CODING NOMADS

8) Making Forms with Flask-WTF Lesson

Handling Web Forms with Flask-WTF

46 min to complete · By Brandon Gigous

Contents

In this lesson, you'll learn all about forms and how to manage them as a Flask developer using the Flask-WTF library.

Forms Are Everywhere

Forms—those blank bits of information you fill in once in a while—are everywhere! Each year, I have to fill out tax forms and give them to the Internal Revenue Service. They are definitely not as fun as web forms, which are on pretty much all of the websites you visit.

You've been getting familiar with how to make new routes in Flask, how to flesh out those routes with templates, and how to present any errors should something go wrong. But all of those things are the *user* getting information from the *server*. There must be some use in telling the website *some* information about you, right? Like your email and password to log in, your search terms, or what your favorite K-pop band is.

What are Web Forms?

That's where **web forms** come in. Web forms are as old as the internet, and you can even make them with basic HTML. A user enters their information, and once they submit it, the data goes from the web browser to the server in the form of a **POST request**. The POST request contains the user information and gives the server access to that information. In your case, the server will be Flask. These POST requests will contain things like login info, personal details, and favorite flavor of ice cream. POST requests will be covered more later in this lesson, but think of a POST request like the user sending a postcard (submitting the form) to a mailbox (the server).

Making form components with HTML is straightforward, but then there's validating. For example, "wood" isn't exactly a flavor of ice cream... and hang on, how do we get all this functionality using Python?

For that there is Flask-WTF! Don't let the name fool you, as it makes the making of forms pretty darn easy and painless. You'll learn about this tool in the next lesson.

Getting Started with Flask-WTF

If you're reading this, then you're probably curious about Flask-WTF and why it's called that, or you just happen to be taking the CodingNomads Flask course and are about to learn how to make forms with the Flask-WTF framework, or both. Either way, you've come to the right place!

Flask-WTF did not get its name from people screaming, "WTF!" when working with it. Instead, it comes from the WTForms library, which is what the Flask-WTF extension is based on. Making forms will instead cause you to scream, "Flask-WTF FTW (for the win)!" because it's actually quite a pleasant experience.

The days of tediously writing HTML code for generating forms is over. Flask-WTF gives you the ability to define forms *in Python* and instantly render them in a template. You can

define various validators, including *custom* ones that can, for example, make sure you didn't type something like `2 + 2 = chair`

To get you off to a running start, you'll obviously have to install Flask-WTF, but that's easy:

```
(env) $ pip install flask-wtf
```

Assuming that went without a hitch, you might be thinking about the next step. "Oh, now I have to initialize Flask-WTF, right? Like I did for that Bootstrap thing. Y'know, `Bootstrap(app)`"

Hold your horses, bucko. Flask-WTF doesn't actually need to be initialized! But there's still one quick thing we need to do. If you want complete secrecy about your favorite flavor of ice cream, to only confide it to the server and no one else, the framework has to ensure the data you send is *encrypted*. To enable that secrecy, you need a **secret key**. You can configure this right in your `hello.py` file:

```
app = Flask(__name__)
app.config['SECRET_KEY'] = "keep it secret, keep it safe" # This is
```

So what's this `app.config` thing? Your `Flask` application object has a built-in, general-purpose dictionary to store all kinds of configuration variables. This dictionary is used by Flask extensions, and you can even utilize it for your own application's purposes. You can define values in the `config` dictionary at the start of your program, or you can import values straight from a file or from the environment which you'll do later.

Did you type in that exact string, `"keep it secret, keep it safe"`? While it might be fine for your first Flask project, you should *never* use the same key for one of your more serious apps, nor should you upload your secret key to source control (i.e., don't commit it to GitHub!).

More will be covered about how to keep your secret key *a secret*, but for now you can continue with this string or with another totally made-up string, like `"T0mmy wrot3 a`

`poem_about butterflies"` or even better, something obnoxiously difficult like `"93c9fd51b6f68d8cf88185656b4f2eb3815819"`, just make sure it's not that exactly, but something you generated yourself or with a password generator.

Why all this rambling about secrets and security? Well, it's important your or your users' information is safe. By defining your secret key, you will 1) make Flask-WTF happy and 2) protect your forms against nasty cross-site request forgery (CSRF) attacks. Flask-WTF uses that security key to generate security tokens for every form, which are then stored in the **user session**. That precious, private information in the user session is only accessible with the secret key, so that's why it's so secret!

How does one *form* a form with Flask-WTF, though? The answer to this question shall be answered in the next section.

- Weekly 1-on-1 screen share meetings with your personal, professional mentor
- Constant, personal, in-depth support 7 days a week via Discord
- Accountability, feedback, encouragement, and success

Get Mentorship

Use Flask-WTF to Make Forms

Alright so let's make your very first Flask-WTF form! This one's gonna be simple so you know what's going on. You'll be asking the user a *very difficult* question: "What is your name?"

In your `hello.py` file, you'll import a few objects and define a form class which is derived from `FlaskForm`:

```
from flask_wtf import FlaskForm
from wtforms import StringField, SubmitField
from wtforms.validators import DataRequired

class NameForm(FlaskForm):
    name = StringField("What is your name?", validators=[DataRequire
    submit = SubmitField("Submit")
```

"Wow, that's it?!" I think I heard you say that when you finished typing it. Yes, you've already defined your form and its fields! The fields are defined as *class* variables, and

their values are various `*Field` classes, the `*` just being a convenient way of saying `StringField`, `SubmitField`, `SelectField`, `DateField`, etc. The `*Field` classes are imported from the `wtforms` module, and you could, of course, import more of those if your form needs additional kinds of fields.

Let's break it down a little further. Your form, `NameForm`, has a text field called `name`. It also has a submit button called `submit`. A fancy web form wouldn't be as fancy without validators, and Flask-WTF makes checking inputs easy. Validators can be declared within a list inside `*Field` constructors, and one example is the `DataRequired` validator. The `name` field has this validator, meaning that the information in a `NameForm` cannot be submitted without putting a name in that field. There are also plenty more validators should you need them.

What's that? You want to see the form really bad, huh? You want to see it work before your eyes? To do that, you'll have to render it first, but that part might be easier than the above! Nevertheless, head on over to the next lesson for your next mission.

Create a Template to Render Your Form

Great, you've just *defined* your form and its fields, but as you may have realized, that doesn't equate to *seeing* it on the screen as a real, live, breathing form. It's now time to create the template that will *show* your new form on a webpage.

Rendering Forms using Jinja

First thing's first, it's best to know how to render forms "the hard way" in HTML. This particular quick lesson in rendering forms will be a little easier than starting from scratch as it will include how to use Jinja placeholders to render parts of your `NameForm`. If you want, you can follow along by putting this code in your `index.html` template file. Note that later in this lesson, you'll replace your form rendering "the hard way" into a much easier way.

Let's assume a few things to set the stage.

1. You've defined your form along with a few fields.
    - You did this above.
2. You have a view function that passes in your form to a template via `render_template()`. The argument passed in to Jinja is called `form`.

- You will make this view function in the next lesson.
- The argument containing the form doesn't have to be named `form`, but it's used in the following examples.

3. The template that you rendered the form to exists.
    - In your case this might be `index.html`.

These steps apply generally to any `FlaskForm` you want to render using HTML and Jinja. So with that, let's dive in!

```
<form method="POST">
    {{ form.hidden_tag() }}
    {{ form.name.label }} {{ form.name() }}
    {{ form.submit() }}
</form>
```

The first part of rendering a form in HTML is to start with a `<form>` tag with a `method` attribute. You know a little about POST, so this shouldn't be too much of a surprise. Remember the long spiel about security? There's a little more to do than just render the form. The `form.hidden_tag()` is needed for Flask-WTF to implement CSRF protection, even though it's invisible on the page when the form is rendered. You'll want to include it because it's important!

To actually show the form and not render just the invisible parts, next is to display the fields. For fields that have labels, you'd start with rendering the label first, and then the field itself is rendered by calling the field. Yes, form fields are callable! When called, they do whatever is needed to render themselves to HTML. For the submit button, all that's needed is to simply render the button.

You can even define HTML `id` attributes for fields in your form so that you can define CSS styles for them. For example, to do this for the `name` field, you'd add a keyword argument to its call.

```
{{ form.name.label }} {{ form.name(id="my-field-id")}}
```

Bootstrapping Your Forms

Doing it this way may have you thinking, "Gee whiz! I don't want to spend all this time to render a form, *especially* if it's for something like an extensive questionnaire about accordions!", as you cover your face with your hands in fear.

Presented to you here is the antidote, which you can use as a complete replacement for all that tedious rendering above. Just add water!... I mean, Flask-Bootstrap!

```
{% import "bootstrap/wtf.html" as wtf %}
{{ wtf.quick_form(form) }}
```

Bootstrap once again saves the day with its own slick form styles, but even more noteworthy here is that Flask-Bootstrap gives you a sweet helper function called `quick_form()`, which renders an entire Flask-WTF in one go. No more rendering the form fields individually! Makes life easy, doesn't it?

To make our life easier, we're gonna use Flask-Bootstraps predefined CSS styles. That way, we don't have to cringe at a form that looks like it came from the Internet of the 90's, we can have a cool modern look right outta the box (`import "bootstrap/wtf.html" as wtf`, `wtf.quick_form(form)`, `form` is a variable, and we don't have to define the form fields individually)

Putting It All Together

Whether or not you have already tried rendering your `NameForm` from "the hard way" to using the Flask-Bootstrap `quick_form()` helper function, go ahead and pull up `index.html`. To include the `NameForm` for the Ragtime app, it will look something like this:

```
{% extends 'base.html' %}
{% import "bootstrap/wtf.html" as wtf %}

{% block title %}Ragtime{% endblock %}

{% block page_content %}
<h1>Welcome to Ragtime, {% if name %}{{name}}{% else %}Anonymous{% e
```

```
    <p>Enjoy your stay.</p>
    {{ wtf.quick_form(form) }}
    {% endblock page_content %}
```

Whenever this template is rendered for the first time, it will show a message, "Welcome to Ragtime, Anyonymous!" plus a nice little greeting. It will also show your newly rendered form you defined! Coolio.

But before you can see it, you'll need to change your `index()` view function. Back to the code above, the only thing that looks a little fishy is the conditional starting with `{% if name %}` . . . . . `name` . . . . . `form.name` . . . . can you possibly show a name if you haven't even entered one into the `form.name` field yet?

This is a mystery that will have to be solved in the next section, so read on!

Get and Post Requests

In this section, you'll be learning about how to bring your form to life with a view function to render your template. On the last episode of "Rendering Forms: Crime Scene Investigation" AKA the last lesson, the `name` variable was in the template with seemingly no explanation. No one with the name `name` can answer it because he or she or they haven't put a name in the `NameForm` yet! Who or what is behind this crime against course-taking students?" Let's stay calm, I'm sure it *might* be answered shortly...

Viewing Your Form

So you have your `NameForm`, you have your template to render the form, and now it's finally time to see it in its natural habitat: on your web browser. For that you'll need to edit your `index()` view function to make your form and pass it to the `index.html` template.

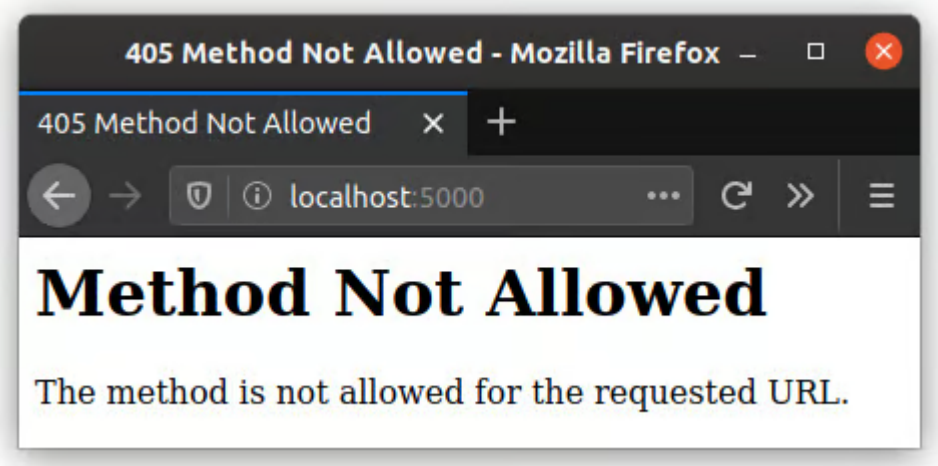
```
@app.route('/')
def index():
    name = None
    form = NameForm()
    if form.validate_on_submit():
        name = form.name.data
```

```
        form.name.data = ''
    return render_template('index.html', form=form, name=name)
```

Aha! And there's `name`! The evidence is piling up, yet at the same time `name` doesn't leave a trace: its value is `None`. This `name` must gets its *true* value somewhere. Let's keep going...this detective stuff is fun.

More seriously though, The form gets created here as well, but if the `index.html` template is rendered without a successfully submitted form, the `if validate_on_submit():` block will not execute. It returns `True` only when the form is submitted without errors and is ready to be processed, meaning the data in the form passed all validation by the fields validators. Either way, the form is created and can be rendered whether or not the form submission is successful.

Alright, let's try this out! Get your environment ready and do a `flask run` and pull up your web browser.



Wait a sec, what's with this "Method not allowed" message? Of course it's allowed! You just put the form in the view function or method or whatever and told it to render! Why won't it work? Oh, hold on, it's talking about another kind of "method." Add this to your `route` decorator:

```
@app.route('/', methods=['GET', 'POST'])
```

View functions are only `GET` by default. By adding the `POST` method to the route, it indicates to Flask that the `index()` view function is a handler for both `GET` and `POST`

requests. That's exactly what you'll need in order to use a form, as was foretold!

Requests and GET vs. POST

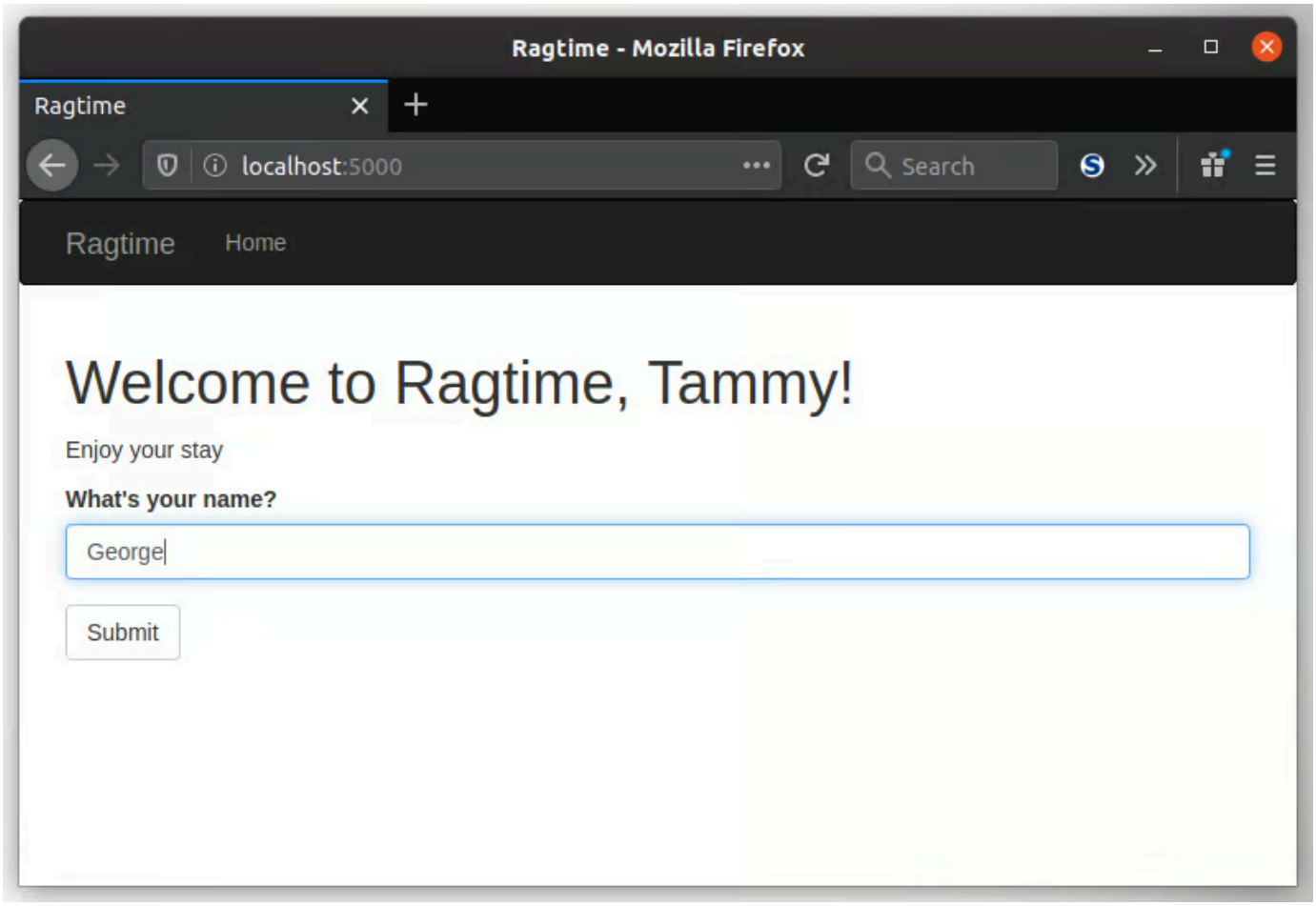
Little did you know, when the user makes a request, Flask exposes a `request` object that you can access at any time within your view functions. If you did happen to know that, go find someone and ask them to pat you on the back because your course creator said so.

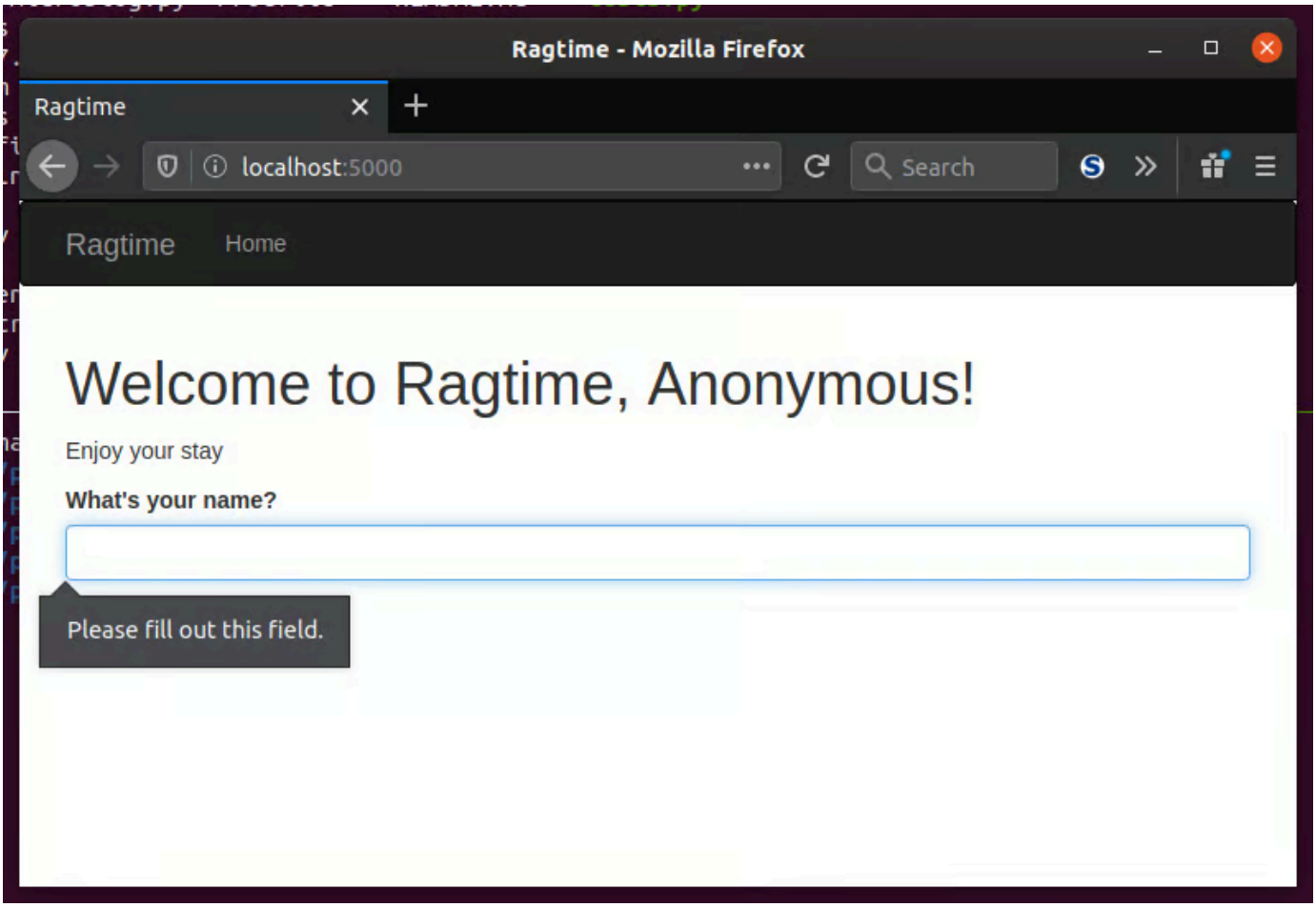
Anyway, this `request` object holds a lot of goodies as attributes in the form of dictionaries. They expose all the information sent by the web browser or client. One of those is `request.form` and contains all the form fields submitted with the request. Remember, the request happens *before* the appropriate view function gets called. In your case, the `form` variable in `index()` grabs the data from `request.form` without you having to do much of anything else.

GET requests *can* be used to send form data to the server, but in practice, this is almost never the case. Anyone who wants to expose their submitted information in a URL is free to do so, which is really the only way to handle form data with a GET request. The obvious alternative is to use a POST request. The difference here is that GET requests don't have a body. POST requests do, and the request body contains any form information the client may have submitted. Combine that with cryptographic magic and secret keys discussed earlier, and you have a much more secure way to exchange information with the server!

Take another look at the `index()` view function. When navigating to the index page `/` of your Flask webapp, the view function gets called after a GET request so that it can render the form and anything else in the template. At this point, the `validate_on_submit()` function returns `False`, because there is no body and therefore no form data. In other words, `request.form` is empty. Once the user on the other end clicks the submit button, *only then* does the request, then become a POST request. The `validate_on_submit()` doesn't return `False` right away, but it could when processing and *validating* the form inputs. Once all input is validated, `name` gets assigned the form input data from the `form.name` field, and that gets passed into your template!

If you haven't already, go ahead and try out your form. You'll see that you can easily put a name in, hit submit, and be presented with a nice welcome message with your or someone else's name in it. But if you forget to put a name in, the form will tell you once the page loads again.


Ragtime - Mozilla Firefox
Ragtime
localhost:5000
Search
Ragtime
Home
Welcome to Ragtime, Tammy!
Enjoy your stay
What's your name?
George
Submit


Ragtime - Mozilla Firefox
Ragtime
localhost:5000
Search
Ragtime
Home
Welcome to Ragtime, Anonymous!
Enjoy your stay
What's your name?
Please fill out this field.

I think this crime scene investigation has come to a close. Finally, `name`'s alibi has come to light and you are ready to move on to `redirect`s and message flashing. Well done, detective! The next mystery: how to prevent a warning from popping up once refreshing the page...

Redirects, Sessions and Message Flashing

You finally have a form displayed on your index page, but there are a few tweaks you can make to bring the user a more natural experience. These are: redirects to avoid a confusing warning and message flashing to notify the user about various information.

Redirects

There's a problem with your current index page. When you enter a name, submit it, then refresh the page manually, you'll see an obscure message that will ask you to confirm before it potentially submits the form again. You get that warning because because browsers repeat the last request they sent upon a manual refresh, which might mean submitting a form *again*.

Why is this bad? Think back to whenever you made a big purchase online, perhaps some fancy speakers or a robotic vacuum cleaner. You put your credit card info and click submit, but the darn page keeps loading and won't stop! In your desperation, you refresh the browser. Two things could happen: either the server implemented a **redirect** to reload the page without resubmitting the form, or it didn't, and you get a surprise on your credit card bill for twice the amount you paid for. Yikes! Let's avoid that kind of scenario altogether by putting in a redirect of your index page.

You never want to repeat a form submission if the user refreshes. You can prevent that with a `redirect()` call. How would you do this? Let's first think about the simpler fix, which you don't have to edit any files for. The simpler fix is to just add a `redirect` at the end of the `if` block of `index()`, like so:

```
    if form.validate_on_submit():
        # ...
        return redirect(url_for('index'))
```

That would work, but... what's that? Oh, the `url_for()`? This Flask helper function is a convenient way to generate URLs from within your app. When you type in the name of a view function, Flask will look it up in the app's URL map that Flask creates (discussed in Section 5). So typing `'index'` in here will generate `'/'`, which Flask knows is the index page since you decorated `index()` with `route('/')`. This is great to use if you can never decide or remember which path a view function should handle, because it will always be correct!

Indeed, URL generation is the way of the future! Well, maybe not, but it definitely helps to use in your `redirect()` call, which takes a URL to redirect *to*. Why would you redirect to the same place? That redirect command is clearly going right back to the index page!

It all comes back to the `methods`. When the form loads the first time, before submitting the form, it's a GET request. Once the form is submitted successfully, it's a POST request. Then the user manually refreshes the page. What happens? Read these prophetic words once more:

> *browsers repeat the last request they sent upon a manual refresh*

That's right, the POST request is repeated. "Submit the form again!" the user is inadvertently telling the browser. To thwart the user from causing his or her own potential destruction, your Flask server can send a redirect after the form is submitted instead of responding back normally. A redirect is, in fact, a special type of response that contains a URL instead of an HTML string, like `render_template()` ultimately does. Redirects tell the browser to issue a GET request for the URL indicated. So, the redirect's superpower is that it can transform a POST request into a GET request!

User Sessions

Hang on, now there's another problem! Once the redirect happens, what *was* a POST request that could access the form data then becomes a GET request that *cannot*. Because remember, GET requests have no body, and therefore no access to form data! Oh no!

Have no fear, **user sessions** are here! In fact, they've always been here. The user session is some private storage available to each connected client. Think of it like a bucket of data just for you that allows the application to "remember" things between requests. You, or rather the browser you use, owns the bucket but the application can fill it with data and use that data later. Flask gives you access to the user session in every view function

via the `session` object. It acts just like a Python dictionary, and with it you can store away information that you want the application to remember. Let's try it out below along with a redirect!

No More Amnesia or Double Takes

The new and improved `index()` view function in `hello.py` will look like this:

```
@app.route('/', methods=['GET', 'POST'])
def index():
    form = NameForm()
    if form.validate_on_submit():
        session['name'] = form.name.data
        return redirect(url_for('index'))
    return render_template('index.html', form=form, name=session.get
```

Now your users never have to worry about double form submissions, nor do they have to live in fear that the application will never remember their name! Both issues are solved with this implementation. The `name` from before has been replace with `session['name']`, which just means the application puts `name` in your data bucket with the value of whatever you submitted in the form. Then same value is passed into `render_template()` with a call to `session.get()`.

As you can try out yourself with the redirect, hitting refresh after submitting the name doesn't trigger an alert. On top of that, the way you've done it, we can actually visit a different page, then come back and see our name is still there!

Message Flashing

NEWS FLASH. Sometimes your users forget what they were doing (me) or you need to bring their attention to something important. Or they made a mistake, which happens to the best of us. Showing them some status updates here and there doesn't hurt, and in some cases, can add some personality to your web app. The best part with Flask? You don't need an extension for this because it's built in!

Let's try it out:

```
from flask import flash

@app.route('/', methods=['GET', 'POST'])
def home():
    form = NameForm()
    if form.validate_on_submit():
        session['name'] = form.name.data
        flash('Great! We hope you enjoy the community')
        return redirect(url_for('home'))
    return render_template('home.html', form=form, name=session.get(
```

That was pretty easy. Once the form is validated, the `flash()` function queues a message to be displayed on the next request sent back to the user. Your next goal is to show this new message, and any other flash messages, at the top of the page after the `flash()` message is. While you can *make* messages with `flash()` all you want, without rendering them you'll never see them!

For that, you'll need to add it to a template. (Starting to see the pattern?) Now, you might already have your `index.html` template up, and you're thinking, "Oh, okay, I'll just add it there! What do I have to do?" Hold up, buttercup! If you do that, you'll have to add the rendering for the flash messages in *every other template*. You remember how tedious rendering forms the hard way was, don't you?*

You can save yourself the trouble by adding the flash-message-rendering functionality to the base template, `base.html` . By taking advantage of template inheritance once again, you can display any queued flash messages on any page of your web app. Since now you understand a bit better, onto the template additions:

```
{% block content %}
<div class="container">
    {% for msg in get_flashed_messages() %}
        <div class="alert alert-warning">
            <button type="button" class="done" data-dismiss="alert">
            {{ msg }}
        </div>
    {% endfor %}
    {% block page_content %}
```

```
        {% endblock page_content %}
    </div>
    {% endblock content %}
```

Flask provides for you a convenient way of getting those queued flash messages through the function `get_flashed_messages()`. What it returns is simply a list of messages. Again, there could be other messages queued that you want the user to see also, so that's why you should use a `for` loop to iterate over them.

For now you should only be generating one each time the user submits the `NameForm`. Making use of Bootstrap styles, you can make a popup that shows the message with an 'x' [illegible] `get_flashed_messages()` [illegible] its queue of messages, and the same message will not appear again unless `flash()` is called again for said message.

Now you are a master of redirection, user sessions, and message flashing!

Register Here

Summary: Handling Web Forms with Flask-WTF

You've learned quite a lot about handling forms in Flask with the Flask-WTF library. Let's recap your new knowledge — in this lesson you:

- Discovered how web forms enable further user interaction on websites, allowing information to be sent from the browser to the server via POST requests.
- Got to know Flask-WTF, a Flask extension that simplifies form creation, validation, and rendering.
- Installed Flask-WTF and set up a secret key in your Flask app configuration to secure form data and protect against CSRF attacks.
- Learned to define forms with field classes and validators in Python.
- Prevented repeated form submissions by implementing redirects.
- Used Flask's user session objects to store user-related data between requests.
- Played around with message flashing to provide feedback to users after they interact with your forms.

Now that you have these skills, you're better equipped to create interactive and secure web applications with Flask. Keep experimenting and building, and these concepts will soon feel like second nature!

Previous Next → Video: Making Forms