y esto es algo que en determinadas aplicaciones no es una opción válida.

Generando un error 404, por ejemplo:

ENCUESTA/VIEWS.PY

```
from django.http import Http404
from django.shortcuts import render

```

```
4  from .models import Question
5  # ...
6  def detail(request, question_id):
7      try:
8          question = Question.objects.get(pk=question_id)
9      except Question.DoesNotExist:
10         raise Http404("La pregunta no existe")
11     return render(request, 'encuensta/detalle.html', {'question':
12 question})
```

El nuevo concepto aquí: la vista genera la excepción `Http404` si no existe una pregunta con la identificación solicitada.

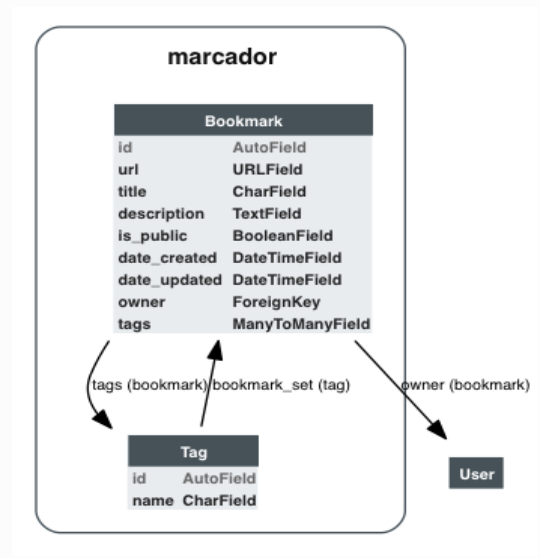
ENCUESTAS/TEMPLATES/ENCUESTAS/DETALLE.HTML

```
1  {{ question }}
```

```
1  raise ZeroDivisionError("Información de la excepción")
```

CREANDO MODELOS

Los modelos definen cómo se guardan los datos. Normalmente, un modelo representa una tabla en una base de datos, tiene campos, metadatos y métodos. Con esta información Django puede generar automáticamente una interfaz a la base de datos que permite el acceso orientado a objetos a ella. Esto se llama mapeo objeto-relacional (OMR).



S