# Templates

Flask leverages Jinja2 as template engine. You are obviously free to use a different template engine, but you still have to install Jinja2 to run Flask itself. This requirement is necessary to enable rich extensions. An extension can depend on Jinja2 being present.

This section only gives a very quick introduction into how Jinja2 is integrated into Flask. If you want information on the template engine's syntax itself, head over to the official Jinja2 Template Documentation for more information.

## Jinja Setup

Unless customized, Jinja2 is configured by Flask as follows:

- autoescaping is enabled for all templates ending in `.html`, `.htm`, `.xml` as well as `.xhtml` when using **`render_template()`**.
- autoescaping is enabled for all strings when using **`render_template_string()`**.
- a template has the ability to opt in/out autoescaping with the `{% autoescape %}` tag.
- Flask inserts a couple of global functions and helpers into the Jinja2 context, additionally to the values that are present by default.

## Standard Context

The following global variables are available within Jinja2 templates by default:

### `config`

The current configuration object (**`flask.config`**)

▶ *Changelog*

### `request`

The current request object (**`flask.request`**). This variable is unavailable if the template was rendered without an active request context.

### `session`

The current session object (**`flask.session`**). This variable is unavailable if the template was rendered without an active request context.

### `g`

The request-bound object for global variables (**`flask.g`**). This variable is unavailable if the template was rendered without an active request context.

### `url_for`()

The **`flask.url_for()`** function.

### `get_flashed_messages`()

The **`flask.get_flashed_messages()`** function.

## The Jinja Context Behavior:

These variables are added to the context of variables, they are not global variables. The difference is that by default these will not show up in the context of imported templates. This is partially caused by performance considerations, partially to keep things explicit.

What does this mean for you? If you have a macro you want to import, that needs to access the request object you have two possibilities:

1. you explicitly pass the request to the macro as parameter, or the attribute of the request object you are interested in.
2. you import the macro "with context".

Importing with context looks like this:

```
{% from '_helpers.html' import my_macro with context %}
```

# Standard Filters

These filters are available in Jinja2 additionally to the filters provided by Jinja2 itself:

### `tojson`()

This function converts the given object into JSON representation. This is for example very helpful if you try to generate JavaScript on the fly.

```
<script type=text/javascript>
    doSomethingWith({{ user.username|tojson }});
</script>
```

It is also safe to use the output of |*tojson* in a *single-quoted* HTML attribute:

```
<button onclick='doSomethingWith({{ user.username|tojson }})'>
    Click me
</button>
```

Note that in versions of Flask prior to 0.10, if using the output of `|tojson` inside `script`, make sure to disable escaping with `|safe`. In Flask 0.10 and above, this happens automatically.

# Controlling Autoescaping

Autoescaping is the concept of automatically escaping special characters for you. Special characters in the sense of HTML (or XML, and thus XHTML) are `&`, `>`, `<`, `"` as well as `'`. Because these characters carry specific meanings in documents on their own you have to replace them by so called "entities" if you want to use them for text. Not doing so would not only cause user frustration by the inability to use these characters in text, but can also lead to security problems. (see Cross-Site Scripting (XSS))

Sometimes however you will need to disable autoescaping in templates. This can be the case if you want to explicitly inject HTML into pages, for example if they come from a system that generates secure HTML like a markdown to HTML converter.

There are three ways to accomplish that:

- In the Python code, wrap the HTML string in a **`Markup`** object before passing it to the template. This is in general the recommended way.
- Inside the template, use the `|safe` filter to explicitly mark a string as safe HTML (`{{ myvariable|safe }}`)
- Temporarily disable the autoescape system altogether.

To disable the autoescape system in templates, you can use the `{% autoescape %}` block:

```
{% autoescape false %}
    <p>autoescaping is disabled here
    <p>{{ will_not_be_escaped }}
{% endautoescape %}
```

Whenever you do this, please be very cautious about the variables you are using in this block.

# Registering Filters

If you want to register your own filters in Jinja2 you have two ways to do that. You can either put them by hand into the **`jinja_env`** of the application or use the **`template_filter()`** decorator.

The two following examples work the same and both reverse an object:

```
@app.template_filter('reverse')
def reverse_filter(s):
    return s[::-1]

def reverse_filter(s):
    return s[::-1]
app.jinja_env.filters['reverse'] = reverse_filter
```

In case of the decorator the argument is optional if you want to use the function name as name of the filter. Once registered, you can use the filter in your templates in the same way as Jinja2's builtin filters, for example if you have a Python list in context called *mylist*:

```
{% for x in mylist | reverse %}
{% endfor %}
```

# Context Processors

To inject new variables automatically into the context of a template, context processors exist in Flask. Context processors run before the template is rendered and have the ability to inject new values into the template context. A context processor is a function that returns a dictionary. The keys and values of this dictionary are then merged with the template context, for all templates in the app:

```
@app.context_processor
def inject_user():
    return dict(user=g.user)
```

The context processor above makes a variable called *user* available in the template with the value of *g.user*. This example is not very interesting because *g* is available in templates anyways, but it gives an idea how this works.

Variables are not limited to values; a context processor can also make functions available to templates (since Python allows passing around functions):

```
@app.context_processor
def utility_processor():
    def format_price(amount, currency=u'€'):
        return u'{0:.2f}{1}'.format(amount, currency)
    return dict(format_price=format_price)
```

The context processor above makes the *format_price* function available to all templates:

```
{{ format_price(0.33) }}
```

You could also build *format_price* as a template filter (see Registering Filters), but this demonstrates how to pass functions in a context processor.