undefined Logo

The Art of Routing in Flask

Todd Flask 11 Min Read

It's hard to imagine a more critical feature of web frameworks than routing: the humble act of mapping URLs to actions, such as serving pages or data. It isn't often you find somebody sad or miserable enough to expand on such an inglorious feature. As it turns

out, I am apparently both and miserable enough to be the kind of person who writes tutorials about routing.

At first glance, it's hard to imagine routing to be an "art." We'll typically reserve a URL path, such as `/` or `/home`, associate this with a page template, and serve said template to the user, perhaps with added business logic. That perspective works fine for small-scale applications, but meaningful applications (or APIs) aren't static one-to-one mappings. Apps are a medium for data such as user-generated content such as user profiles or author posts, and routes define the way our users will access data which is always changing. To build products larger than ourselves, we need to arm them with the ability to grow in ways we can't foresee, which means defining dynamic routing opportunities that can potentially grow endlessly. This is where we have the chance to be artistic.

Today we're covering the finer details of how to define and build smart routes to accommodate dynamic applications and APIs. If you have *any* prior experience with MVC frameworks, you should be able to follow along just fine; almost no previous knowledge of Flask is needed to keep up.

Defining Routes & Views

Every web framework begins with the concept of serving content at a given URL. *Routes* refer to URL patterns of an app (such as *myapp.com/**home*** or *myapp.com/**about***). *Views* refer to the content to be served at these URLs, whether that be a webpage, an API response, etc.

Flask's "Hello world" example defines a *route* listening at the root our app and executes a view function called `home()`:

Flask's "Hello world!"

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def home():
    return "Hello World!"
```

`@app.route("/")` is a Python decorator that Flask provides to assign URLs in our app to functions easily. It's easy to understand what is happening at first glance: the decorator is telling our `@app` that whenever a user visits our app domain (*myapp.com)* at the given `.route()`, execute the `home()` function. If you aren't familiar with Python decorators, they're essentially logic which "wraps" other functions; they always match the syntax of being a line above the function they're modifying.

The names we chose for our view functions hold significance in Flask. Years of web development taught us that URLs tend to change all the time, typically for business or SEO purposes, whereas the *names* of pages generally always stay the same. Flask recognizes this by giving us ways to move users around our app by referring to the names of views by essentially saying, *"redirect the user to whatever the URL for* `home()` *happens to be."* We'll demonstrate that in a moment, but it's worth recognizing the importance of view names for this reason.

Protip

We can handle multiple routes with a single function by simply stacking additional route decorators above any route! The following is a valid example of serving the same "Hello World!" message for 3 separate routes:

```
...

@app.route("/")
@app.route("/home")
@app.route("/index")
def home():
    return "Hello World!"
```

Route HTTP Methods

In addition to accepting the URL of a route as a parameter, the `@app.route()` decorator can accept a second argument: a list of accepted HTTP methods. Flask views will only accept GET requests by default, unless the route decorator explicitly lists which HTTP methods the view should honor. Providing a list of accepted methods is a good way to build constraints into the route for a REST API endpoint, which only makes sense in specific contexts.

Routes with restricted HTTP methods.

```
from flask import Flask
app = Flask(__name__)

@app.route("/api/v1/users/", methods=['GET', 'POST', 'PUT'])
def users():
    ...
```

Dynamic Routes & Variable Rules

Static route URLs can only get us so far, as modern-day web applications are rarely straightforward. Let's say we want to create a profile page for every user that creates an account within our app or dynamically generate article URLs based on the publication date. Here is where variable rules come in.

Dynamic values in routes.

```
...

@app.route('/user/<username>')
def profile(username):
    ...

@app.route('/<int:year>/<int:month>/<title>')
def article(year, month, title):
    ...
```

When defining our route, values within carrot brackets `<>` indicate a variable; this enables routes to be dynamically generated. Variables can be type-checked by adding a colon, followed by the data type constraint. Routes can accept the following variable types:

- **string**: Accepts any text without a slash (the default).
- **int**: Accepts integers.
- **float**: Accepts numerical values containing decimal points.
- **path**: Similar to a string, but accepts slashes.

Unlike static routes, routes created with variable rules *do* accept parameters, with those parameters being the route variables themselves.

Types of View Responses

Now that we're industry-leading experts in defining route URLs, we'll turn our attention to something a bit more involved: route logic. The first thing we should recap is the types of responses a view can result in. The top 3 common ways a route will conclude will be with generating a **page template**, providing a **response**, or **redirecting** the user somewhere else (we briefly looked over these in part 1).

Remember: views always conclude with a `return` statement. Whenever we encounter a `return` statement in a route, we're telling the function to serve whatever we're returning to the user.

Rendering Page Templates

You're probably aware by now that Flask serves webpages via a built-in templating engine called Jinja2 (and if you don't know, now you know). To render a Jinja2 page template, we first must import a built-in Flask function called `render_template()`. When a view function returns `render_template()`, it's telling Flask to serve an HTML page to the user which we generate via a Jinja template. Of course, this means we need to be sure our app has a **templates** directory:

Setting a "templates" directory

```
from flask import Flask, render_template

app = Flask(__name__, template_folder="templates")
```

With the above, our app knows that calling `render_template()` in a Flask route will look in our app's **/templates** folder for the template we pass in. In full, such a route looks like this:

Render a page template

```
from flask import Flask, render_template


app = Flask(__name__, template_folder="templates")

@app.route("/")
def home():
    """Serve homepage template."""
    return render_template("index.html")
```

`render_template()` accepts one positional argument, which is the name of the template found in our templates folder (in this case, **index.html**). In addition, we can pass *values* to our template as keyword arguments. For example, if we want to set the title and content of our template via our view as opposed to hardcoding the, into our template, we can do so:

Render template with variables

```
from flask import Flask, render_template
app = Flask(__name__,
            template_folder="templates")

@app.route("/")
def home():
    """Serve homepage template."""
    return render_template(
```

```
        'index.html',
        title='Flask-Login Tutorial.',
        body="You are now logged in!"
    )
```

For a more in-depth look into how rendering templates work in Flask, check out our piece about creating Jinja templates.

Making a Response Object

If we're building an endpoint intended to respond with information to be used programmatically, serving page templates isn't what we need. Instead, we should look to `make_response()`.

`make_response()` allows us to serve up information while also providing a status code (such as 200 or 500), and also allows us to attach headers to the said response. We can even use `make_response()` in tandem with `render_template()` if we want to serve up templates with specific headers! Most of the time `make_response()` is used to provide information in the form of JSON objects:

Serving JSON

```
from flask import Flask, make_response


app = Flask(__name__)

@app.route("/api/v2/test_response")
def users():
    headers = {"Content-Type": "application/json"}
    return make_response(
        'Test worked!',
        200,
        headers=headers
    )
```

There are three arguments we can pass to make_response(). The first is the body of our response: usually a JSON object or message. Next is a 3-digit integer representing the HTTP response code we provide to the requester. Finally, we can pass response headers if so chose.

Redirecting Users Between Views

The last of our big three route resolutions is `redirect()`. Redirect accepts a string, which will be the path to redirect the user to. This can be a relative path, absolute path, or even an external URL:

Route redirects

```
from flask import Flask, redirect


app = Flask(__name__)

@app.route("/login")
def login():
    return redirect('/dashboard.html')
```

But wait! Remember when we said its best practice to refer to routes by their names and *not* by their URL patterns? That's where we use `url_for()` comes in: this built-in function takes the name of a view function as *input,* and will *output* the URL route of the provided view. This means that changing route URLs won't result in broken links between pages! Below is the *correct* way to achieve what we did above:

Redirecting to route names

```
from flask import Flask, redirect, url_for


app = Flask(__name__)

@app.route("/login")
def login():
    return redirect(url_for('dashboard'))
```

Building Smarter Views

Building a respectable view requires us to excel at both the soft-skills of working with web frameworks as well as the hard-skills of merely knowing the tools available to us.

The basic "soft-skill" of building a route is conceptually straightforward, but difficult in practice for many newcomers. I'm referring to the basics of MVC: the concept that views should only contain logic resolves the response of a view (*not* extraneous business logic which should be encapsulated elsewhere). It's a skill that comes from habit and example: a bit out of this post's scope but bears repeating regardless.

Luckily, the "hard-skills" are a bit more straightforward. Here are a couple of tools essential to building elegant routes.

The Request Object

`request()` is one of the "global" objects we mentioned earlier. It's available to every route and contains all the context of a request made to the said route. Take a look at what things are attached to `request` which we can access in a route:

- **request.method**: Contains the method used to access a route, such as GET or POST. `request.method` is absolutely essential for building smart routes: we can use this logic to have one route serve multiple different responses depending on what method was used to call said route. This is how REST APIs provide different results on a GET request versus a POST request (`if request.method == 'POST':` can open a block only pertaining to POST requests in our route).
- **request.args**: Contains the query-string parameters of a request that hit our route. If we're building an endpoint that accepts a **url** parameter, for example, we can get this from the request as `request.args.get('url')`.
- **request.data**: Returns the body of an object posted to a route.
- **request.form**: If a user hits this route as a result of form submission, `request.form` is our way of accessing the information the form posted. For example, to fetch the provided username of a submitted form, `request.form['username']` is used.
- **request.headers**: Contains the HTTP response headers of a request.

Here's an example I took from a popular plugin called Flask-Login: a library that handles user authentication in Flask. This is a complex example of a view that utilizes most of the things we just covered in a single route. I don't expect you to understand everything that's happening here, but it's good to see how powerful and versatile we can make a single route simply by using the properties of `request`:

A complex Flask route

```
...

@app.route('/signup', methods=['GET', 'POST'])
def signup_page():
    """User sign-up page."""
    signup_form = SignupForm(request.form)
    # POST: Sign user in
    if request.method == 'POST':
        if signup_form.validate():
            # Get Form Fields
            name = request.form.get('name')
            email = request.form.get('email')
```

```
            password = request.form.get('password')
            website = request.form.get('website')
            existing_user = User.query.filter_by(email=email).first()
            if existing_user is None:
                user = User(name=name,
                            email=email,
                            password=generate_password_hash(password, method='sha256'),
                            website=website)
                db.session.add(user)
                db.session.commit()
                login_user(user)
                return redirect(url_for('main_bp.dashboard'))
            flash('A user already exists with that email address.')
            return redirect(url_for('auth_bp.signup_page'))
    # GET: Serve Sign-up page
    return render_template('/signup.html',
                           title='Create an Account | Flask-Login Tutorial.',
                           form=SignupForm(),
                           template='signup-page',
                           body="Sign up for a user account.")
```

The "g" Object

Let's say we want a view to access data that *isn't* passed along as part of a `request` object. We already know we can't pass parameters to routes traditionally: this is where we can use Flask's `g`. "G" stands for "global," which isn't a great name since we're restricted by the application context, but that's neither here nor there. The gist is that `g` is an object we can attach values to.

We assign values to `g` as such:

Assign a value to g.

```
from flask import g


def get_test_value():
    if 'test_value' not in g:
        g.test_value = 'This is a value'
    return g.test_value
```

Once set, accessing `g.test_value` will give us `'This is a value'`, even inside a route.

The preferred way of purging values from `g` is by using `.pop()`:

Removing a value from g.

```
from flask import g


@app.teardown_testvalue
def remove_test_value():
   test_value = g.pop('test_value', None)
```

It's best not to dwell on `g` for too long. It is useful in some situations, but can quickly become confusing and unnecessary: most of what we need can be handled by the `request()` object.

Additional Route-related Logic

We've seen how to map static and dynamic routes to functions/views using the `@app.route()` decorator. Flask also empowers us with a number of powerful decorators to supplement the routes we create with `.route()`:

- `@app.before_request()`: Defining a function with the `.before_request()` decorator will execute said function *before every request is made.* Examples of when we might use this could include things like tracking user actions, determining user permissions, or adding a "back button" feature by remembering the last page the user visited before loading the next.
- `@app.endpoint('function_name')`: Setting an "endpoint" is an alternative to `@app.route()` that accomplishes the same effect of mapping URLs to logic. This is a fairly advanced use-case which comes into play in larger applications when certain logic must transverse modules known as Blueprints (don't worry if this terminology sounds like nonsense – most people will likely never run into a use case for endpoints).

Error handling

What happens when a user of our app experiences a fatal error? Flask provides us a decorator called `errorhandler()` which serves as a catch-all route for a given HTTP error. Whenever a user experiences the error code we pass to this decorator, Flask immediately serves a corresponding view:

Routes for error codes

```
...

@app.errorhandler(404)
def not_found():
    """Page not found."""
    return make_response(render_template("404.html"), 404)
```

See how we used `make_response()` in conjunction with `render_template()`? Not only did we serve up a custom template, but we also provided the correct error message to the browser. We're coming full-circle!

The above example passes **404** to the `@app.errorhandler()` decorator, but we can pass *any* numerical HTTP error code. We might not know the nuances of *how* a complex app will fail under a load of thousands of users, but unforeseen errors will certainly occur. We can (and should) account for errors before they occur by using the `@app.errorhandler()` decorator to ensure our users won't be left in the dark.

Routes for error codes

```
...

@app.errorhandler(404)
def not_found():
    """Page not found."""
    return make_response(render_template("404.html"), 404)


@app.errorhandler(400)
def bad_request():
    """Bad request."""
    return make_response(render_template("400.html"), 400)


@app.errorhandler(500)
def server_error():
    """Internal server error."""
    return make_response(render_template("500.html"), 500)
```

Additional Route Decorators via Flask Plugins

Lastly, I can't let you go without a least mentioning some of the awesome decorators provided by Flask's powerful plugins. These are a testament to how powerful Flask routes can become with the addition of custom decorators:

- `@login_required` (from Flask-Login): Slap this before any route to immediately protect it from being accessed from logged-out users. If the user *is* logged in, @login_required lets them in accordingly. If you're interested in user account management in Flask, check our post about Flask-Login.
- `@expose` (from Flask-Admin): Allows views to be created for a custom admin panel.

- `@cache.cached()` (from Flask-Cache): Cache routes for a set period of time, ie: `@cache.cached(timeout=50)`.

On The Route to Greatness

Clearly there's a lot we can do with routes in Flask. Newcomers typically get started with Flask armed only with basic knowledge of routes and responses and manage to get along fine. The beauty of Flask development is the flexibility to build meaningful software immediately with the ability to add functionality when the time is right, as opposed to upfront.

As far as routing goes, we've probably covered more route-related logic than what 95% of Flask apps currently need or utilize in the wild. You're well-equipped to build some cool stuff already, so stop reading and get coding!

Flask | Python | Software Development

Todd Birchard

118 Posts | Website | Twitter

Engineer with an ongoing identity crisis. Breaks everything before learning best practices. Completely normal and emotionally stable.

Srividya Iyer
0 points
4 days ago

Maybe this is an older version of flask, but I got this error message. make*response() got an unexpected keyword argument 'headers' I am using flask 1.11.2. The documentation has this example and it works. But it was unclear to me from the make*response document resp = make_response('Test Worked!', 200) resp.headers["Content-Type"] = "application/json" return resp

Ananta Kumar Roy
0 points
3 months ago

The `DELETE` method is also supported in the `app.route` , isn't it?