دليل شامل حول Django Templates Engine وفلاتر Django

1. محرك قوالب Django (Django Templates Engine)

محرك قوالب Django هو نظام قوي ومصمم لإنشاء محتوى HTML ديناميكيًا. يعتمد هذا النظام على القوالب، وهي مستندات نصية أو سلاسل بايثون تحتوي على أجزاء ثابتة من مخرجات HTML المطلوبة، بالإضافة إلى بناء جملة خاص يصف كيفية إدراج المحتوى الديناميكي.

فلسفة محرك قوالب Django

تم تصميم لغة قوالب Django (DTL) لتحقيق توازن بين القوة وسهولة الاستخدام. إنها مصممة لتكون مريحة للمطورين المعتادين على العمل مع HTML. على عكس بعض أنظمة القوالب الأخرى، لا تهدف DTL إلى تضمين كود بايثون مباشر في HTML، بل تركز على فصل منطق العرض عن منطق الأعمال. هذا التصميم يضمن أن القوالب تُستخدم للعرض التقديمي فقط، وليس لمعالجة البيانات أو تنفيذ منطق برمجي معقد.

مكونات لغة قوالب Django

تتضمن لغة قوالب Django أربعة مكونات رئيسية:

أ. المتغيرات (Variables)

تُستخدم المتغيرات لعرض القيم من السياق (context)، وهو كائن يشبه القاموس (dict-like object) يربط المفاتيح بالقيم. تُحاط المتغيرات بعلامتي `{{` و `}}`.

مثال:

```
مرحباً، اسمي {{ user.first_name }} {{ user.last_name }}.
```

إذا كان السياق يحتوي على `{'user': {'first_name': 'جون', 'last_name': 'دو'}}`، فسيتم عرض:

`مرحباً، اسمي جون دو.`

يمكن الوصول إلى سمات الكائنات أو عناصر القواميس أو عناصر القوائم باستخدام نقطة (`.`).

مثال:

```
{{ my_dict.key }}
{{ my_object.attribute }}
{{ my_list.0 }}
```

ب. الوسوم (Tags)

توفر الوسوم منطقًا تعسفيًا في عملية العرض. تُحاط الوسوم بعلامتي `{%` و `%}` . يمكن للوسوم إخراج المحتوى، أو العمل كهياكل تحكم (مثل عبارات `if` أو حلقات `for`)، أو جلب معلومات من قاعدة البيانات، أو حتى تمكين الوصول إلى وسوم قوالب أخرى.

مثال:

```
{% if user.is_authenticated %}
    أهلاً بك، {{ user.username }}.
{% else %}
    الرجاء تسجيل الدخول.
{% endif %}

{% for item in item_list %}
    <li>{{ item.name }}</li>
{% endfor %}
```

ج. الفلاتر (Filters)

تقوم الفلاتر بتحويل قيم المتغيرات ووسيطات الوسوم للعرض. تُطبق الفلاتر باستخدام علامة الأنبوب (`|`). يمكن ربط الفلاتر معًا، حيث يتم تطبيق مخرجات فلتر واحد على الفلتر التالي.

مثال:

```
{{ my_text|lower }}
{{ my_date|date:"Y-m-d" }}
{{ value|filesizeformat }}
```

في المثال الأول، يحول الفلتر `lower` النص إلى أحرف صغيرة. في المثال الثاني، يقوم الفلتر `date` بتنسيق التاريخ. في المثال الثالث، يقوم الفلتر `filesizeformat` بتحويل حجم الملف إلى تنسيق سهل القراءة.

د. التعليقات (Comments)

تُستخدم التعليقات لإضافة ملاحظات داخل القالب دون أن يتم عرضها في المخرجات النهائية. تُحاط التعليقات بعلامتي `{#` و `#}` للتعليقات السطرية، أو `{% comment %}` و `{% endcomment %}` للتعليقات متعددة الأسطر.

مثال:

```
{# هذا التعليق لن يتم عرضه #}

{% comment "ملاحظة: هذا القسم قيد التطوير" %}
    <p>هذا النص لن يتم عرضه في الصفحة.</p>
{% endcomment %}
```

إعداد محرك القوالب

يتم تكوين محركات القوالب في ملف `settings.py` الخاص بمشروع Django، ضمن إعداد `TEMPLATES`. يمكن تحديد محركات قوالب متعددة، بما في ذلك محرك قوالب Django الافتراضي (DTL) و Jinja2.

مثال على الإعداد في `settings.py`:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

- `BACKEND`: يحدد مسار فئة محرك القوالب.
- `DIRS`: قائمة بالمسارات التي سيبحث فيها المحرك عن ملفات القوالب.
- `APP_DIRS`: إذا كانت `True`، سيبحث المحرك عن القوالب داخل دليل `templates` في كل تطبيق مثبت.

- `OPTIONS`: خيارات إضافية للمحرك، مثل `context_processors` التي تضيف بيانات مشتركة إلى السياق في كل طلب.

2. الفلاتر المتبقية في Django (The remaining filters for Django)

توفر Django مجموعة واسعة من الفلاتر المدمجة التي يمكن استخدامها لتعديل عرض المتغيرات في القوالب. فيما يلي شرح لبعض الفلاتر الشائعة والمفيدة:

الفلتر	الوصف	مثال
`add`	يضيف الوسيط إلى القيمة. يحاول تحويل كلا القيمتين إلى أعداد صحيحة. إذا فشل، فإنه يحاول إضافتهما معًا على أي حال (مثل السلاسل أو القوائم).	`{{ "value|add:"2 }}`
`addslashes`	يضيف شرطة مائلة عكسية قبل علامات الاقتباس. مفيد للهروب من السلاسل في CSV.	`{{ value|addslashes }}`
`capfirst`	يحول الحرف الأول من القيمة إلى حرف كبير.	`{{ value|capfirst }}`
`center`	يوسط القيمة في حقل بعرض معين.	`{{ "value|center:"15 }}`
`cut`	يزيل كل تكرارات الوسيط من السلسلة المحددة.	`{{ " ":value|cut }}`
`date`	ينسق كائن التاريخ أو التاريخ والوقت وفقًا لسلسلة تنسيق معينة.	`"value|date:"D d M Y }} {{`
`default`	إذا كانت القيمة خاطئة أو فارغة، فإنه يستخدم القيمة الافتراضية المحددة.	`{{ value|default:"لا شيء" }} {{`
`dictsort`	يأخذ قائمة من القواميس ويعيد القائمة مرتبة حسب المفتاح المحدد في الوسيط.	`"value|dictsort:"name }} {{`
`divisibleby`	يعيد `True` إذا كانت القيمة قابلة للقسمة على الوسيط.	`"value|divisibleby:"3 }} {{`
`escape`	يهرب من أحرف HTML الخاصة في سلسلة.	`{{ value|escape }}`
`filesizeformat`	ينسق القيمة كحجم ملف "يمكن قراءته من قبل الإنسان" (مثل `102 13`، `4.1 MB`، `KB`، `bytes`).	`value|filesizeformat }} {{`
`first`	يعيد العنصر الأول في القائمة.	`{{ value|first }}`
`floatformat`	يقرب الرقم العشري إلى عدد محدد من المنازل العشرية.	`value|floatformat:2 }} {{`
`join`	يربط قائمة بسلسلة، مثل `list.join()` في بايثون.	`{{ " ,":value|join }}`
`last`	يعيد العنصر الأخير في القائمة.	`{{ value|last }}`
`length`	يعيد طول القيمة. يعمل مع السلاسل والقوائم.	`{{ value|length }}`

الفلتر	الوصف	مثال
`linebreaks`	يستبدل فواصل الأسطر في النص العادي بعلامات HTML المناسبة (` ` و `<p>`).	`{{ value|linebreaks }}`
`lower`	يحول السلسلة إلى أحرف صغيرة.	`{{ value|lower }}`
`make_list`	يحول القيمة إلى قائمة. بالنسبة للسلسلة، إنها قائمة من الأحرف.	`{{ value|make_list }}`
`random`	يعيد عنصرًا عشوائيًا من القائمة المحددة.	`{{ value|random }}`
`safe`	يميز السلسلة بأنها "آمنة" من المزيد من الهروب من HTML.	`{{ value|safe }}`
`slice`	يعيد شريحة من القائمة.	`{{ "value|slice:":2 }}`
`slugify`	يحول إلى ASCII. يحول المسافات إلى واصلات. يزيل الأحرف التي ليست أبجدية رقمية أو شرطات سفلية أو واصلات.	`{{ value|slugify }}`
`stringformat`	ينسق المتغير وفقًا لوسيط التنسيق، على غرار مُحدد التنسيق `%` .	`{{ "value|stringformat:"03d }}`
`striptags`	يزيل جميع علامات HTML[X] من الإخراج.	`{{ value|striptags }}`
`time`	ينسق الوقت وفقًا لسلسلة التنسيق المحددة.	`{{ "value|time:"H:i }}`
`title`	يحول السلسلة إلى حالة العنوان (الحرف الأول من كل كلمة كبير).	`{{ value|title }}`
`truncatechars`	يقتطع سلسلة إذا كانت أطول من عدد الأحرف المحدد.	`{{ value|truncatechars:9 }}`
`truncatewords`	يقتطع سلسلة بعد عدد معين من الكلمات.	`{{ value|truncatewords:2 }}`
`upper`	يحول السلسلة إلى أحرف كبيرة.	`{{ value|upper }}`
`urlencode`	يهرب من قيمة للاستخدام في عنوان URL.	`{{ value|urlencode }}`
`wordcount`	يعيد عدد الكلمات في السلسلة.	`{{ value|wordcount }}`

الفلتر	الوصف	مثال
`yesno`	يعين القيم لـ `True` و `False` و (اختياريًا) `None`، إلى السلاسل "yes" و "no" و "maybe".	`{{ value|yesno:"نعم,لا,ربما" }}`

3. إضافة فلاتر جديدة (Add New filter)

يمكنك توسيع محرك قوالب Django عن طريق تعريف فلاتر مخصصة باستخدام بايثون، ثم إتاحتها لقوالبك باستخدام وسم `{% load %}`.

هيكل الكود (Code layout)

المكان الأكثر شيوعًا لتحديد فلاتر القوالب المخصصة هو داخل تطبيق Django. إذا كانت الفلاتر مرتبطة بتطبيق موجود، فمن المنطقي تجميعها هناك؛ وإلا، يمكن إضافتها إلى تطبيق جديد. عند إضافة تطبيق Django إلى `INSTALLED_APPS`، تصبح أي فلاتر يحددها في الموقع التقليدي الموضح أدناه متاحة تلقائيًا للتحميل داخل القوالب.

يجب أن يحتوي التطبيق على دليل `templatetags`، في نفس مستوى `views.py`، `models.py`، وما إلى ذلك. إذا لم يكن هذا الدليل موجودًا بالفعل، فقم بإنشائه - لا تنسَ ملف `__init__.py` لضمان التعامل مع الدليل كحزمة بايثون.

ملاحظة: بعد إضافة وحدة `templatetags`، ستحتاج إلى إعادة تشغيل الخادم قبل أن تتمكن من استخدام الفلاتر في القوالب.

ستعيش الفلاتر المخصصة الخاصة بك في وحدة داخل دليل `templatetags`. اسم ملف الوحدة هو الاسم الذي ستستخدمه لتحميل الفلاتر لاحقًا، لذا كن حذرًا في اختيار اسم لا يتعارض مع الفلاتر المخصصة في تطبيق آخر.

مثال على هيكل التطبيق:

```
polls/
    __init__.py
    models.py
    templatetags/
        __init__.py
        poll_extras.py
    views.py
```

وفي قالبك، ستستخدم ما يلي:

```
{% load poll_extras %}
```

يجب أن يكون التطبيق الذي يحتوي على الفلاتر المخصصة موجودًا في `INSTALLED_APPS` لكي يعمل وسم `{% load %}`.

يجب أن تحتوي الوحدة على متغير على مستوى الوحدة يسمى `register` وهو مثيل `template.Library`، حيث يتم تسجيل جميع الفلاتر. لذا، بالقرب من أعلى الوحدة النمطية الخاصة بك، ضع ما يلي:

```
from django import template

register = template.Library()
```

كتابة فلاتر القوالب المخصصة (Writing custom template filters)

الفلاتر المخصصة هي دوال بايثون تأخذ وسيطًا واحدًا أو وسيطين:

- قيمة المتغير (الإدخال) - ليس بالضرورة سلسلة.
- قيمة الوسيط - يمكن أن يكون لها قيمة افتراضية، أو يمكن حذفها تمامًا.

مثال: في الفلتر `{{ "var|foo:"bar" }}`، سيتم تمرير المتغير `var` والوسيط `"bar"` إلى الفلتر `foo`.

نظرًا لأن لغة القوالب لا توفر معالجة الاستثناءات، فإن أي استثناء يتم رفعه من فلتر القالب سيظهر كخطأ في الخادم. وبالتالي، يجب أن تتجنب دوال الفلتر رفع الاستثناءات إذا كانت هناك قيمة احتياطية معقولة لإرجاعها.

مثال على تعريف فلتر:

```
def cut(value, arg):
    """يزيل جميع قيم الوسيط من السلسلة المحددة"""
    return value.replace(arg, "")
```

مثال على استخدام هذا الفلتر:

```
{{ somevariable|cut:"0" }}
```

معظم الفلاتر لا تأخذ وسيطات. في هذه الحالة، اترك الوسيط خارج دالتك:

```
def lower(value):  # وسيط واحد فقط.
    """يحول السلسلة إلى أحرف صغيرة"""
    return value.lower()
```

تسجيل الفلاتر المخصصة (Registering custom filters)

بمجرد كتابة تعريف الفلتر الخاص بك، تحتاج إلى تسجيله باستخدام مثيل `Library` الخاص بك، لجعله متاحًا للغة قوالب Django:

```
register.filter("cut", cut)
register.filter("lower", lower)
```

تأخذ الدالة `Library.filter()` وسيطين:

1. اسم الفلتر - سلسلة.
2. دالة التجميع - دالة بايثون (وليس اسم الدالة كسلسلة).

يمكنك استخدام `register.filter()` كـ decorator بدلاً من ذلك:

```
@register.filter(name="cut")
def cut(value, arg):
    return value.replace(arg, "")

@register.filter
def lower(value):
    return value.lower()
```

إذا حذفت وسيط `name`، كما في المثال الثاني أعلاه، فسيستخدم Django اسم الدالة كاسم للفلتر.

فلاتر القوالب التي تتوقع سلاسل (Template filters that expect strings)

إذا كنت تكتب فلتر قالب يتوقع سلسلة فقط كوسيط أول، فيجب عليك استخدام decorator `stringfilter`. سيؤدي هذا إلى تحويل الكائن إلى قيمته السلسلة قبل تمريره إلى دالتك:

```
 from django import template
from django.template.defaultfilters import stringfilter

register = template.Library()

@register.filter
@stringfilter
def lower(value):
    return value.lower()
```

بهذه الطريقة، ستتمكن من تمرير، على سبيل المثال، عدد صحيح إلى هذا الفلتر، ولن يتسبب ذلك في حدوث `AttributeError` (لأن الأعداد الصحيحة لا تحتوي على دوال `lower()`).

الفلاتر والهروب التلقائي (Filters and auto-escaping)

عند كتابة فلتر مخصص، فكر في كيفية تفاعل الفلتر مع سلوك الهروب التلقائي في Django. لاحظ أن نوعين من السلاسل يمكن تمريرهما داخل كود القالب:

- **السلاسل الخام (Raw strings):** هي سلاسل بايثون الأصلية. عند الإخراج، يتم هروبها إذا كان الهروب التلقائي ساري المفعول ويتم تقديمها دون تغيير بخلاف ذلك.
- **السلاسل الآمنة (Safe strings):** هي سلاسل تم تمييزها على أنها آمنة من المزيد من الهروب عند وقت الإخراج. تم بالفعل إجراء أي هروب ضروري. تُستخدم عادةً للمخرجات التي تحتوي على HTML خام يُقصد تفسيره كما هو على جانب العميل.

إذا كان الفلتر الخاص بك لا يُدخل أي أحرف غير آمنة لـ HTML (`<`، `>`، `'`، `"` أو `&`) في النتيجة لم تكن موجودة بالفعل، فيمكنك السماح لـ Django بالتعامل مع جميع معالجة الهروب التلقائي لك. كل ما عليك فعله هو تعيين علامة `is_safe` إلى `True` عند تسجيل الفلتر الخاص بك:

```
 @register.filter(is_safe=True)
def myfilter(value):
    return value
```

تخبر هذه العلامة Django أنه إذا تم تمرير سلسلة "آمنة" إلى الفلتر الخاص بك، فستظل النتيجة "آمنة"، وإذا تم تمرير سلسلة غير آمنة، فسيقوم Django بتهريبها تلقائيًا، إذا لزم الأمر.