Forms

Contents

Overview

This module will cover the implementation of forms in a Flask application using the WTForms (and Flask-wtf) libraries.

Creating a Form

We define the structure of a form by creating a class that inherits from the `FlaskForm` class. Within the class, we define the input `fields` (elements) of the form.

For example:

```
class DogForm(FlaskForm):
    name = StringField("Name")
    age = IntegerField("Age")
    breed = SelectField("Dog Breed", choices=[
        ("collie", "Collie"),
        ("retriever", "Retriever"),
        ("pug", "Pug")
    ])
    submit = SubmitField("Submit")
```

The above class will allow us to create a form with four fields on oue web page:

- a text field with the label "Name".
- an integer field with the label "Age".
- a drop-down box with the options "Collie", "Retriever" and "Pug".
- a submit button with the label "Submit".

The first argument provided to each field will be the field's 'label'. We can use this to display information about the field with our HTML templates.

Once we define the class, we can instantiate an object based on it in the route function we want to display the form:

```
@app.route('/add_dog', methods = ['GET'])
def add_dog():
    # instantiate the DogForm object
    form = DogForm()
    # pass object to Jinja2 template
    return render_template('add_dog.html', form=form)
```

We then add the `form` object as an argument to the `render_template()` function, allowing us to pass it through to our template.

We can then use Jinja2 substitution to add our fields into the web page:

```
<body>
    <h1>Add a new dog:</h1>
    <p>
        <div class='form'>
            <form method='POST' action=''>
                <!-- hidden_tag() is required for form validation -->
                {{ form.hidden_tag() }}
                <!-- Each field displays its label followed by the input field
itself -->
                {{ form.name.label }} {{ form.name }}
                <br>
                {{ form.age.label }} {{ form.age }}
                <br>
                {{ form.breed.label }} {{ form.breed }}
                <br>
                <!-- form.submit is a button -->
                {{ form.submit }}
                <br>
            </form>
        </div>
    </p>
</body>
```

> Note: You need to have both `form.hidden_tag()` embedded in the HTML *and* the app's `SECRET_KEY` value set in order for the form page to display. If either are missing, the app will throw an error.

This will allow us to display our form, but currently it doesn't actually do anything when the user presses **Submit**.

`SubmitField`s will send a `POST` request to the current page with the data inputted into the form. With this in mind, we can use conditional logic to change the functionality of the `/add_dog` route based on whether it receives a `GET` or `POST` request:

```
@app.route('/add_dog', methods = ['GET','POST'])
def add_dog():
    # instantiate the DogForm object
    form = DogForm()

    # checks is the http request is a post request
    if request.method == 'POST':
        # checks if the form passes validation
        if form.validate_on_submit():
            # adds the dog to the database
            dog = Dog(
                name = form.name.data,
                age = form.age.data,
                breed = form.breed.data
            )
            db.session.add(dog)
            db.session.commit()
            # redirects the user to the home page
            return redirect(url_for('home'))

    # pass object to Jinja2 template
    return render_template('add_dog.html', form=form)
```

With the above code, the `add_dog()` function will get the form for the user if they send a `GET` request, and will create a new dog entry and add it to the database if the form is submitted with a `POST` request.

> Note: Because form submission sends a `POST` request, we have to make sure that `POST` methods are allowed by the route.

Notice how we can retrieve the data from the submitted form by accessing the `form` object's field attributes:

```
name = form.name.data
```

Fields

Fields define the type of user input that a form provides. In WTForms they are classes which contain various attributes.

Field types	Description	Optional Parameters	Example
StringField	A field which takes in string input.	`size=()`, `maxlength=()`	`name = StringField('Name', maxlength=20)`
IntegerField	A field which takes in integer input.	none	`number = IntegerField('', size=20)`
BooleanField	A field which takes in true or false input.	`false_values=None`	`bool = BooleanField()`
DateField	A field which takes in date inputs.	`format='%Y-%m-%d'`	`date = DateField()`
DateTimeField	A field which takes in date and time inputs	`format='%Y-%m-%d %H:%M:%S'`	`datetime = DateTimeField()`
DecimalField	A field which takes in decimal input.	`places=2`, `rounding=None`	`decimal = DecimalField()`
SubmitField	A field which allows for checking of a given submit button being pressed.	none	`input type="submit"`

Field types	Description	Optional Parameters	Example
SelectField	A field which allows us to make use of the `choices` parameter which is simply a list of value and label pairs. This allows us to give users a list of options to interact with on the webpage.	`choices=[], validate_choice=True or False`	`language = SelectField('Programming Language', choices=[('cpp', 'C++'), ('py', 'Python'), ('text', 'Plain Text')])`

Field attributes

Each field has the following attributes:

Attribute	Example	Description
data	firstname.data	The data submitted for a field named `firstname`.
label	firstname.label	The label of a field named `firstname`.
errors	firstname.errors	Any errors thrown at submission.
type	firstname.type	The type of field `firstname` belongs to, such as `StringField`.

Populating the SelectField

Sometimes you may want your `SelectField` drop-down boxes to be populated dynamically based on what information has been added to the database.

For example, consider a web app with two database entities: cars and owners. After creating an owner using a web form, you could navigate to the car form and see the new owner appear in the `SelectField` drop-down box.

We don't know what owners are going to be added to the database in the future, so we can't hardcode the options into the class definition.

Instead, we can query the database and add the `SelectField` choices at the point we instantiate the form. For example:

```
# Create a form object
form = CarForm()

# Query for all owners in table
owners = Owners.query.all()

# Populate the select field with owners
for owner in owners:
    form.owner.choices.append(
        (owner.id, f"{owner.first_name} {owner.last_name}")
    )
```

The choices for a `SelectField` are defined as a list where each item is a tuple. These tuples are in the format:

```
(<value>, <label>)
```

Where `<value>` is what data will be sent when the form is submitted, and `<label>` is what's displayed in the dropdown box on the webpage.

In the for loop, we are appending to that list in order to populate it with the owners it finds from the database.

```
for owner in owners:
    form.owner.choices.append(
        (owner.id, f"{owner.first_name} {owner.last_name}")
    )
```

This example uses each owner's `id` (the primary key) as the information that the form actually submits, and using the owner's `first_name` and `last_name` attributes to format a string to display the owner's full name.

Tutorial

This tutorial gets you to make a basic Flask app that displays a form and shows you how to make use of the inputted information.

Setup

This tutorial assumes you are working on an Ubuntu VM, at least version `18.04 LTS`.

First, install apt dependencies:

```
sudo apt update
sudo apt install python3 python3-venv python3-pip
```

Create a directory named `flask-forms` and make it your current working directory:

```
mkdir flask-forms && cd $_
```

We now need to create a Python virtual environment to install our pip requirements in. Create a new virtual environment named `venv` and activate it:

```
python3 -m venv venv
source venv/bin/activate
```

Finally, we need to install Flask, Flask-WTF and WTForms with pip:

```
pip3 install flask flask-wtf wtforms
```

Create the Application

Run the following commands to build our basic file structure:

```
mkdir templates
touch app.py templates/home.html
```

First we need to import the following into `app.py` in order to use our installed pip packages:

```
from flask import Flask, render_template, request
from flask_wtf import FlaskForm
from wtforms import StringField, SubmitField
```

Next, we can instantiate our Flask app object, set our `SECRET_KEY` value and define a form class within `app.py`:

```
app = Flask(__name__)

app.config['SECRET_KEY'] = 'YOUR_SECRET_KEY'

class BasicForm(FlaskForm):
    first_name = StringField('First Name')
    last_name = StringField('Last Name')
    submit = SubmitField('Add Name')
```

The `SECRET_KEY` is required for WTForm's CSRF protection and needs to be set.

Next, let's create our page template in `home.html`:

```
<html>
    <head>
        <title>Simple form</title>
    </head>
    <body>
        <div class='form'>
            <form method='POST' action=''>
                {{ form.hidden_tag() }}
                {{ form.first_name.label }} {{ form.first_name }}
                <br>
                {{ form.last_name.label }} {{ form.last_name }}
                <br>
                {{ form.submit }}
                <br>
            {{ message }}
            </form>
        </div>
    </body>
</html>
```

The HTML below will display the form using Jinja2 templating. The sections wrapped in double curly braces will be substituted for the HTML required to display the form's field.

Finally, we need to create our home page's route. Paste the following code into `app.py` after the form's class definition:

```
@app.route('/', methods=['GET', 'POST'])
@app.route('/home', methods=['GET', 'POST'])
def register():
    message = ""
    form = BasicForm()

    if request.method == 'POST':
        first_name = form.first_name.data
        last_name = form.last_name.data

        if len(first_name) == 0 or len(last_name) == 0:
            message = "Please supply both first and last name"
        else:
            message = f'Thank you, {first_name} {last_name}'

    return render_template('home.html', form=form, message=message)

if __name__ == '__main__':
     app.run(debug=True, host='0.0.0.0')
```

Run the Application

Run the application with the command:

```
python3 app.py
```

When the user sends a `GET` request to the server, this code will display the form to the user as formatted in the `home.html` template.

When the user enters their details and presses **Submit**, the `if request.method == 'POST':` statement will return `True` and write a message.

There's a basic check to see if the user has omitted entering either name, in which case it will display an error prompt; otherwise, it will display a personalised thank you message based on what the user has submitted.

Notice how we need to allow `POST` requests so that the user can send data to our Flask application.

Clean Up

To stop your Flask application running, navigate back to your terminal and press `Ctrl+C`. You should now have control over your terminal again.

To deactivate the virtual environment, run:

```
deactivate
```

If you wish to delete the virtual environment, run:

```
rm -rf venv
```

Exercises

Update the form to prompt the user for:

- Their date of birth.
- Their favourite number.
- Their favourite food out of pizza, spaghetti or chilli (or whatever foods you like!).

Generate a personalised username based on this information and display it on the web page.