CODING NOMADS

6) Flask Templates, Jinja, and You Lesson

How to Use the Jinja Template Engine

33 min to complete · By Brandon Gigous

Contents

Hopefully, in the last section, you've gotten comfortable with the basics of how to make a Flask app and how to go about making basic routes.

But get ready, because you're about to go from 0 to 60 in this course. You'll be picking up the pace going forward. There won't be full code examples, but there will be code snippets with enough information to guide you through to making a full Flask app. As long as you read carefully, or ask for guidance on Discord or from your mentor, you should be just fine!

How did you do with making your own routes? Did you try adding some flavor with HTML to make it look more professional? Did your wrists start getting sore reaching for those angle brackets all the time? Seems like a pain, right?

Having to make a brand new HTML string every time you want to make a new route feels very tedious. But here's the great news with Flask: You don't have to do that all the time! This lesson introduces the Jinja2 template engine to make your life easier!

What is a Flask Template?

The only job of our view functions should be to generate a response to a request; that's it. Having to also *make* the pages? What a pain! For that, the creators of Flask also built Jinja, also called Jinja2, which is a Flask template engine.

Flask **templates** are files that contain the text of a response. Pages that say things like, "Upload your corn flake collection here!" are great, but that's only text.

Templates also support placeholder variables for the dynamic parts of the page, meaning those that change depending on the context of a request (whose corn flake collection, if the corn flakes are gluten-free, etc). These variables get replaced with real values through a process called **rendering**. And that's what Jinja's job is, to take the values those variables should take on passed in from Flask, then render those values in with the surrounding text. Let's try this thing out!

Making a Jinja2 Template

Before you get started, make a folder named `templates`, which will live right inside your `flask-webdev` directory. That's because, by default, Flask looks for your templates in a folder called exactly that: `templates`. Once you've done that, you're ready to go!

It's Just HTML!

First things first, let's get a simple template rendered. Think of a template as just an HTML file. Make a new template file, `templates/index.html`. Put in the following:

```
<h1>Hello Web World!</h1>
```

Now in your `hello.py`, you'll need to import `render_template`, then replace your `index()` function:

```
@app.route('/')
def index():
    return render_template('index.html')
```

With the `render_template()` function, all you need to do is indicate the path to the template relative to your `templates` folder, which is simply `"index.html"`. When a request is received for the index page and `render_template()` is called, Flask will enlist the help of Jinja2 to render the template.

Let 'er rip! Run the app with `flask run` and watch as you get the same output. A little boring, huh? Just wait, you're gonna go *dynamic*.

Flask `render_template`

While a template is pretty much HTML, it's also a little more than that with Jinja. To help demonstrate, let's make another route, this time a dynamic one. Feel free to use one you might have made before in order to complete this next part.

The `render_template()` has more power than it might look. This function utilizes the well-known Pythonism known as *keyword arguments*, often seen as `**kwargs` in the wild. To call the function in the previous `user()` function example and still get the same effect, you'd do this:

```
@app.route('/user/<username>')
def user(username):
    return render_template("user.html", username=username)
```

In this example, you indicate which template you want Jinja to render and also pass in our `name` parameter along to Jinja. The `name` on the left side of the `=` is what Jinja will use as the placeholder name inside the template. The other `name` on the right side is the variable in the function scope.

Now when you load the `user.html` template, you'll see—whah!? You say you haven't made such a template yet that can handle this passed-in variable? Oh dear, let's take care of that right away!

Using Variables in Jinja Templates

From the last lesson, you know from believing your nomadic Flask course creator and all his sage, vast wisdom (who couldn't quite type the previous with a straight face) that the Jinja template engine can handle variables as long as the Flask application does its part by passing them over. For a simple but dynamic page that does this relay, the view function would look like this:

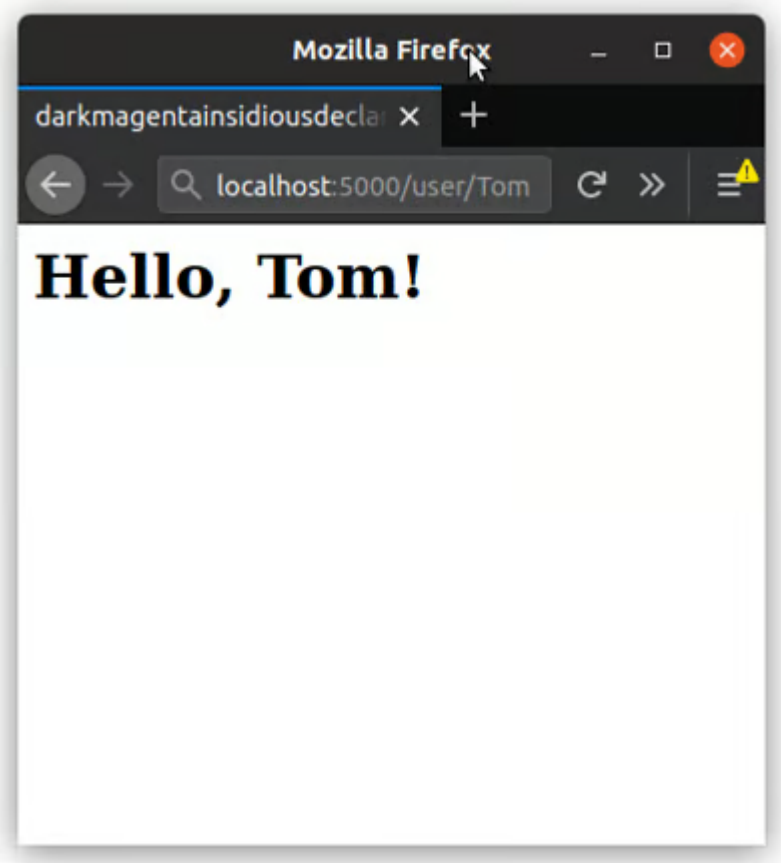
```
@app.route('/user/<username>')
def user(username):
    return render_template("user.html", username=username)
```

Crack open a new `user.html` template file and crack your knuckles 'cause here's the template content:

```
<h1>Hello, {{ username }}!</h1>
```

Wuzzat?! Looks like some weird HTML, but this is how Jinja likes its variables prepared for breakfast. Er, to render. The double curly brackets `{{}}` tell Jinja that whatever's inside them is a placeholder that should have a value assigned to it, and to render the

template with that value instead. If you head to `localhost:5000/user/Tom` in your browser, you'll be greeted as Tom, even though that's probably not your name!



Flexible Placeholders

Here's the really neat thing: you can type almost anything Pythonic in these placeholder sections, and Jinja will know how to render it. Even cooler, you can put these placeholders almost *anywhere* in the template, and Jinja will still expand them to their actual values. That is, as long as you pass every value Jinja needs to render into the template. Here are some other other examples:

```
<p>Sticks and stones may break my bones but Jinja understands my dic
<p>George Washington once said: "With Jinja, I can put two variables
<font size="{{ my_int }}">THIS TEXT IS HUGE! or tiny</font>
<p>I scream for {{ my_str.upper() }}</p>
```

Dictionaries, indexing a list with another variable, changes to styling, even calling object methods. All of it works in a placeholder if it's 1) Python code and 2) Jinja knows about it.

Oh, and one more thing! Jinja also includes filters which can go after a variable with a pipe character in between. For that last example about screaming, you can also do the following to get the same result:

```
<p>I scream for {{ my_str|upper }}</p>
```

`upper` is just one filter to choose from, but here are some others:

Filter name	Description
`capitalize`	Uppercase the first character
`lower`	Lowercase all characters
`safe`	Render value without applying escaping
`striptags`	Remove any HTML tags before rendering
`title`	Return a titlecased version of the value
`trim`	Strip leading and trailing characters
`upper`	Uppercase all characters

You just learned about variables in Jinja, but don't you want more *control* over your templates? You're in luck because coming up, you'll apply control structures to your templates.

You were just introduced to variables in Jinja and how they put the power of Python into your templates, but there must be more, right? There is, and in this section, you'll get to know the control structures that Jinja provides for you—the `if` s, the `for` s, and a few mores!

Control Structures

Python wouldn't be Python without its conditionals and control flow. And it turns out Jinja wouldn't be Jinja for the same reason! Let's just go straight through what control structures Jinja has, and take note that each of these uses a `{% ... %}` construct:

Jinja `if`, `else`

"Should I display that element, or shouldn't I?" That, along with other hard and not-so-hard questions, can be handled with conditional statements:

```
{% if user %}
    Hello, {{ user }}!
{% else %}
```

```
    Hi, Anonymous!
{% endif %}
```

Now you can know who's looking at the page! Unless they don't tell us...

for

Lists of elements are no match for Jinja; they shall be rendered consecutively if you so choose! These are great for using in HTML list tags.

```
<ul>
    {% for song in favorite_songs %}
        <li>{{ song }}</li>
    {% endfor %}
</ul>
```

Delicious!

macro

Macros? What the heck are those? Think of them as the Jinja version of Python functions. They're great for avoiding monotonous tasks. Watch:

```
{% macro render_song_title(song) %}
    <li>{{ song }}</li>
{% endmacro %}

<ul>
    {% for song in favorite_songs %}
        {{ render_song_title(song) }}
    {% endfor %}
</ul>
My least favorite song ever: {{ render_song_title(bad_song) }}
```

Automation! The best thing since sliced bread.

`import`

> *To copy others is necessary, but to copy oneself is pathetic.*
>
> *Pablo Picasso*

Y'know, Picasso is onto something here. In some cases, it's extremely useful to copy one template into another, and that's exactly what Jinja's `import` statement does. But, "to copy oneself is pathetic?" If he's talking about Jinja, I think maybe he meant "silly."

```
{% import 'common.html' %}
```

Would some `macro` statements and an `import` or two be useful? You betcha. In fact, that's the definition of "synergy":

```
{% import 'macros.html' as macros %}
<ul>
    {% for song in favorite_songs %}
        {{ macros.render_song_title(song) }}
    {% endfor %}
</ul>
```

Defining one or more macros and one template, then importing them for painless access in another? Sweet.

`extends` and `block`

Are you a Java(Script) coder? 'Cause the `extends` keyword makes a comeback in Jinja (if you can call it that). You're darn tootin': **inheritance** in templates! Using `block` structures, you can define and then **extend** or even override *blocks* of template code.

If you make a "base" template called `base.html`, you can "reserve" blocks of code to hold certain content in certain places that can be inherited later.

```
<html>
<head>
    {% block head %}
    <title>{% block title %}{% endblock %} - My App</title>
    {% endblock %}
</head>
<body>
    {% block body %}
    {% endblock %}
</body>
</html>
```

You'll see here that blocks of code are defined with arbitrary names, in this case, `head`, `title`, and `body`. Another template that extends this base template (let's call it `index.html`) might look like this:

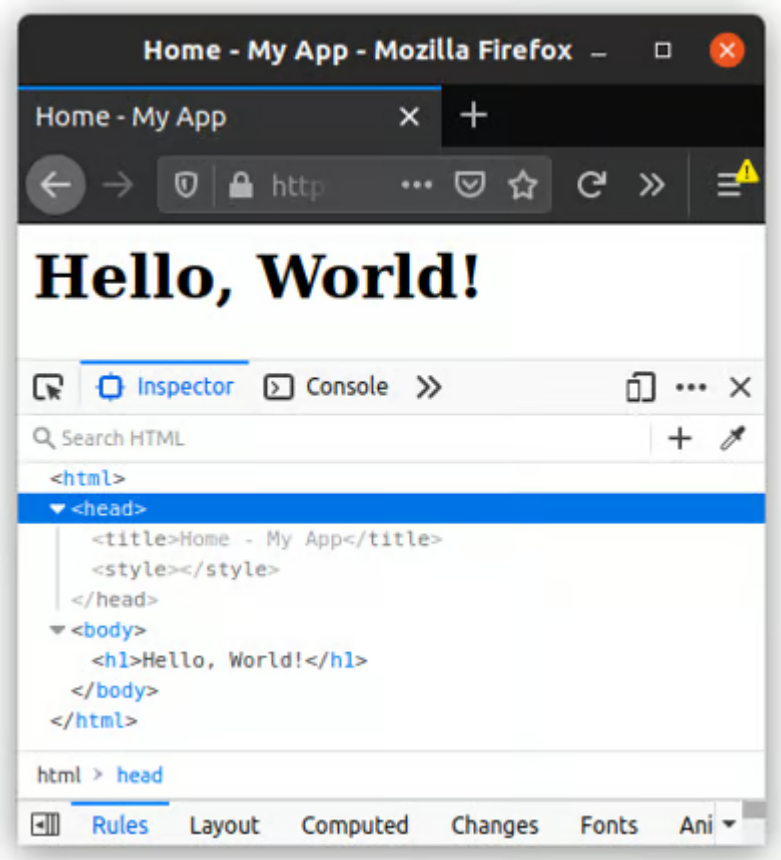
```
{% extends "base.html" %}
{% block title %}Home{% endblock %}
{% block head %}
    {{ super() }}
    <style></style>
{% endblock %}
{% block body %}
<h1>Hello, World!</h1>
{% endblock %}
```

It has an `extends` control structure, which means it's definitely a derived template. Alrighty, try to follow along here: so you see how the `title` block is inside the `head` block in `base.html`? It is *outside* the `head` block in `index.html`. For blocks that show up in both base and derived templates, the content in the derived template *always* gets used instead of the content in the base template. Here's how `index.html` is rendered:

- `title` - content is `Home`
- `head` - the textual content is `Home - My App` since `title` is contained within `head`, and shows up as the page's title (the browser tab text); `super()` is used to

bring in the content from the base class, then afterwards it is *extended* with a `<style>` tag

- `body` - content is just `Hello World!`



Jinja Comments

These don't actually *control* anything, but they can be useful! Instead of using `<!-- HTML comments -->` for templates, it's better to use `{# these comments #}`, mainly because the Jinja comments won't get included in the final HTML, whereas HTML comments will get sent to the client and visible to anyone who opens the inspector.

What's Next: Style Points

Phew! Those are all the important control structure you'll need to be familiar with in this course. Don't worry, you'll see them again and again so you'll get used to them.

But gosh, does any of this make you feel a little uneasy? Now don't get me wrong, Flask is freakin' cool, but these pages look like they came from the Internet Stone Age, back when dial-up was a thing. And to answer the next question you almost certainly have: *YES!* Of course, there's an easy way to get a much prettier, modern look! Continue on with the next lessons for some serious style.

Summary: How to Use the Jinja Template Engine

You've now dipped your toes into creating more sophisticated applications with Flask, and it looks like you're picking up steam! In this lesson, you've:

- Discovered that **templates** are your best friends for avoiding repetition and that you can use the Jinja template engine to render responses dynamically.
- Set up a `templates` folder and learned how to render simple HTML templates using `render_template()`.
- Understood how to pass **variables** to templates in Flask, enabling dynamic content to be displayed to the user, such as personalized greetings.
- Played with dynamic placeholders in templates, learning that you can essentially run Python code in your HTML using Jinja syntax.
- Grasped **control structures** in Jinja, like `if`, `elif`, `else`, `for`, and `import`, which allow you to add conditional logic and loops to your templates.
- Tackled template **inheritance** with the `extends` and `block` keywords.
- Touched on **macros** and `import` in Jinja, which act like functions in Python and can be reused across different templates.

Inheritance can be tricky to wrap your head around, so here is an FAQ of common questions students have.

Get Mentorship

Template Inheritance FAQ

Here are questions that have come up from students and may come up for you. :)

Q: I inherited from bootstrap base template in template A, but when rendering template A, nothing shows up. What's wrong?

This could be because you didn't place down any `block` s in your inherited template. Flask-Bootstrap has pre-defined `block` s that are "copied" over when you use `extends` . Plain HTML gets overridden.

If you did place down `block` s, you may not have used `{{ super() }}` inside of it, which would override anything you defined in that `block` in the base template.

Q: What is the difference between `<head>` & `<header>` ?

Here's a helpful Stackoverflow post that clarifies: What is the real difference between the "head" and "header" tag?

Q: Some parts of my HTML show up twice when using blocks

This happens sometimes because you place down a parent `block` along with a child `block` inside it in the inherited template. Usually, it is enough to just place down the *child* `block` only and add the additional HTML you want to it.

Template inheritance allows you to "say a lot with just a little," meaning it gives you the capability to plop in chunks of HTML with only a couple lines of Jinja code. Within

inherited templates, placing down *fewer* `block` s already pre-defined in your base template is usually better. So when in doubt, try using a subset of `block` s from your base template.

Q: I don't know where to start. How do I know the base template is doing what I need it to?

Let's assume you are starting with a good amount of HTML code without any Jinja. There are probably multiple HTML files that have a lot of similarities. The idea of creating a base template is to capture these similarities while still allowing for child templates to have their own content. On somewhat rare occasions, a child template may "overwrite" some parts of the HTML structure. Another nice thing about having a base template is that changes to the base template are carried over to the child templates, which saves oodles of time on the frontend so you can focus on making the backend solid.

If you're stuck, the following is a good workflow. The basis of it is to start with a bunch of already written HTML code and strip it down to the parts that are common to all templates.

1. First, make *two* copies of the index template. Name one copy as `base.html` and the other as `index2.html` . For now you'll focus on the `base.html` template, and you'll come back to `index2.html` a few steps afterward.
    - The index template is often the simplest template of the bunch, so it is easiest to start with
2. Create a couple of temporary routes for displaying this base template. It can be as simple as

```
# Temporary route for viewing and debugging base template
@app.route('/base')
def base_temp():
    return render_template("base.html")

# Temporary route for *recreating* the index template using block
@app.route('/index2')
```

```
def index2_temp():
    return render_template("index2.html")
```

3. In `base.html`, define `block`s based on the structural parts of a page. Blocks make your templates extensible and flexible, otherwise you'd have to rewrite HTML for entire pages.

   a. Start with `head` and `body` to start. The `head` block will be defined right inside the `<head>` tag, which in turn contains the title, links to stylesheets and scripts, and other metadata. The `body` is what your users will see. Example:

```
<head>
{% block head %}
    <title>My Title</title>
    <link rel="stylesheet" href="static/style.css">
{% endblock head %}
</head>
<body>
{% block body %}
    <header>
        ...
    </header>
    ...
{% endblock body %}
</body>
```

   b. After that, define the nested blocks, like `title` inside the `head` block and `header` inside the `body` block:

```
<head>
{% block head %}
    <title>{% block title %}My Title{% endblock title %}</title>
    <link rel="stylesheet" href="static/style.css">
{% endblock head %}
</head>
<body>
```

```
    {% block body %}
        <header>
        {% block header %}
            ...
        {% endblock header %}
        </header>
        ...
    {% endblock body %}
    </body>
```

Starting to see the pattern?

c. Continue onto the next layer of nested blocks. After defining blocks for this layer, you probably have enough blocks for a decent start to a base template! Any deeper than 4 nested blocks starts to become to much. Keep in mind you don't need a block for every HTML tag; just a handful of blocks will do.

4. Launch your app and take a look at how your `base.html` template looks. Does it still look like your index page? If something doesn't look right, go back to step 3 adjust your blocks until things look the same.

5. From your `base.html` template, delete any content that is **exclusive** to the index page, but **keep** the parts that will probably be on almost every page. Things like "Welcome to my site!" would be exclusive to the index page. Things like the header and navigation bar will likely be shown on just about every page. If you're not sure if something should stay or go, err on the side of keeping it. After this, you've created the first draft of your base template!

6. Now you're ready to pull up `index2.html`. If you're using an IDE that supports side-by-side tabs like VSCode, pull up the original `index.html` alongside it. In `index2.html`, *delete* the existing HTML code and try to *reconstruct* the index page by extending `index2.html` from `base.html` and using the `blocks` defined within `base.html`. In other words, use *template inheritance* to re-make the index page. Remember you can check the `/index2` route you made in step 2 to see how it compares to `/index`.

    - Most of the time, you won't need to change the HTML inside an inherited block, but if you do, you can redeclare it in a derived template and change or add anything you need to the block.

Q: I added Flask Bootstrap to my project, and now the blocks in my existing templates are screwed up when rendered. How can I fix it?

When you initialize Flask-Bootstrap in your project and inherit a template from `bootstrap/base.html`, it will define some blocks already, which you can see here. If you have any of these blocks in your templates, keep in mind where you do or don't have `{{ super() }}`.

Don't be intimidated by Flask-Bootstrap! You can actually see what Flask-Bootstrap has defined in its `bootstrap/base.html` template here. It's only 34 lines and might be simpler than you think.

Previous Next → Video: What Are Jinja Templates in Flask?