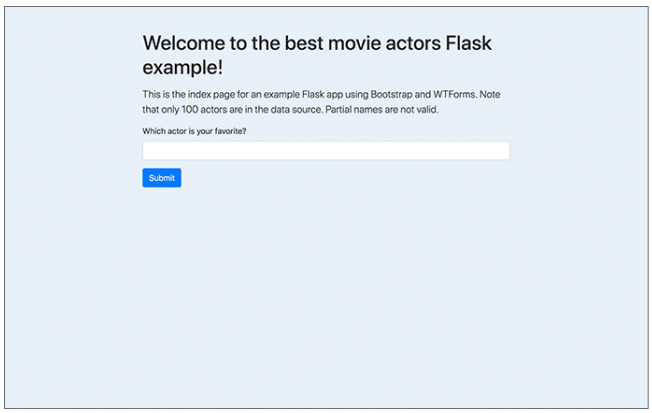
# Flask: Web Forms

Previous:

1. Flask intro: A very simple Flask app
2. Flask, part 2: Values in routes; using an API
3. Flask templates: Write HTML templates for a Flask app
4. Flask: Deploy an app: How to put your finished app online

Code for this chapter is here.

In the **Flask Templates** chapter, we built a functioning Flask app. In this chapter, we'll explore how to add functional web forms to a similar app.





**Flask forms app example** (*actors_app*):

- Live app
- Code

## Introduction

Flask has an extension that makes it easy to create web forms.

WTForms is "a flexible forms validation and rendering library for Python Web development." With **Flask-WTF**, we get WTForms in Flask.

- WTForms includes security features for submitting form data.
- WTForms has built-in validation techniques.
- WTForms can be combined with Bootstrap to help us make clean-looking, responsive forms for mobile and desktop screens.

Read the documentation for Flask-WTF.

## Setup for using forms in Flask

We will install the **Flask-WTF** extension to help us work with forms in Flask. There are many extensions for Flask, and each one adds a different set of functions and capabilities. See the list of Flask extensions for more.

In Terminal, change into your Flask projects folder and **activate your virtual environment** there. Then, at the command prompt — where you see `$` (Mac) or `C:\Users\yourname>` (Windows )—

```
pip install Flask-WTF
```

We will also install the **Flask-Bootstrap4** extension to provide Bootstrap styles for our forms.

```
pip install Flask-Bootstrap4
```

This installation is done *only once* in any virtualenv. It is assumed you already have Flask installed there.

- Flask-WTF docs
- More details in WTForms docs
- Flask-Bootstrap docs
- An *alternative* is Bootstrap Flask — but that is NOT used here

## Imports for forms with Flask-WTF and Flask-Bootstrap

You will have a long list of imports at the top of your Flask app file:

```
from flask import Flask, render_template, redirect, url_for
from flask_bootstrap import Bootstrap
from flask_wtf import FlaskForm
from wtforms import StringField, SubmitField
from wtforms.validators import DataRequired
```

Note as always that Python is case-sensitive, so upper- and lowercase must be used exactly as shown. **The fourth line will change** depending on **your form's contents.** For example, if you have a SELECT element, you'll need to import that. See the complete list of WTForms form field types.

## Set up a form in a Flask app

After the imports, these lines follow in the app script:

```
app = Flask(__name__)

# Flask-WTF requires an encryption key - the string can be anything
app.config['SECRET_KEY'] = 'C2HWGVoMGfNTBsrYQg8EcMrdTimkZfAb'

# Flask-Bootstrap requires this line
Bootstrap(app)
```

Flask allows us to set a "secret key" value. You can grab a string from a site such as RandomKeygen. This value is used to prevent malicious hijacking of your form from an outside submission.

Flask-WTF's `FlaskForm` will automatically create a secure session with CSRF (cross-site request forgery) protection *if this key-value is set*. **Don't publish the actual key on GitHub!**

You can read more about `app.config['SECRET_KEY']` in this StackOverflow post.

### Configure the form

Next, we configure a form that inherits from Flask-WTF's class `FlaskForm`. Python style dictates that a **class** starts with an uppercase letter and uses camelCase, so here our new class is named `NameForm` (we will use the form to search for a name).

In the class, we assign each form control to a unique variable. This form has only one text input field and one submit button.

**Every form control** must be configured here.

```
class NameForm(FlaskForm):
    name = StringField('Which actor is your favorite?', validators=[DataRequired()])
    submit = SubmitField('Submit')
```

Learn more about classes in Python here.

If you had **more than one form** in the app, you would define more than one new class in this manner.

Note that `StringField` and `SubmitField` were **imported** at the top of the file. If we needed other form-control types in this form, we would need to import those also. See a list of all WTForms field types.

Note that several field types (such as `RadioField` and `SelectField`) must have an option `choices=[]` specified, after the label text. Within the list, each choice is a pair in this format: `('string1', 'string2')`.

WTForms also has a long list of validators we can use. The `DataRequired()` validator prevents the form from being submitted if that field is empty. Note that these validators must also be imported at the top of the file.

## Put the form in a route function

Now we will use the form in a Flask route:

*../python_code_examples/flask/actors_app/actors.py*

```
27  @app.route('/', methods=['GET', 'POST'])
28  def index():
29      names = get_names(ACTORS)
30      # you must tell the variable 'form' what you named the class, above
31      # 'form' is the variable name used in this template: index.html
32      form = NameForm()
33      message = ""
34      if form.validate_on_submit():
35          name = form.name.data
36          if name.lower() in names:
37              # empty the form field
38              form.name.data = ""
39              id = get_id(ACTORS, name)
40              # redirect the browser to another route and template
41              return redirect( url_for('actor', id=id) )
42          else:
43              message = "That actor is not in our database."
44      return render_template('index.html', names=names, form=form, message=message)
```

A crucial line is where we assign our configured form object to a new variable:

```
form = NameForm()
```

We must also pass that variable to the template, as seen in the final line above.

Be aware that if we had created **more than one** form class, each of those would need to be assigned to a unique variable.

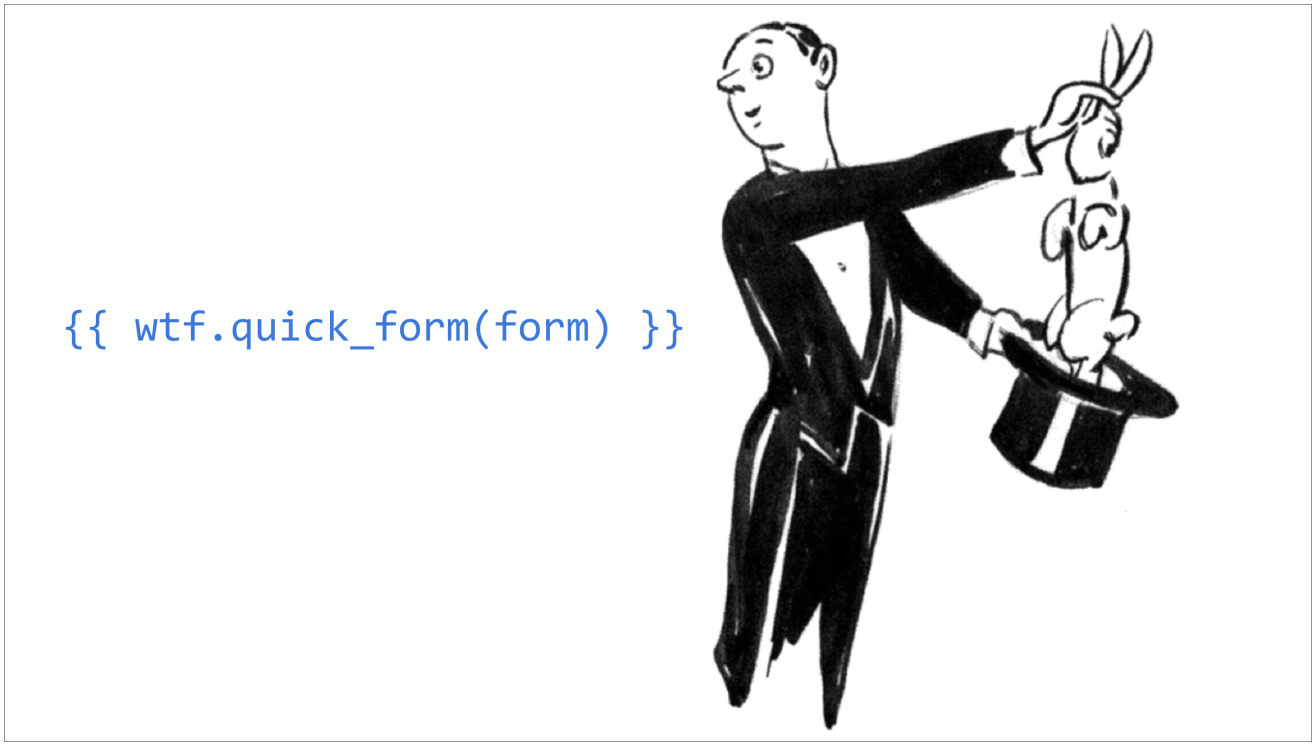
## Put the form in a template

Before we break all that down and explain it, let's look at the code in the template *index.html*:

*../python_code_examples/flask/actors_app/templates/index.html*

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

{% block styles %}
{{ super() }}
        <style>
                body { background: #e8f1f9; }
        </style>
{% endblock %}


{% block title %}
Best Movie Actors
{% endblock %}


{% block content %}

<div class="container">
  <div class="row">
    <div class="col-md-10 col-lg-8 mx-lg-auto mx-md-auto">

      <h1 class="pt-5 pb-2">Welcome to the best movie actors Flask example!</h1>

      <p class="lead">This is the index page for an example Flask app using Bootstrap
and WTForms. Note that only 100 actors are in the data source. Partial names are not
valid.</p>

      {{ wtf.quick_form(form) }}

      <p class="pt-5"><strong>{{ message }}</strong></p>

    </div>
  </div>
</div>

{% endblock %}
```

**Where is the form?** This is the amazing thing about Flask-WTF — by configuring the form as we did *in the Flask app*, we can generate a form with Bootstrap styles in HTML using nothing more than the template you see above. **Line 27 is the form.**




Note that in the Flask route function, we passed the variable `form` to the template *index.html*:

```
return render_template('index.html', names=names, form=form, message=message)
```

So when you use `wtf.quick_form()`, the argument inside the parentheses **must** be the *variable* that represents the form you created in the app.

```
form = NameForm()
```

We discussed the configuration of `NameForm` above.

## A quick note about Bootstrap in Flask

There's more about this in the **Resources** section at the bottom of this page — but to summarize briefly:

- You pip-installed Flask-Bootstrap4 in your Flask virtual environment.
- You wrote `from flask_bootstrap import Bootstrap` at the top of the Flask app file.
- Below that, you wrote `Bootstrap(app)` in the Flask app file.
- In any Flask template using Bootstrap styles, the top line will be:
  `{% extends 'bootstrap/base.html' %}`

That combination of four things has embedded Bootstrap 4 in this app *and* made `wtf.quick_form()` possible.

There's an excellent how-to video (only 9 minutes long) about using Bootstrap styles in Flask if you want to separate the **forms** information from the Bootstrap information in your mind. You can, of course, use Flask-Bootstrap4 *without* the forms!

## Examining the route function

Before reading further, try out a working version of this app. The complete code for the app is in the folder named actors_app.

1. You type an actor's name into the form and submit it.
2. If the actor's name is in the data source (ACTORS), the app loads a detail page for that actor. (Photos of bears 🐻 stand in for real photos of the actors.)
3. Otherwise, you stay on the same page, the form is cleared, and a message tells you that actor is not in the database.

*../python_code_examples/flask/actors_app/actors.py*

```
27  @app.route('/', methods=['GET', 'POST'])
28  def index():
29      names = get_names(ACTORS)
30      # you must tell the variable 'form' what you named the class, above
31      # 'form' is the variable name used in this template: index.html
32      form = NameForm()
33      message = ""
34      if form.validate_on_submit():
35          name = form.name.data
36          if name.lower() in names:
37              # empty the form field
38              form.name.data = ""
39              id = get_id(ACTORS, name)
40              # redirect the browser to another route and template
41              return redirect( url_for('actor', id=id) )
42          else:
43              message = "That actor is not in our database."
44      return render_template('index.html', names=names, form=form, message=message)
```

First we have the route, as usual, but with a new addition for handling form data: `methods`.

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

Every HTML form has two possible methods, `GET` and `POST`. `GET` simply requests a response from the server. `POST`, however, sends a request **with data attached** in the body of the request; this is the way most web forms are submitted.

**This route needs to use both methods** because when we simply *open the page*, no form was submitted, and we're opening it with `GET`. When we submit the form, this same page is opened with `POST` if the actor's name (the form data) was not found. Thus we cannot use only one of the two options here.

```
def index():
    names = get_names(ACTORS)
```

At the start of the route function, we get the data source for this app. It happens to be in a list named `ACTORS`, and we get just the names by running a function, `get_names()`. The function was imported from the file named *modules.py*.

```
form = NameForm()
message = ""
```

We assign the previously configured form object, `NameForm()`, to a new variable, `form`. This has been discussed above.

We create a new, empty variable, `message`.

```
if form.validate_on_submit():
    name = form.name.data
```

`validate_on_submit()` is a built-in WTForms function, called on `form` (our variable). **If it returns True,** the following commands and statements in the block will run. If not, the form is simply not submitted, and invalid fields are flagged. It will return True if the form was filled in and submitted.

`form.name.data` is the contents of the text input field represented by `name`. Perhaps we should review how we configured the form:

```
class NameForm(FlaskForm):
   name = StringField('Which actor is your favorite?', validators=[DataRequired()])
   submit = SubmitField('Submit')
```

That `name` is the `name` in `form.name.data` — the contents of which we will now store in a new variable, `name`. To put it another way: The variable `name` *in the app* now contains whatever the user typed into the text input field on the web page — that is, the actor's name.

```
36          if name.lower() in names:
37              # empty the form field
38              form.name.data = ""
39              id = get_id(ACTORS, name)
40              # redirect the browser to another route and template
41              return redirect( url_for('actor', id=id) )
42          else:
43              message = "That actor is not in our database."
```

This if-statement is specific to this app. It checks whether the `name` (that was typed into the form) matches any name in the list `names`. If not, we jump down to `else` and text is put into the variable `message`. If `name` DOES match, we clear out the form, run a function called `get_id()` (from *modules.py*) and — **important!** — open a *different route* in this app:

```
return redirect( url_for('actor', id=id) )
```

Thus `redirect( url_for('actor', id=id) )` is calling a different route here in the same Flask app script. (See *actors.py*, lines 46-55.) The `redirect()` function is specifically for this use, and we **imported** it from the `flask` module at the top of the app. We also imported `url_for()`, which you have seen previously used within templates.

As far as **using forms with Flask** is concerned, you don't need to worry about the actors and their IDs, etc. What is important is that **the route function** can be used to *evaluate the data sent from the form*. We check to see whether it matched any of the actors in a list, and *a different response* will be sent based on match or no match.

Any kind of form data can be handled in a Flask route function.

You can do *any* of the things that are typically done with HTML forms — handle usernames and passwords, write new data to a database, create a quiz, etc.

The final line in the route function calls the template *index.html* and passes three variables to it:

```
return render_template('index.html', names=names, form=form, message=message)
```

# Conclusion

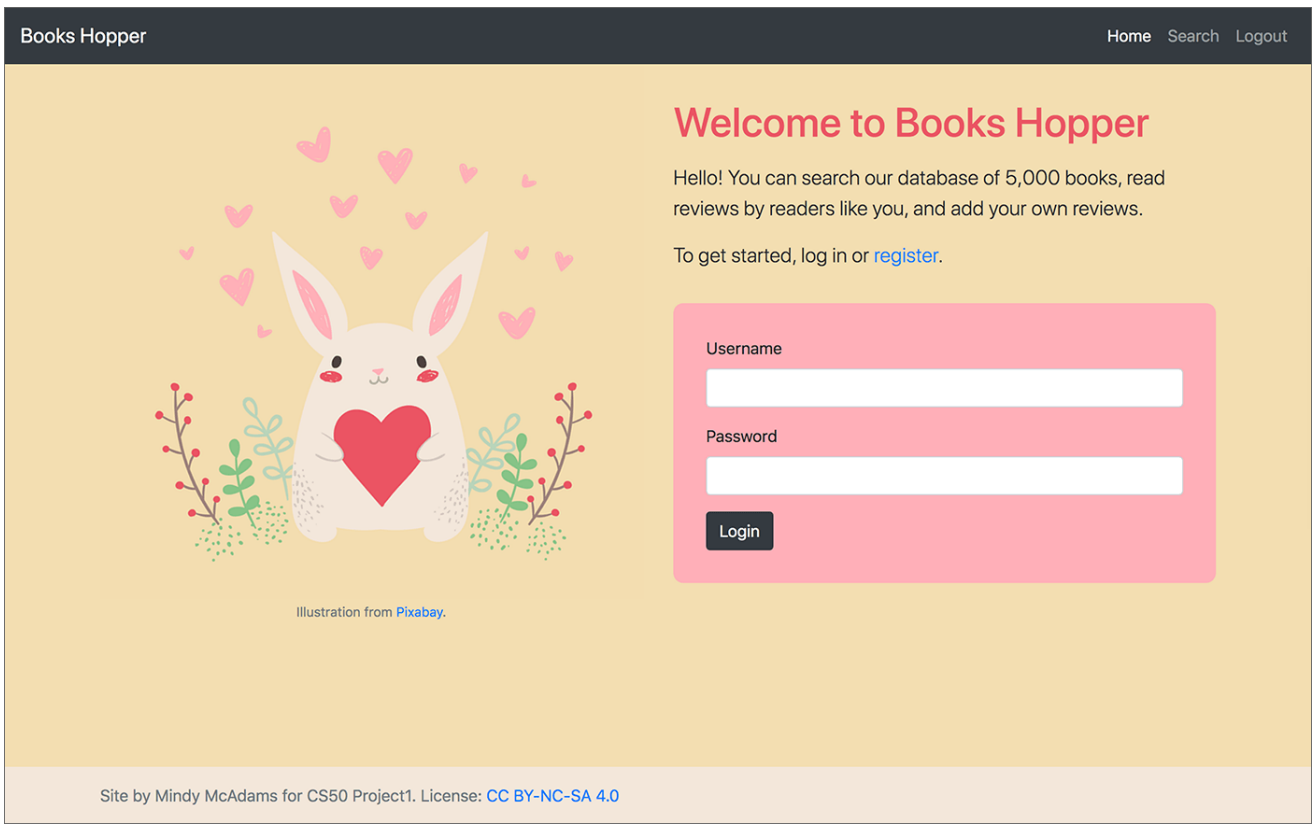
**Flask-WTF** provides convenient methods for working with forms in Flask. Forms can be built easily and also processed easily, with a minimum of code.

Adding **Flask-Bootstrap** ensures that we can build mobile-friendly forms with a minimum amount of effort.

Note that it is possible to build a customized form layout using Bootstrap 4 styles in a Flask template, or to build a custom form with no Bootstrap styles. In either case, you cannot use `{{ wtf.quick_form(form) }}` but would instead write out all the form code in your Flask template as you would in a normal HTML file. To take advantage of WTForms, you would still create the form class with `FlaskForm` in the same way as shown above.

An example is the demo Flask app Books Hopper, which includes four separate Bootstrap forms:

- a login form
- a registration form
- a search form
- a form for writing a book review and selecting a rating




Bootstrap 4 was used in all templates in the Books Hopper app, but Flask-Bootstrap was not.

Important

You are using Bootstrap 4 in Flask if you installed with `pip install Flask-Bootstrap4`. In early 2018, Bootstrap 4 replaced Bootstrap 3. The differences are significant.

## Resources

- Sending form data — how web browsers interact with servers; request/response
- Flask-WTF documentation
- Complete WTForms documentation
- Flask-Bootstrap documentation
- About Flask-Bootstrap templates

.