

django

template engine

template engine

Quando ci si avvicina a django è facile credere che template e file html siano la stessa cosa. In realtà un Template è un oggetto django che fa parte di una complessa struttura interna che si occupa di analizzare, elaborare, generare un context, gestire le django tag, popolare il codice dinamico, fino alla generazione di una lunga stringa rappresentante il contenuto HTML da mostrare su browser.

Tutto questo fa parte del **template engine** di django.

Un progetto django può essere configurato con uno o più *template engine* (o anche nessuno se il progetto non usa template). Django offre backend integrati per il proprio sistema di template, **django template language** (DTL) e per la popolare alternativa **Jinja2**. Il backend per altri *template language* può essere integrato da librerie esterne.

Django definisce un'API standard per il caricamento e il rendering dei template indipendentemente dal backend.

Il **caricamento** (*loading*) consiste nel trovare il template che fa riferimento ad un determinato identificatore e pre-elaborarlo (preprocessing), solitamente compilandolo in una rappresentazione in memoria.

Rendering significa interpolare il template con il context e ritornare la stringa risultante.

template engine

I template engine vengono configurati in settings.**TEMPLATES**. Questa è una lista di configurazioni così strutturata:

```
TEMPLATES = [  
    {  
        'BACKEND': 'django.template.backends.django.DjangoTemplates',  
        'DIRS': [],  
        'APP_DIRS': True,  
        'OPTIONS': {  
            # opzioni  
        }  
    }  
]
```



template engine

BACKEND è una path Python ad una classe del template engine di django che implementa le API. I backend integrati sono:

`django.template.backends.django.DjangoTemplates`

`django.template.backends.jinja2.Jinja2`

Dato che molti engine caricano template da file, le opzioni di configurazione per ciascun engine contengono due setting comuni:

- **DIRS** definisce una lista di directory dove l'engine deve cercare i file di origine dei template, definiti in ordine di ricerca.
- **APP_DIRS** definisce se l'engine deve cercare template dentro alle app installate. Ogni backend definisce un nome convenzionale per la sub-directory all'interno delle applicazioni dove i loro template vengono memorizzati.

OPTIONS contiene dei setting specifici per il backend.



template engine

Il modulo **django.template.loader** definisce due funzioni per caricare template.

- **get_template(template_name, using=None)** → questa funzione carica un template e ritorna un oggetto **Template**. L'esatto tipo del valore di ritorno dipende dal backend che carica il template. Ogni backend ha una sua classe Template.

`get_template()` prova ogni template engine nell'ordine dato finché uno non ha successo.
Se si vuole restringere la ricerca a un particolare template engine, si assegna il valore di NAME del template engine al parametro `using`.

- **select_template(template_name_list, using=None)** → il suo funzionamento è analogo a quello di `get_template()` salvo che accetta una lista di nomi di template.

Se il template non può essere trovato, viene sollevata un'eccezione **TemplateDoesNotExist**.

Se il template viene trovato ma contiene una sintassi non valida, viene sollevata **TemplateSyntaxError**.



template engine

django offre inoltre una funzione che automatizza il processo di renderizzazione dei template.

render_to_string(template_name, context=None, request=None, using=None) → carica un template (come `get_template()`) e chiama il suo metodo **render()** immediatamente.

- **template_name** → il nome del template da caricare e renderizzare. Se è una lista di template, django utilizza `select_template()` anziché `get_template()` per trovare i template.
- **context** → un dizionario usato come template context per il rendering.
- **request** → una **HttpRequest** opzionale che è disponibile durante il processo di rendering del template.



La shortcut **render()** (già affrontata nella prima parte del corso) è una funzione che chiama `render_to_string()` e inserisce il risultato in una `HttpResponse` adatta per essere ritornata da una view.



template language

template language

Come abbiamo già avuto modo di vedere, django offre una serie di strumenti che possono essere usati all'interno dei template per rappresentare dati. I template possono contenere **variable** (variabili), che vengono sostituite con valori quando il template viene elaborato, e **tag**, che controllano la logica del template.



variable → utilizzano la seguente sintassi: `{{ variable }}`.

Quando il template engine incontra una variabile, questa viene tradotta e rimpiazzata con il suo valore.

Il nome di una variabile può contenere qualsiasi combinazione di caratteri alfanumerici e underscore ma **NON** può cominciare con un underscore.

Il punto (.) viene usato per accedere agli attributi di una variabile, es: `{{ variable.attribute }}`.



template language

Quando il template engine incontra un punto nel nome di una variabile, prova le seguenti ricerche in quest'ordine:

- Cerca un dizionario;
- Cerca un attributo o un metodo;
- Cerca un indice numerico.



Se il valore risultante è *callable*, viene richiamato senza argument. Il risultato della chiamata diventa il valore sul template.



Se si utilizza una variabile non esistente, il template system inserisce il valore dell'opzione `string_if_invalid`, che è una stringa vuota di default.



template language

È possibile modificare il valore mostrato di una variabile utilizzando i **template filters**. 

I filtri utilizzano la seguente sintassi: `{{ name|lower }}`

Questo mostra il valore di name dopo essere filtrato con il filtro lower che converte i caratteri di una stringa in lowercase.


I filtri possono essere concatenati. L'output di un filtro è passato al successivo:

`{{ variable|filter|another_filter }}`

Alcuni filtri accettano degli argument: `{{ text|truncatewords:30 }}`

Questo mostrerà solo le prime 30 parole di text.

Gli argument che accettano spazi devono essere inseriti in double-quotes ("") o quotes (' ').

A small, stylized illustration of a green and blue rocket ship with a flame at the bottom, positioned to the left of the text box.

Per un elenco completo dei built-in filter di django, visita:

<https://docs.djangoproject.com/en/2.2/ref/templates/builtins/#ref-templates-builtins-filters>

template language

tag → utilizzano la seguente sintassi: `{% tag %}`.

I tag svolgono varie funzioni. Alcuni generano un output testuale, altri controllano il flow con loop e operatori condizionali, altri ancora caricano informazioni esterne all'interno del template.

Alcuni tag richiedono una tag di apertura e una di chiusura, es: `{% tag %} [...] {% endtag %}`

Tra i più comuni ed utilizzati, ricordiamo:

- `{% for %}` → esegue un loop su un oggetto iterabile

es:

```
{% for element in elements_list %}  
    {{ element.detail }} <br>  
{% endfor %}
```



template language

- `{% if %}`, `{% elif %}`, `{% else %}` → esegue il blocco di codice se la condizione è vera

es:

```
{% if elements_list %}
```

```
    Numero di elementi: {{ elements_list|length }}
```

```
{% elif elements_not_listed %}
```

```
    Numero di elementi fuori dalla lista: {{ elements_not_listed|length }}
```

```
{% else %}
```

```
    Non è presente nessun elemento.
```

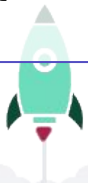
```
{% endif %}
```

Se `element_list` è vuota, viene valutata la condizione di `{% elif %}`, e così via.



template language

- **{% extends %}** → permette ad un template di ereditare il contenuto di un altro. Accetta un argument sotto forma di stringa che fa riferimento al template da “estendere”.
- **{% block %}** → permette di definire un contenuto che i template “figli” (i template che ereditano) potranno personalizzare. Quello che fa un tag **{% block %}** è dire al template engine che la sezione al suo interno è sovrascrivibile.
Accettano un parametro che serve come riferimento ai template che ereditano per definire il contenuto.
- **{# comment #}** / **{% comment %}** **{% endcomment %}** → django offre anche la possibilità di commentare una porzione di codice sia con la specifica sintassi **{# . . . #}** che con l’apposito tag **{% comment %}**



views.py

views.py

Fino ad ora abbiamo creato, modificato e cancellato oggetti nelle tabelle del database attraverso l'interfaccia di amministrazione di django. Com'è ovvio, è necessario fornire anche ad un utente non amministratore la possibilità di eseguire queste azioni.

Per modificare i dati su database attraverso view, django offre i seguenti strumenti attraverso il modulo `django.views.generic.edit`:

- **FormView** → una view che mostra un form. Se si presenta un errore nella compilazione, mostra nuovamente il form con gli errori di validazione. Se compilato con successo, reindirizza ad un nuovo url [essendo una classe strettamente legata al tema dei django form, verrà affrontata successivamente].
- **CreateView** → una view che mostra un form per creare un oggetto. Mostra nuovamente il form con errori di validazione (se presenti) e salva l'oggetto.
- **Updateview** → una view che mostra un form per modificare un oggetto esistente. Mostra nuovamente il form in caso di errori di validazione e salva i cambiamenti dell'oggetto.
- **DeleteView** → una view che mostra una pagina di conferma e cancella un oggetto esistente.



views.py

CreateView si utilizza per inserire nuove entry in una specifica tabella del database. Necessita di una serie di attributi, due dei quali abbiamo già incontrato in `DetailView` e `ListView`:

- **model** → il model al quale si vuole aggiungere un oggetto;
- **template_name** → il template utilizzato per renderizzare il contenuto.

Utilizza inoltre:

- **fields** → i campi da compilare per inserire una nuova entry su database, elencati in una lista o tupla;
- **success_url** → l'url al quale si viene reindirizzati a salvataggio avvenuto.



views.py

Nel file `views.py` all'interno dell'app `soci`, inserisci il codice che segue:

```
from django.urls import reverse_lazy
from django.views.generic.edit import CreateView

class PersonaCreate(CreateView):

    model = Persona
    fields = ('nome', 'cognome', 'ruolo')
    template_name = 'soci/persona_create.html'
    success_url = reverse_lazy('soci:persona-list')
```

`reverse()` e **`reverse_lazy()`** sono funzioni importate da `django.urls` e vengono utilizzate per usare una sintassi simile a quella dei tag url dei template di django, facendo riferimento all'argument `name` assegnato alla path.

Per ora, ci basti sapere che `reverse()` si utilizza con le function-based views, mentre `reverse_lazy()` si utilizza con le class-based views

`path()` da aggiungere a `soci/urls.py`

```
path('persona-create', views.PersonaCreate.as_view(), name='persona-create'),
```

views.py

In `soci/templates/soci` crea il file `persona_create.html` e aggiungi al `<body>` il seguente codice:

```
<form method="post">{% csrf_token %}  
  {{ form.as_p }}  
  <input type="submit" value="Save">  
</form>
```

`{% csrf_token %}` (Cross Site Request Forgery) è uno strumento utilizzato come protezione dagli attacchi csrf. È obbligatorio usarlo nei form con metodo POST.

Per renderizzare il form, si passa l'oggetto `form` con la sintassi delle template variables. django permette di renderizzare i form con i seguenti metodi:

- **as_p** → ogni `<input>` viene inserito in un `<p>`;
- **as_table** → ogni `<input>` viene inserito in una cella di tabella all'interno di un `<tr>`;
- **as_ul** → ogni `<input>` viene inserito in una tag `` all'interno di una ``.

* è necessario inserire nel file html `<table>` o ``

views.py

UpdateView permette di modificare i valori degli attributi di un oggetto già presente su una tabella del database.

DeleteView permette di eliminare un'entry da una specifica tabella.

Entrambe utilizzano un codice per la view e una path identici a quello di CreateView, salvo per il fatto che DeleteView non deve specificare l'attributo `fields` ed entrambe le classi, dovendo modificare un'entry specifica della tabella, useranno un url contenente un identificativo per accedere all'oggetto specifico, come a esempio `<int:pk>`.

```
path('persona-update/<int:pk>', views.PersonaUpdate.as_view(), name='persona-update'),  
path('persona-delete/<int:pk>', views.PersonaDelete.as_view(), name='persona-delete')
```



django

francesco.faenza@unimore.it