Templates

Being a web framework, Django needs a convenient way to generate HTML dynamically. The most common approach relies on templates. A template contains the static parts of the desired HTML output as well as some special syntax describing how dynamic content will be inserted. For a hands-on example of creating HTML pages with templates, see Tutorial 3.

A Django project can be configured with one or several template engines (or even zero if you don't use templates). Django ships built-in backends for its own template system, creatively called the Django template language (DTL), and for the popular alternative Jinja2. Backends for other template languages may be available from third-parties. You can also write your own custom backend, see Custom template backend

Django defines a standard API for loading and rendering templates regardless of the backend. Loading consists of finding the template for a given identifier and preprocessing it, usually compiling it to an in-memory representation. Rendering means interpolating the template with context data and returning the resulting string.

The Django template language is Django's own template system. Until Django 1.8 it was the only built-in option available. It's a good template library even though it's fairly opinionated and sports a few idiosyncrasies. If you don't have a pressing reason to choose another backend, you should use the DTL, especially if you're writing a pluggable application and you intend to distribute templates. Django's contrib apps that include templates, like django.contrib.admin, use the DTL.

For historical reasons, both the generic support for template engines and the implementation of the Django template language live in the `django.template` namespace.

Warning: The template system isn't safe against untrusted template authors. For example, a site shouldn't allow its users to provide their own templates, since template authors can do things like perform XSS attacks and access properties of template variables that may contain sensitive information.

The Django template language

Syntax

About this section: This is an overview of the Django template language's syntax. For details see the language syntax reference.

A Django template is a text document or a Python string marked-up using the Django template language. Some constructs are recognized and interpreted by the template engine. The main ones are variables and tags.

A template is rendered with a context. Rendering replaces variables with their values, which are looked up in the context, and executes tags. Everything else is output as is.

The syntax of the Django template language involves four constructs.

Variables

A variable outputs a value from the context, which is a dict-like object mapping keys to values.

Variables are surrounded by `{{` and `}}` like this:

```
My first name is {{ first_name }}. My last name is {{ last_name }}.
```

With a context of `{'first_name': 'John', 'last_name': 'Doe'}`, this template renders to:

```
My first name is John. My last name is Doe.
```

Dictionary lookup, attribute lookup and list-index lookups are implemented with a dot notation:

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

If a variable resolves to a callable, the template system will call it with no arguments and use its result instead of the callable.

Tags

Tags provide arbitrary logic in the rendering process.

This definition is deliberately vague. For example, a tag can output content, serve as a control structure e.g. an “if” statement or a “for” loop, grab content from a database, or even enable access to other template tags.

Tags are surrounded by `{%` and `%}` like this:

```
{% csrf_token %}
```

Most tags accept arguments:

```
{% cycle 'odd' 'even' %}
```

Some tags require beginning and ending tags:

```
{% if user.is_authenticated %}Hello, {{ user.username }}.{% endif %}
```

A reference of built-in tags is available as well as instructions for writing custom tags.

Filters

Filters transform the values of variables and tag arguments.

They look like this:

```
{{ django|title }}
```

With a context of `{'django': 'the web framework for perfectionists with deadlines'}`, this template renders to:

```
The Web Framework For Perfectionists With Deadlines
```

Some filters take an argument:

```
{{ my_date|date:"Y-m-d" }}
```

A reference of built-in filters is available as well as instructions for writing custom filters.

Comments

Comments look like this:

```
{# this won't be rendered #}
```

A `{% comment %}` tag provides multi-line comments.

Components

About this section: This is an overview of the Django template language's APIs. For details see the API reference.

Engine

`django.template.Engine` encapsulates an instance of the Django template system. The main reason for instantiating an `Engine` directly is to use the Django template language outside of a Django project.

`django.template.backends.django.DjangoTemplates` is a thin wrapper adapting `django.template.Engine` to Django's template backend API.

Template

`django.template.Template` represents a compiled template. Templates are obtained with `Engine.get_template()` or `Engine.from_string()`.

Likewise `django.template.backends.django.Template` is a thin wrapper adapting `django.template.Template` to the common template API.

Context

`django.template.Context` holds some metadata in addition to the context data. It is passed to `Template.render()` for rendering a template.

`django.template.RequestContext` is a subclass of `Context` that stores the current `HttpRequest` and runs template context processors.

The common API doesn't have an equivalent concept. Context data is passed in a plain `dict` and the current `HttpRequest` is passed separately if needed.

Loaders

Template loaders are responsible for locating templates, loading them, and returning `Template` objects.

Django provides several built-in template loaders and supports custom template loaders.

Context processors

Context processors are functions that receive the current `HttpRequest` as an argument and return a `dict` of data to be added to the rendering context.

Their main use is to add common data shared by all templates to the context without repeating code in every view.

Django provides many built-in context processors, and you can implement your own additional context processors, too.

Support for template engines

Configuration

Templates engines are configured with the `TEMPLATES` setting. It's a list of configurations, one for each engine. The default value is empty. The `settings.py` generated by the `startproject` command defines a more useful value:

```
TEMPLATES = [
    {
        "BACKEND": "django.template.backends.django.DjangoTemplates",
        "DIRS": [],
        "APP_DIRS": True,
        "OPTIONS": {
            # ... some options here ...
        },
    },
]
```

`BACKEND` is a dotted Python path to a template engine class implementing Django's template backend API. The built-in backends are `django.template.backends.django.DjangoTemplates` and `django.template.backends.jinja2.Jinja2`.

Since most engines load templates from files, the top-level configuration for each engine contains two common settings:

- `DIRS` defines a list of directories where the engine should look for template source files, in search order.
- `APP_DIRS` tells whether the engine should look for templates inside installed applications. Each backend defines a conventional name for the subdirectory inside applications where its templates should be stored.

While uncommon, it's possible to configure several instances of the same backend with different options. In that case you should define a unique `NAME` for each engine.

`OPTIONS` contains backend-specific settings.

Usage

The `django.template.loader` module defines two functions to load templates.

`get_template(template_name, using=None)` [source]

This function loads the template with the given name and returns a `Template` object.

The exact type of the return value depends on the backend that loaded the template. Each backend has its own `Template` class.

`get_template()` tries each template engine in order until one succeeds. If the template cannot be found, it raises `TemplateDoesNotExist`. If the template is found but contains invalid syntax, it raises `TemplateSyntaxError`.

How templates are searched and loaded depends on each engine's backend and configuration.

If you want to restrict the search to a particular template engine, pass the engine's `NAME` in the `using` argument.

`select_template(template_name_list, using=None)` [source]

`select_template()` is just like `get_template()`, except it takes a list of template names. It tries each name in order and returns the first template that exists.

If loading a template fails, the following two exceptions, defined in `django.template`, may be raised:

`exception TemplateDoesNotExist(msg, tried=None, backend=None, chain=None)` [source]

This exception is raised when a template cannot be found. It accepts the following optional arguments for populating the template postmortem on the debug page:

`backend`

The template backend instance from which the exception originated.

`tried`

A list of sources that were tried when finding the template. This is formatted as a list of tuples containing `(origin, status)`, where `origin` is an origin-like object and `status` is a string with the reason the template wasn't found.

`chain`

A list of intermediate `TemplateDoesNotExist` exceptions raised when trying to load a template. This is used by functions, such as `get_template()`, that try to load a given template from multiple engines.

`exception TemplateSyntaxError(msg)` [source]

This exception is raised when a template was found but contains errors.

`Template` objects returned by `get_template()` and `select_template()` must provide a `render()` method with the following signature:

```
Template.render(context=None, request=None)
```

Renders this template with a given context.

If `context` is provided, it must be a `dict`. If it isn't provided, the engine will render the template with an empty context.

If `request` is provided, it must be an `HttpRequest`. Then the engine must make it, as well as the CSRF token, available in the template. How this is achieved is up to each backend.

Here's an example of the search algorithm. For this example the `TEMPLATES` setting is:

```
TEMPLATES = [
    {
        "BACKEND": "django.template.backends.django.DjangoTemplates",
        "DIRS": [
            "/home/html/example.com",
            "/home/html/default",
        ],
    },
    {
        "BACKEND": "django.template.backends.jinja2.Jinja2",
        "DIRS": [
            "/home/html/jinja2",
        ],
    },
]
```

If you call `get_template('story_detail.html')`, here are the files Django will look for, in order:

- `/home/html/example.com/story_detail.html` (`'django'` engine)
- `/home/html/default/story_detail.html` (`'django'` engine)
- `/home/html/jinja2/story_detail.html` (`'jinja2'` engine)

If you call `select_template(['story_253_detail.html', 'story_detail.html'])`, here's what Django will look for:

- `/home/html/example.com/story_253_detail.html` (`'django'` engine)
- `/home/html/default/story_253_detail.html` (`'django'` engine)
- `/home/html/jinja2/story_253_detail.html` (`'jinja2'` engine)
- `/home/html/example.com/story_detail.html` (`'django'` engine)
- `/home/html/default/story_detail.html` (`'django'` engine)
- `/home/html/jinja2/story_detail.html` (`'jinja2'` engine)

When Django finds a template that exists, it stops looking.

Use `django.template.loader.select_template()` for more flexibility: You can use `select_template()` for flexible template loading. For example, if you've written a news story and want some stories to have custom templates, use something like `select_template(['story_%s_detail.html' % story.id, 'story_detail.html'])`. That'll allow you to use a custom template for an individual story, with a fallback template for stories that don't have custom templates.

It's possible – and preferable – to organize templates in subdirectories inside each directory containing templates. The convention is to make a subdirectory for each Django app, with subdirectories within those subdirectories as needed.

Do this for your own sanity. Storing all templates in the root level of a single directory gets messy.

To load a template that's within a subdirectory, use a slash, like so:

```
get_template("news/story_detail.html")
```

Using the same `TEMPLATES` option as above, this will attempt to load the following templates:

- `/home/html/example.com/news/story_detail.html` (`'django'` engine)
- `/home/html/default/news/story_detail.html` (`'django'` engine)
- `/home/html/jinja2/news/story_detail.html` (`'jinja2'` engine)

In addition, to cut down on the repetitive nature of loading and rendering templates, Django provides a shortcut function which automates the process.

`render_to_string(template_name, context=None, request=None, using=None)` **[source]**

`render_to_string()` loads a template like `get_template()` and calls its `render()` method immediately. It takes the following arguments.

`template_name`

The name of the template to load and render. If it's a list of template names, Django uses `select_template()` instead of `get_template()` to find the template.

`context`

A `dict` to be used as the template's context for rendering.

`request`

An optional `HttpRequest` that will be available during the template's rendering process.

`using`

An optional template engine `NAME`. The search for the template will be restricted to that engine.

Usage example:

```
from django.template.loader import render_to_string

rendered = render_to_string("my_template.html", {"foo": "bar"})
```

See also the `render()` shortcut which calls `render_to_string()` and feeds the result into an `HttpResponse` suitable for returning from a view.

Finally, you can use configured engines directly:

`engines`

Template engines are available in `django.template.engines`:

```
from django.template import engines

django_engine = engines["django"]
template = django_engine.from_string("Hello {{ name }}!")
```

The lookup key — `'django'` in this example — is the engine's `NAME`.

Built-in backends

`class DjangoTemplates` [source]

Set `BACKEND` to `'django.template.backends.django.DjangoTemplates'` to configure a Django template engine.

When `APP_DIRS` is `True`, `DjangoTemplates` engines look for templates in the `templates` subdirectory of installed applications. This generic name was kept for backwards-compatibility.

`DjangoTemplates` engines accept the following `OPTIONS`:

- `'autoescape'`: a boolean that controls whether HTML autoescaping is enabled.

  It defaults to `True`.

  > **Warning:** Only set it to `False` if you're rendering non-HTML templates!

- `'context_processors'`: a list of dotted Python paths to callables that are used to populate the context when a template is rendered with a request. These callables take a request object as their argument and return a `dict` of items to be merged into the context.

  It defaults to an empty list.

  See `RequestContext` for more information.

- `'debug'` : a boolean that turns on/off template debug mode. If it is `True` , the fancy error page will display a detailed report for any exception raised during template rendering. This report contains the relevant snippet of the template with the appropriate line highlighted.

  It defaults to the value of the `DEBUG` setting.

- `'loaders'` : a list of dotted Python paths to template loader classes. Each `Loader` class knows how to import templates from a particular source. Optionally, a tuple can be used instead of a string. The first item in the tuple should be the `Loader` class name, and subsequent items are passed to the `Loader` during initialization.

  The default depends on the values of `DIRS` and `APP_DIRS`.

  See Loader types for details.

- `'string_if_invalid'` : the output, as a string, that the template system should use for invalid (e.g. misspelled) variables.

  It defaults to an empty string.

  See How invalid variables are handled for details.

- `'file_charset'` : the charset used to read template files on disk.

  It defaults to `'utf-8'` .

- `'libraries'` : A dictionary of labels and dotted Python paths of template tag modules to register with the template engine. This can be used to add new libraries or provide alternate labels for existing ones. For example:

  ```
  OPTIONS = {
      "libraries": {
          "myapp_tags": "path.to.myapp.tags",
          "admin.urls": "django.contrib.admin.templatetags.admin_urls",
      },
  }
  ```

  Libraries can be loaded by passing the corresponding dictionary key to the `{% load %}` tag.

- `'builtins'` : A list of dotted Python paths of template tag modules to add to built-ins. For example:

  ```
  OPTIONS = {
      "builtins": ["myapp.builtins"],
  }
  ```

  Tags and filters from built-in libraries can be used without first calling the `{% load %}` tag.

`class Jinja2` **[source]**

Requires Jinja2 to be installed:

```
$ python -m pip install Jinja2
```

```
...\> py -m pip install Jinja2
```

Set `BACKEND` to `'django.template.backends.jinja2.Jinja2'` to configure a Jinja2 engine.

When `APP_DIRS` is `True`, `Jinja2` engines look for templates in the `jinja2` subdirectory of installed applications.

The most important entry in `OPTIONS` is `'environment'`. It's a dotted Python path to a callable returning a Jinja2 environment. It defaults to `'jinja2.Environment'`. Django invokes that callable and passes other options as keyword arguments. Furthermore, Django adds defaults that differ from Jinja2's for a few options:

- `'autoescape'`: `True`
- `'loader'`: a loader configured for `DIRS` and `APP_DIRS`
- `'auto_reload'`: `settings.DEBUG`
- `'undefined'`: `DebugUndefined if settings.DEBUG else Undefined`

`Jinja2` engines also accept the following `OPTIONS`:

- `'context_processors'`: a list of dotted Python paths to callables that are used to populate the context when a template is rendered with a request. These callables take a request object as their argument and return a `dict` of items to be merged into the context.

  It defaults to an empty list.

  > **Using context processors with Jinja2 templates is discouraged.:** Context processors are useful with Django templates because Django templates don't support calling functions with arguments. Since Jinja2 doesn't have that limitation, it's recommended to put the function that you would use as a context processor in the global variables available to the template using `jinja2.Environment` as described below. You can then call that function in the template:
  >
  > ```
  > {{ function(request) }}
  > ```
  >
  > Some Django templates context processors return a fixed value. For Jinja2 templates, this layer of indirection isn't necessary since you can add constants directly in `jinja2.Environment`.
  >
  > The original use case for adding context processors for Jinja2 involved:
  >
  > - Making an expensive computation that depends on the request.
  > - Needing the result in every template.
  > - Using the result multiple times in each template.
  >
  > Unless all of these conditions are met, passing a function to the template is more in line with the design of Jinja2.

The default configuration is purposefully kept to a minimum. If a template is rendered with a request (e.g. when using `render()`), the `Jinja2` backend adds the globals `request`, `csrf_input`, and `csrf_token` to the context. Apart from that, this backend doesn't create a Django-flavored environment. It doesn't know about Django filters and tags. In order to use Django-specific APIs, you must configure them into the environment.

For example, you can create `myproject/jinja2.py` with this content:

```
from django.templatetags.static import static
from django.urls import reverse

from jinja2 import Environment


def environment(**options):
    env = Environment(**options)
    env.globals.update(
        {
            "static": static,
            "url": reverse,
        }
    )
    return env
```

and set the `'environment'` option to `'myproject.jinja2.environment'`.

Then you could use the following constructs in Jinja2 templates:

```
<img src="{{ static('path/to/company-logo.png') }}" alt="Company Logo">

<a href="{{ url('admin:index') }}">Administration</a>
```

The concepts of tags and filters exist both in the Django template language and in Jinja2 but they're used differently. Since Jinja2 supports passing arguments to callables in templates, many features that require a template tag or filter in Django templates can be achieved by calling a function in Jinja2 templates, as shown in the example above. Jinja2's global namespace removes the need for template context processors. The Django template language doesn't have an equivalent of Jinja2 tests.