undefined Logo

Rendering Pages in Flask Using Jinja

Todd Flask 9 Min Read

If the name **Dreamweaver** means anything to you, you're probably old enough to remember how shitty web development was in the 90s. CSS wasn't a "thing," we abused HTML tables, and tags like `<marquee>` were considered legitimate.Worse than all of this

was the *manual repetition;* HTML was never intended to recognize elementary programming concepts like *inheritance* or *imports.* Even though sites *seemed* to have common elements between pages (such as navigation), the reality was a nightmare of hardcoding these elements by hand on every individual page of a site. Changing a navigation link meant manually making the same change across hundreds of pages. It was great.

We somehow managed to crawl out of that stone-age and into an era of reusable "templates," "partials," "components," and all sorts of ways to ensure we code as little as possible, if at all (looking at you, NPM). Flask's approach to this is via a templating system called **Jinja2**: and HTML preprocessor which lets us write HTML, which changes based on *logic* and *context*. Unlike static HTML, templating systems like Jinja empowers us to do things like share snippets between pages, or render pages conditionally based on context.

If you're familiar with Handlebars or even Django's templating system, Jinja is going to feel quite familiar. The same concepts of any templating engine still apply in Jinja, but Jinja happens to be better. Don't @ me.

Layouts, Pages, and Partials

If we were to dissect the "types" of HTML elements which make up a site, it would be easy to classify chunks of HTML as belonging to one of 3 types:

- **Layout**: The majority of web apps have some sort of "skeleton" structure shared between pages. This can be as simple as repeating the same <head> across multiple pages or enforcing a general physical structure, such as a standard *content + sidebar* look. A coherent web app will likely have very few unique layouts (I don't think I've ever needed more than 1 per project).
- **Page**: Page templates are the self-explanatory meat and potatoes of templating. While this blog has hundreds of posts, each "post" is merely an instance of a single **post template**: a uniquely assembled section of our site that can be replicated with different data.
- **Partial**: Partials are standalone snippets that can be shared by pages where needed. Think of navigation elements, widgets, or anything else designed to be shared across parts of your site

Setting up for Action

We're going to create a tiny Flask app to demonstrate how we can use Jinja to build websites utilizing the three concepts we just learned. We'll start by creating an app that uses 1 of each of the three template types: **layout.html, home.html,** and **nav.html**. These templates will live in a */templates* folder, thus leaving us with an app looking as such:

```
/flask-jinja-tutorial
├── /flask_jinja_tutorial
│   ├── __init__.py
│   ├── routes.py
│   ├── /static
│   └── /templates
│       ├── home.html
│       ├── layout.html
│       └── navigation.html
└── wsgi.py
```

Templates won't do us any good without an app to serve them. After creating a Flask app object in **init.py**, let's create a route in **routes.py** to set up a basic homepage:

routes.py

```
"""Route declaration."""
from flask import current_app as app
from flask import render_template


@app.route('/')
def home():
    """Landing page."""
    return render_template('home.html',
                           title="Jinja Demo Site",
                           description="Smarter page templates \
                                with Flask & Jinja.")
```

All we're doing here is setting up a route called `home()`, which serves the **home.html** template any time a user visits our Flask app. Along with specifying which page template to serve, we also pass in two keyword arguments to `render_template()`: **title** and **description.** We'll see these in action in just a moment.

Rendering a Template

Let's check out what goes down when we fulfill the seemingly simple task of serving a page template. Here's what I have for **home.html**:

home.html

```
{% extends 'layout.html' %}

{% block content %}
    <div class="container">
```

```
        <h1>{{title}}</h1>
        <p>{{description}}</p>
    </div>
{% endblock %}
```

Whoa, what is this `{% extends 'layout.html' %}` nonsense? And what does `{% block content %}` mean?

You'll recall earlier when we created a file called **layout.html** in our */templates* folder. We've covered that "layouts" contain repetitive structures (or boilerplate) shared across pages. To understand **home.html** better, let's peek at **layout.html**:

layout.html

```
<!doctype html>
<html>

  <head>
    <title>{{title}}</title>
    <meta charset="utf-8">
    <meta name="description" content={{description}}>
    <link rel="shortcut icon" href="/favicon.ico">
  </head>

  <body>
    {% include 'navigation.html' %}
    {% block content %}{% endblock %}
  </body>

</html>
```

When a requested page template "extends" another, the requested template will contain all HTML of the extended parent. When **home.html** extends **layout.html**, we need to figure out *where* in **layout.html** our page will be mounted. Jinja handles this by declaring and matching reserved spaces in our templates named "blocks."

layout.html has a single block called "content" which happens to appear in *both* our page template and the layout it extends- this lets Jinja know that the contents of **home.html** should be loaded to this block, which effectively results in the following template at runtime:

The result of **home.html** extending **layout.html**.

```
<!doctype html>
<html>
```

```
  <head>
    <title>{{title}}</title>
    <meta charset="utf-8">
    <meta name="description" content={{description}}>
    <link rel="shortcut icon" href="/favicon.ico">
  </head>

  <body>
    {% include 'navigation.html' %}
    <div class="container">
        <h1>{{title}}</h1>
        <p>{{description}}</p>
    </div>
  </body>

</html>
```

The above demonstrates what a rendered **home.html** page after being loaded into **layout.html**. Our page is already looking a lot more like HTML.

Now that you've wrapped your head around the concept of blocks, here's a curveball mindfuck to keep you in check: Layouts and pages can contain *multiple* blocks, which allows us to do some exciting things. For example, let's say we want the ability to inject styles on to our layout, depending on which page extends it. Here's **home.html** with two blocks: one for CSS styles to be inserted in our page `<head>`, and another for loading page content:

home.html

```
{% extends 'layout.html' %}

{% block css %}
  <link href="{{ url_for('static', filename='css/home.css') }}" rel="stylesheet">
{% endblock %}

{% block content %}
    <div class="container">
        <h1>{{title}}</h1>
        <p>{{description}}</p>
    </div>
{% endblock %}
```

I bet you could guess what would happen when we load **home.html** into this new layout:

layout.html

```
<!doctype html>
<html>

  <head>
    <title>{{title}}</title>
    <meta charset="utf-8">
    <meta name="description" content={{description}}>
    <link rel="shortcut icon" href="/favicon.ico">
    {% block css %}{% endblock %}
  </head>

  <body>
    {% include 'nav.html' %}
    {% block content %}{% endblock %}
  </body>

</html>
```

The *"CSS"* and *"content"* blocks from **home.html** are able to be matched and loaded into their respective blocks in **layout.html** This empowers us to change layout templates without being constrained by the DOM structure of **layout.html**.

Let's address the `{{title}}` and `{{description}}` blocks in the room. These double-bracket items represent reserved variable names. Back in **routes.py**, we passed a couple of keyword arguments matching these names when we used `render_template()`. Our template will render these values at runtime:

Our page with rendered kwargs.

```
<!doctype html>
<html>

  <head>
    <title>Jinja Demo Site</title>
    <meta charset="utf-8">
    <meta name="description" content="Smarter page templates with Flask & Jinja.">
    <link rel="shortcut icon" href="/favicon.ico">
    <link href="/static/css/home.css" rel="stylesheet">
  </head>

  <body>
    {% include 'navigation.html' %}
    <div class="container">
        <h1>Jinja Demo Site</h1>
        <p>Smarter page templates with Flask & Jinja.</p>
    </div>
  </body>

</html>
```

This leaves us with one last Jinja statement to cover in our template: `{% include 'navigation.html' %}`. `include` tells Jinja "load a separate template named **nav.html** here." These standalone templates which get dropped into larger templates are called *partials*. My **navigation.html** is a simple header nav which looks like this:

navigation.html

```
<header>
   <nav>
    <a href="https://example.com">Link 1</a>
    <a href="https://example.com">Link 2</a>
    <a href="https://example.com">Link 3</a>
  </nav>
</header>
```

Now we can see what our rendered homepage looks like in its entirety:

Our rendered homepage.

```
<!doctype html>
<html>

  <head>
    <title>Jinja Demo Site</title>
    <meta charset="utf-8">
    <meta name="description" content="Smarter page templates with Flask & Jinja.">
    <link rel="shortcut icon" href="/favicon.ico">
    <link href="/static/css/home.css') }}" rel="stylesheet">
  </head>

  <body>
    <header>
       <nav>
        <a href="https://example.com">Link 1</a>
        <a href="https://example.com">Link 2</a>
        <a href="https://example.com">Link 3</a>
      </nav>
    </header>
    <div class="container">
        <h1>Jinja Demo Site</h1>
        <p>Smarter page templates with Flask & Jinja.</p>
    </div>
  </body>

</html>
```

We've done it! We've dissected Jinja's three template "types," and used each one to build a single dynamic HTML page. This level of working knowledge is already respectable- it's quite possible to build web apps armed only with the things we've covered thus far. That said, Jinja can go much, much deeper.

Logic in Jinja

Jinja isn't just about sharing HTML between pages. Jinja can perform some significant business logic to render parts of templates conditionally. The most obvious example is a simple `if` statement:

Standard Jinja `if` statement.

```
<div class="status">
  {% if status.active %}
        <p>Activated.</p>
  {% elif status.disabled %}
        <p>Disabled.</p>
  {% else %}
    <p>OH GOD WHAT IS THE STATUS?!</p>
  {% endif %}
</div>
```

Jinja `if` statements can contain any conditional operator you'd expect. `{% if status == 1 %}` and `{% if "active" in status %}` are both valid conditionals assessing numerical and string values, respectively.

Looping & Data Types

We passed a couple of strings to **home.html** earlier, but we could actually almost any conceivable data type as well. Let's reimagine **navigation.html** as a dynamic template. Instead of hardcoding links into HTML, let's instead build these nav links based on data we pass to **navigation.html** from **routes.py**.

An excellent way to represent this data would be a *list* of *dictionaries:*

routes.py

```
...

@app.route('/')
def home():
    """Landing page."""
    nav = [{'name': 'Home', 'url': 'https://example.com/1'},
```

```
            {'name': 'About', 'url': 'https://example.com/2'},
            {'name': 'Pics', 'url': 'https://example.com/3'}]
    return render_template('home.html',
                           nav=nav,
                           title="Jinja Demo Site",
                           description="Smarter page templates \
                                with Flask & Jinja.")
```

We can pass a Python list like nav to Jinja templates the same way we passed title and description earlier. Now we can adjust navigation.html to build links dynamically based on the contents of this list:

navigation.html

```
<header>
   <nav>
    {% for link in nav %}
      <a href="{{ link.url }}">{{ link.name }}</a>
    {% endfor %}
  </nav>
</header>
```

Jinja can loop through iterable data types (such as lists) using almost identical syntax to Python. The above example is looping through our list `nav`. Each member of `nav` is a dictionary, thus we can get the values of each dictionary using dot notation, as seen via `{{ link.url }}` and `{{ link.name }}`.

The Path of a Jinja Ninja

As you can gather from the official Jinja docs, we've only scratched the surface of what Jinja can handle. Knock yourself out, but be warned: mastering every corner of templating is like a more worthless version of being a RegEx guru. It's cool to have powerful tools available as offhand knowledge, but there's a hard line between mastering commonly applicable patterns versus being "kind of an asshole."

In case you're interested, the source code for this tutorial can be found on GitHub here: