codecademy

# Jinja2 Templates and Forms

## Flask Templates

Flask uses templates to expand the functionality of a web application while maintaining a simple and organized file structure. Templates are enabled using the Jinja2 template engine and allow data to be shared and processed before being turned in to content and sent back to the client.

## `render_template` Function

The `render_template()` function renders HTML files for display in the web browser using the Jinja2 template engine. Its sole positional argument is the name of the template to be rendered. Keyword arguments can be added to support application variables within the template.

where `render_template` processes template files in the "templates" directory. The "templates" directory located inside the running application directory.

├── app_dir │ ├── templates │ │ ├── my_template.html │ ├── app.py

## Template Variables

Template variables are representations of application data that are used inside templates. Created using keyword arguments in `render_template`, template variables are substituted into a template when using expression delimiters `{{ }}`

Used in Jinja templates, expression delimiters `{{ }}` surround expressions such as variables to be identified by the template engine.

```
return render_template('my_template.html',
template_var1=flask_var1,
template_var2=flask_var2)

## The template variables are substituted
into "my_template.html" using the
following format
## {{ template_var1 }} and {{
template_var2 }}
```

## Template Variable Filters

Filters are applied to template variables and perform a small focused action. Filters follow a variable and are separated by the pipe, `|`, character. This syntax works inside expression `{{ }}` and statement `{% %}` delimiters.
More information on filters can be found in the Jinja documentation.

```
<!-- Capitalize the string -->
<h1>{{ heading_var | upper  }}</h1>

<!-- Default string when
no_var not declared -->
<h1>{{ no_var | default("My Website") }}
</h1>
```

## Template `if` Statements

`{% if condition %}` statements are used in templates to control the content that is returned from Flask routes. Content associated with an `if` statement is enclosed in blocks starting with `{% if condition %}` and closing with `{% endif %}`. `else` and `elif` can be used with `if` statements to further control the content.
Statement delimiters `{% %}` surround control statements such as `if` and `for` to be identified by the Jinja 2 template engine.

```
<html>
  <body>
    <h1>
      {% if template_feel == 'happy' %}
        This Page Is Happy
      {% elif template_feel == 'sad' %}
        This Page Is Sad
      {% else %}
        This Page Is Not Sure How It Feels
      {% endfor %}
    </h1>
  </body>
</html>
```

## Template `for` Loops

For loops are used in templates to repeat content or iterate through data during Jinja2 template engine processing. For loops are contained inside statement delimiters with the following syntax {% for local_var in iterator %}. Content inside the block is repeated where the block start with the `for` statement and is closed by {% endfor %}.
Statement delimiters `{% %}` surround control statements such as `if` and `for` to be identified by the Jinja 2 template engine.

```
<ul>
  {% for content_item in content_list %}
    <li>content_item</li>
  {% endfor %}
</ul>
```

## Template Inheritance

Jinja templates use inheritance to share content from one template across multiple templates.
The `block` statement is used to identify the location in each file to establish inheritance. In the parent file `{% block identifier %}` and `{% endblock %}` are used to identify what line to substitute inherited content. In the child file `{% block identifier %}` and `{% endblock %}` surround the content to be substituted.
The `extends` statement identifies the parent file another template will share content with. Inserted at the top of the child file, `{% extends filename %}` identifies which parent file to insert the `block` content into.

```
<!-- base.html file -->
<html>
  <body>
    {% block content %}{% endblock %}
  </body>
</html>

<!-- template1.html file -->
{% extends base.html %}

{% block content %}
<p>Because of inheritance this file has
html and body tags.</p>
{% endblock %}
```

## Flask Web Forms

Flask provides a simple workflow to implement web forms and gather data while ensuring proper form validation. Using the modules WTForms and Flask-WTF, forms can be created using a class structure to create form elements and collect their data through inherited functionality. Requests can be analyzed using Flasks `request` object.

## `request` Object

When it comes to HTML web forms, the Flask `request` object provides a way to collect data through the client POST request. The data from a form field can be accessed using the field name as a key to the request objects `form` attribute.
A Flask route must add `"POST"` to its `methods` keyword argument. By default `methods` is set to `["GET"]`.

```
## accessing data from "first_name" field

@app.route('/', methods=["GET", "POST"])
def index():
  if len(request.form) > 0:
    name_data =
request.form["first_name"]
```

## `url_for` Function

The function `url_for()` is used within a template to navigate a site through the route function names and not the possibly volatile URL strings. Calling `url_for()` with the route function name as it's parameter is safer way to indicate routes within templates since URL strings can change based on a sites changing file structure.
The `url_for()` function needs to be called within expression delimiters `{{ }}` in order to be processed by the Jinja2 template engine. Keyword arguments can be added to the function call to pass variables to the Flask app route.

```
<a href="{{ url_for('index') }}">Link To
Index</a>

<a href="{{ url_for('another_route',
route_var=some_template_var) }}">Another
Route With Variable</a>
```

## `FlaskForm` Class

Inheriting from the `FlaskForm` class gives an app defined class the functionality such as template representation, gathering data and providing validation. Used in conjunction with WTForm Fields, FlaskForm functionality gives an easy path over how forms are displayed and how valid data is collected within an application. `FlaskForm` is imported through the `flask_wtf` module.
The `"SECRET_KEY"` variable in the Flask application's configuration must be set in order to enable CSRF protection. This is necessary to create forms using `FlaskForm`.

```
from flask import Flask
from flask_wtf import FlaskForm

app = Flask(__name__)
app.config["SECRET_KEY"] = "secret_string"


class MyForm(FlaskForm):
  ## Form elements initialized here
```

## WTForm Fields

WTForm field objects are a simple way to implement web form fields such as text fields and submit buttons. Imported from the `wtforms` module and initialized inside a FlaskForm subclass, field objects provide the representation and functionality of web form elements. Some common tasks of field objects are label management, field validation and data handling.
Here is a detailed list of WTForm fields and their functionality. Some common fields used are:

- StringField()
- SubmitField()
- TextAreaField()
- BooleanField()
- RadioField()

```
from flask_wtf import FlaskForm
from wtforms import StringField,
SubmitField

class MyForm(FlaskForm):
  myStringField("My Text")
  mySubmitField("Submit My Form")
```

## Form Validation

Web form validation is the process of ensuring the correct data is collected by the form before it is sent to the server for processing.
In a web form, validators are used to identify the fields that require specifc data or formats upon submitting the form. This can be as simple as requiring data to be present, to more specific formats such as dates, phone numbers and email addresses. Form field validators can be added to the field's instantiation and are imported from the `wtforms.validators` module. Check the WTForms Validators documentation for more information.
The `validate_on_submit()` function is part of the FlaskForm class and is used with a route to indicate when valid form data has been submitted. This creates an additional branch in the route where data processing can occur.

```
# Validation examples
from flask import Flask
from flask_wtf import FlaskForm
from wtforms import StringField
from wtforms.validators import
DataRequired, Email

app = Flask(__name__)

class MyForm(FlaskForm):
        my_field = StringField("Email",
validators=[DataRequired(), Email()])

@app.route('/', methods=["GET", "POST"])
def my_route():
  my_form = MyForm()
  if my_form.validate_on_submit():
    my_data = my_form.my_field.data
```

## `redirect()` Function

The `redirect()` function is used within a Flask application to move from one route to another. Used in a routes `return` statement, `redirect()` takes a URL string as an argument to identify the correct route to jump to. In many cases `redirect()` is used with `url_for()` as its argument to avoid using URL strings.

Redirecting to a new route is useful for many reasons. In general, a redirect happens after a certain process is done in one route and the logic of another route is needed. A redirect could happen after a form is processed in the form's original route and rendering a template in a new route is desired.

```
# get form data and send to route_two with
redirect
@app.route("/1")
def route_one():
  my_form = MyForm()
  if my_form.validate_on_submit():
    my_data = my_form.my_field.data
    return redirect(url_for("route_two",
data=my_data))
  return render_template("one.html",
template_form=my_form)

# render template with some data
@app.route("/2/<data>")
def route_two(data):
  return render_template("two.html",
template_data=data)
```