CODING NOMADS

7) Handling HTTP Errors Lesson

Handling HTTP Errors with Flask

13 min to complete · By Brandon Gigous

Contents

By this stage in the course, you've got a handle on the basics of Jinja templates and routing in Python Flask. But what if something goes wrong? If you type in a bad address or misspell one of your routes, you'll probably get an ominous "Not Found" error. Or even an "Internal Server Error." Let's demystify all that stuff, plus you'll get to make your own template for errors. This will allow your Flask server to handle errors gracefully and communicate what went wrong to the user effectively.

First off, let's clarify what an HTTP error means.

What are HTTP Errors?

HTTP errors are standard responses from a server indicating various types of issues, where each error has a specific status code, like 404 for "Not Found."

Although you may not want to admit it, you've seen them. Errors. Even some you've made yourself. Well, that's okay because the internet is full of **HTTP errors**, but the internet (usually) still works just fine. Websites have all kinds of ways to show you when an error has occurred on their site. But what are these "errors?" Turns out they are just numbers, at least to a server, and your app will be no different. Here's a table of the **status codes** you've probably come across.

Common HTTP Error Codes

Error	HTTP Status Code
Forbidden	403
Not Found	404
Internal Server Error	500

When you hit a "Not Found" error, that just means you went to a page that the server can't give you because, well, it can't find it! Whenever that happens, the HTTP response includes a status code of 404. "Internal Server Error" means the server got into a mess on its own and can't show you whatever you wanted to see, and the HTTP response includes status code 500. The "Forbidden" error? That's forbidden to talk about... but if you really must know, it means you don't have the ability to see it, and the HTTP status code is 403.

Not all servers return the appropriate status code, but for the vast majority of servers, they at least try to return the appropriate error code, with the 500 error being a fallback in case something unexpected happens.

For typical successful requests, the appropriate status code is 200, but you'll almost never see that on an error page because 2xx codes are, by definition, successful!

Build Routes for Error Handling

In this section, you'll build the routes needed to process these errors and the templates to show them. No one likes cryptic messages like "404"! Yikes. We can fix those easily with just a few more with templates and routes.

Now that you know what kinds of errors people encounter, you're ready to define your own error handlers. You've been told about handlers previously, but **error handlers** in Flask are a little different from the ones you've seen so far.

With the following handlers, you'll use the Flask `errorhandler` decorator instead of `route`. This new decorator requires one argument, which is the status code it is tasked with handling. Go ahead and whip out your code editor to `hello.py` and type this in:

```
@app.errorhandler(403)
def forbidden(e):
    error_title = "Forbidden"
    error_msg = "You shouldn't be here!"
    return render_template('error.html',
                           error_title=error_title,error_msg=error_ms


@app.errorhandler(404)
def page_not_found(e):
    error_title = "Not Found"
    error_msg = "That page doesn't exist"
    return render_template('error.html',
                           error_title=error_title,error_msg=error_ms


@app.errorhandler(500)
def internal_server_error(e):
    error_title = "Internal Server Error"
    error_msg = "Sorry, we seem to be experiencing some technical di
    return render_template("error.html",
                           error_title=error_title,
                           error_msg=error_msg), 500
```

Now, you can't just go to your browser and type in `localhost:5000/500`. That won't work like it did with `route`! (You'll get yet another error, go figure.) But do notice that, like with a `route`, it still needs to return a response. These view functions each pass in a couple of keyword arguments to the template. What's different here is that there's also the status code, again, as a second return value.

Responses in Flask, by default, return 200, but if you explicitly need to return a different status code, it can be passed as a second argument. Yes, that means your error handlers have to return the number they are supposed to handle.

Create an Error Template

You're anxious to be able to see the result of your new handlers, aren't you? Of course, of course. To get there, you'll need to create that missing `error.html` template.

Okay, now for the grand showing of your error page. Remember that base template you made earlier? Let's use it to make a cool-looking error page. You can actually kill two birds with one stone, so to speak, as your error handlers are pointing to the same template. This means that you can show both kinds of errors, the "not found" and "internal server error" ones, all in one template file.

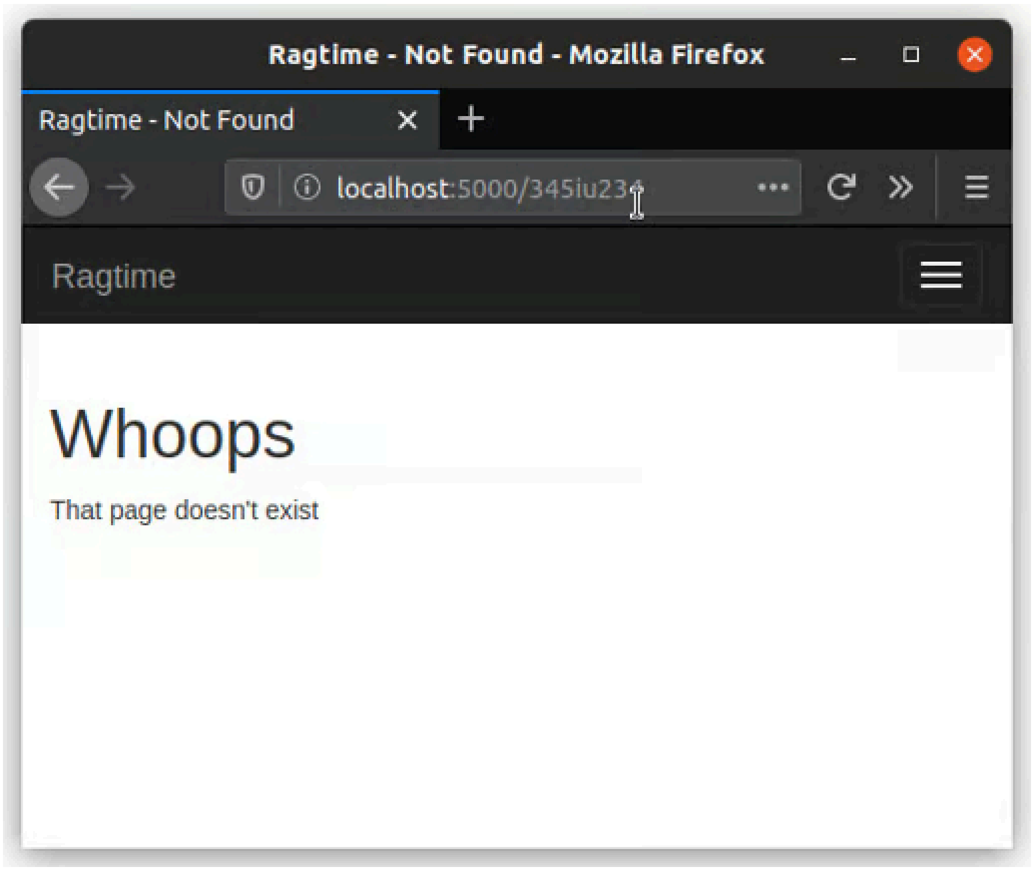
```
{% extends 'base.html' %}

{% block title %}{{super()}} {{error_title}}{% endblock %}

{% block navbar %}
{{super()}}
{% endblock navbar %}

{% block page_content %}
{{super()}}
<h1>Whoops</h1>
<p>{{error_msg}}</p>
{% endblock %}
```

This one is pretty compact and simple, no? The template simply uses everything else from the base template using `extends` and `super()` but adds some extra content to it. In particular, info about the bad stuff that happened. You can see what it should look like below.



Indeed! All the elements from the base template have been carried over to this new error template thanks to template inheritance. The navbar and the styles appear as they did in the user page from before. In this pic, there's also a menu button on the right since the window is physically smaller.

Using the `abort()` Function

Wanna see the other error message? Well, it'll be hard to see it without learning about a new function in Flask.

Whenever you might need to signal to the browser that some mistake happened, you can use the `abort()` helper function. What this does is raise an exception. Think of it like throwing an exception in normal Python code, except an `abort()` is for doing it with a Flask webserver.

We don't need it for our project at this point, but it will become useful later. Here's a quick example: Let's say you have a user loader function called `load_user()` that takes an ID and gives back a user object. If that user doesn't exist, you can signal an error with an `abort()`:

```
from flask import abort

@app.route('/user/<id>')
def get_user(id):
    user = load_user(id)
    if not user:
        abort()(404)
    return f"<h1>Hello, {user}!</h1>"
```

In the case that a user with that ID doesn't exist, the view function will skip returning a response. That's because the `abort()` function raises an exception. The number passed in is the status code. When the `abort()` function is called, Flask raises an HTTPException that is then caught by the error handler registered for that specific status code.

If you were to use this code and hit the `abort()`, your code would then render your `error.html` template with a "Not Found" message because the `page_not_found()` view function would get called.

Summary: Handling HTTP Errors with Flask

In this lesson, you've expanded your knowledge on handling HTTP errors in Flask, learning how to manage different types of responses when things don't go as planned. Here's what you've learned so far:

- **HTTP errors** are standard responses from a server indicating various types of issues, where each error has a specific status code, like 404 for "Not Found."
- You implemented **error handlers** using the `@app.errorhandler` decorator in Flask to intercept specific status codes and provide custom responses.
- You saw how to return a status code along with a rendered template.
- You created an `error.html` template using template inheritance to display error messages.
- You learned about Flask's `abort()` function, which can be used within a view to abruptly end the request with a specific error code.