Constructing Forms With Flask-WTF

Introduction

Flask is an agnostic Python microframework for developing web applications. By agnostic I mean that it is not opinionated as to a preferred way of doing things. This is in contra-distinction to Django and Ruby on Rails which impose conventions as well as a preferred way to develop web applications. There is a Django way of doing things and a Rails way of doing things. However since Flask is not an opinionated framework there is not a Flask way of doing things.

In this article I will discuss form handling in Flask and in particular *Flask-WTF* forms. This is an intermediate guide, so the reader is presumed to have a basic familiarity with Flask, the Jinja template engine as well as rudimentary form management in Flask. Since Flask is a minimal web framework you import the functionality that you need into your web application instead of using the functionality already bundled into a “big” framework such as Django or Rails.

Though not mandatory, I recommend that you set up a Python virtual environment with a 3.5+ version of Python and then follow along. In Computer Science, doing is always the best way to learn.

Create a basic Flask application in your virtual environment as follows. Firstly install Flask in your virtual environment (make sure that you have activated the virtual environment):

```
pip3 install flask
```

Next create a file called *application.py* in the root of your virtual environment and place the following code in it:

```
# application.py

from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_form():
   return 'Hello, Form!'

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

This is all that is needed to create a bare-bones Flask web application. The function *hello_form* is called a view function and it handles the '/' route. At the risk of being pedantic, a view function returns output (typically a string) that is sent to the browser and rendered by it.

In order to run the *hello_form* application, enter *python application.py* on the command line of your virtual environment. This will start the Flask development server on 127.0.0.1:5000. And if you type http://127.0.0.1:5000 in the location bar of your browser you should see: *Hello, Form!*.

The Jinja template engine expects HTML templates to reside in the templates folder of your application root, so go ahead and create this folder.

Creating Flask-WTF Forms

Most developers dislike working with forms. Form management has a well deserved reputation for having a cumbersome syntax. *Flask-WTF* is an extension designed to make form handling in Flask a much more pleasant experience. *Flask-WTF* is a wrapper around the *WTForms* package. *WTForms* is an agnostic forms module built into Flask. To use *Flask-WTF* install this extension first:

```
pip3 install flask-wtf
```

Let us now build a form for user registration. User registration forms are ubiquitous in web applications. The user submits a registration form with his or her email address and a password. The server checks a database to determine whether the email already exists. If it does the user registration fails. Otherwise registration succeeds. In this section, I will implement the registration form using Flask-WTF. In the next section, I will implement a validation routine that tests for the uniqueness of the email address.

The process of rendering a *Flask-WTF* form involves three steps. Firstly the form is defined by specifying a form class. The next step involves the specification of a template to render the form. Lastly, a route to the form is specified and a view function to handle the route is defined.

In *Flask-WTF* a form is represented by a class. Each field in the form is an object in the class and each field object can have attributes including validators. Validators check whether field values input by an user are valid. In the root directory of your application specify the Registration class as follows:

```
from flask_wtf import Form
from wtforms import TextField, PasswordField, SubmitField
from wtforms import validators, ValidationError
from wtforms.validators import InputRequired, EqualTo


class Registration(Form):
  email = TextField("email", [validators.InputRequired("Please enter
your email"), validators.Email('invalid email')])

  password = PasswordField('password',
           [validators.InputRequired("please enter a password")])
  confirm_password = PasswordField("confirm password",
           [validators.InputRequired("password"),
validators.EqualTo("password", "passwords must match")])


  submit = SubmitField("Submit")
```

Notice that every *Flask-WTF* form class must inherit from the *Form* class. Our Registration class has three variables (form fields) called email and password and confirm_password. The email field is a TextField type and has an array of validators as it's second parameter. The *InputRequired* validator returns an error if the field is empty. The *Email* validator performs rudimentary email validation using a regular expression. The password and confirm_password fields are of type PasswordField. This means that user input into these fields will not be displayed. Flask-WTF supports a large number of form field types. You can review the available field types in the official *WTForms* documentation at: https://wtforms.readthedocs.io/en/stable/fields.html.

In order to render the form, we have to specify a template for the form. So go to the templates sub-directory and create the *registration.html* template file as follows:

```
<!-- templates/registration.html -->

<!doctype html>
<html>
<body>
<h2 style = "text-align: center;">User Registration</h2>

{% for message in form.email.errors %}
  <div><p style="color:red;">{{ message }}</p></div>
{% endfor %}

{% for message in form.confirm_password.errors %}
  <div><p style="color:red;">{{ message }}</p></div>
{% endfor %}

<form action = "http://127.0.0.1:5000/registration" method=post>
  <fieldset>

     {{ form.hidden_tag() }}
     {{ form.email.label }}<br>
     {{ form.email }}

     <br><br>

     {{ form.password.label }}<br>
     {{ form.password }}
     <br><br>

     {{ form.confirm_password.label }}<br>
     {{ form.confirm_password }}

     <br><br>

     {{ form.submit }}

  </fieldset>
</form>

</body>
</html>
```

You will notice that this form contains a hidden_tag field. We will discuss this field in a later section.

Next create a *success.html* file in the templates sub-directory. This page will be displayed if the user registration succeeds.

```
<!-- templates/success.html -->
<!doctype html>
<html>
  <body>
    <h2>You have been registered</h2>
  </body>
</html>
```

Finally we specify a route and a view function to handle the route for the registration form in *application.py*. The changes made to this file are in bold type.

```
# application.py

from flask import Flask, render_template,request
from registration import Registration
app = Flask(__name__)

@app.route('/')
def hello_form():
   return '<H1>Hello, Form!</H1>'

@app.route('/registration', methods = ['GET', 'POST'])
def RegisterUser():
   form = Registration()
```

```
    if request.method == 'POST':
        if form.validate() == False:
          return render_template('registration.html', form = form)
        else:
          return render_template('success.html')
    elif request.method == 'GET':
      return render_template('registration.html', form = form)


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

Here we have imported the Registration module that we created earlier. In addition the *request* and *render_template* modules have been imported from flask. The view function *RegisterUser* contains all of the logic to route to the form and render it. If a validator fails then the form is re-rendered and if validation succeeds the application renders the *success.html* page.

Restart the development server and traverse to http://127.0.0.1:5000/registration. The registration form will be displayed. You can play with the form and in particular with the validation that it implements.

Custom Field Validators

WTForms provides a standard set of form field validators (recall that Flask-WTF is a wrapper around this package). Here is heuristic summary of some of the more prominent validators:

DataRequired(message=None)	Tests for the presence of data in a field. A field with only white space is false.
Length(min=-1, max=-1, message=None)	Validates the length of a text form field.
Email(message=None)	Provides very basic validation of email addresses using a regular expressions.
IPAddress(ipv4=True, ipv6=False, message=None)	Validates an IP address.

You will want to refer to https://wtforms.readthedocs.io/en/stable/validators.html for the documentation on the standard validators provided by *WTForms*.

For any reasonably complex web application you will need to provide validators other than those provided by *Flask-WTF*. Flask-WTF makes the specification of these custom validators very easy.

The user registration form that we have developed so far has a glaring deficiency. This form requires a validator which determines whether the provided email address already exists in the application's database. Since WTForms does not provide such a validator we will have to write our own application specific validator. Lets see how this custom validator can be implemented with *Flask-WTF*.

We create an *uniqueEmail* module in the application root folder. This module contains a function *IsUnique* that checks whether the email exists in the database. In the implementation of this function we will abstract the database functionality by throwing an exception if the email consists of an odd number of characters.

```
#testemail.py

from wtforms import ValidationError

def IsUnique(form, field):
    if (len(field.data) % 2 == 1):
       raise ValidationError('email already exists')
```

Next we add the *isUnique* validator to the registration class. The changes in *registration.py* are in bold type:

```
# registration.py

from flask_wtf import Form
from wtforms import TextField, PasswordField, SubmitField
from wtforms import validators, ValidationError
from wtforms.validators import InputRequired, EqualTo
from testemail import IsUnique

class Registration(Form):
  email = TextField("email", [validators.InputRequired("Please
          enter your email"), IsUnique])

  password = PasswordField('password', [InputRequired("please
             enter a password")])
  confirm_password = PasswordField("confirm password",
          [EqualTo('password', message='Passwords must match')])


  submit = SubmitField("Submit")
```

Now restart the Flask development server and traverse to the *http://127.0.0.1:5000/registration* URL. The Flask-WTF form will be displayed and you can test the email uniqueness field validation.

Protecting Against CSRF Attacks

CSRF or Cross Site Request Forgery is major web application attack vector. One of the major advantages of using *Flask-WTF* to generate forms is that it provides CSRF protection by default.

A CRSF attack occurs when an attacker induces a victim into making an unintended request. This request may involve changing a password, an email address or transferring money from a bank account. CSRF relies upon a general property of how browsers handle cookies. When a browser sends a request to a domain, it includes all of the cookies pertaining to the domain.

Lets consider a typical CSRF attack. Suppose we have an user who has obtained elevated privileges on a domain called federalbank.ca. The domain identifies this user through a session cookie that it has sent to the user. Now suppose that an attacker crafts a URL that transfers some funds to him and tricks the user into clicking this URL link. For example this link could have been sent to the victim via an email. If the victim clicks on the link, the browser will send a request to the domain federalbank.ca that includes all of the cookies pertaining to the federalbank.ca domain. The server will note that the victim has elevated privileges and thus transfer the requested funds to the attacker. To reiterate, the session cookie identifying the user with elevated privileges has been sent to the server in the request object and the server identifies the request as coming from a user with elevated privileges. This is an example of a cross-site forgery attack.

It is not required that the victim click on a link. For example, a malicious page may contain an image that contains the link. The link will be automatically sent to the server when the user opens the page.

To provide CSRF protection, the Flask application specifies an encryption key. In *application.py* and below the *app = Flask(__name__)* line, enter:

```
app.config['SECRET_KEY'] = 'a very long string'
```

Before sending a form to the server, *Flask-WTF* requests the server to send it an unique encrypted token. The server creates this token from the secret key and delivers it to the user (the user's browser and not the attacker's machine). This token is then incorporated into a hidden text field in the form and the browser sends the form to the server. The hidden field enables the server to verify the authenticity of the form request. The request forged by the user fails since the attacker does not have the encrypted token.

Implementing CSRF protection with HTML and Javascript only is a significantly complex task and a major reason why *Flask-WTF* forms are to be preferred over HTML forms with Javascript.

Use Cases For Flask-WTF

So when should you use *Flask-WTF* in preference to plain jane HTML forms? There are three major use cases:

- *Flask-WTF* provides CSRF protection by default. For a HTML form implementation you will have to implement CSRF protection in raw Javascript, a non-trivial task.
- HTML does not support the creation of dynamic forms since it has no control flow structures
- *Flask-WTF* enables the specification of custom validators for form fields. HTML forms do not provide for any validation other than that baked into HTML.