Form Validators

Contents

Overview

This module discusses WTForms validators that we can use in our Flask applications to check the validity of user inputs.

Validators

Validators are classes used for sanity checks on the data being passed through forms. For example, in an 'Email' field, we only want strings in the correct format (hello@world.com) to be submitted.

Every WTForms field has a `validators` parameter that expects a **list** of validators:

```
class MyForm(FlaskForm):
    name = StringField('Name', validators=[DataRequired(),
Length(min=2,max=15)])
```

As the validators parameter receives a list of validators, we can assign multiple validators to one field. In the above code, make sure that the field must be filled in (`DataRequired()`) and has a minimum character length of 2 and a maximum character length of 15 (`Length(min=2,max=15)`).

Built-in Validators

WTForms comes packaged with a range of validators ready to be imported and used. The `validators` parameter when defining a field is a list of the different validators to be applied.

Each validator can take arguments for different options. All built-in validators have the `message` argument which defaults to `None`. This is the message presented to the user if the validator rejects the submitted data.

```
class myForm(FlaskForm):
    name = StringField('Name', validators=[DataRequired()])
```

Built-in validators require importing from the `wtforms.validators` package:

```
from wtforms.validators import DataRequired
```

The WTForms validators documentation provides a complete list of built-in validators and more details about their options.

Here are a number of useful built-in validators:

DataRequired

```
class myForm(FlaskForm):
    name = StringField('Name', validators=[DataRequired()])

# The form cannot be submitted without data in the relevant field.
```

Email

```
class myForm(FlaskForm):
    email = StringField('Email', validators=[Email()])

# Field can only submit data in email format.
```

EqualTo

```
class myForm(FlaskForm):
    password = PasswordField('Password', validators=[
        DataRequired(),
        EqualTo('confirm_password')
    ])
    confirm_password = PasswordField('Confirm password')

# 'password' and 'confirm_password' fields must match for submission.
```

Length

```
class myForm(FlaskForm):
    first_name = StringField('First name', validators = [Length(min=2, max=15)])

# First name must be between 2 and 12 characters long.
```

Custom validators

Sometimes we may require the data submitted to follow a format of our choosing. For this, we can implement a **custom validator**.

Custom validators are methods within our form class that will raise a `ValidationError` object on a condition of our choosing.

`ValidationError` is a class from `wtforms.validators`.

The simplest way of implementing a custom validator is an in-line validator: one which is defined as a method in the form class. In-line validators must be named in the format `validate_fieldname`

Consider a registration form with a `username` field, and that there are some reserved usernames that we do not want the users to use. For this example, let's use `admin` as a forbidden username.

```
class myForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired()])

    def validate_username(self, username):
        if username.data.lower() == 'admin':
            raise ValidationError("Invalid username, please choose another")
```

The validator checks if the data entered into the username field is equal to `admin` when converted to lower case (to make it ignore case).

The most robust way of implementing a custom validator is as a class with an initialiser and a call method.

We can also define our custom validators as classes. Here is the same validator implemented as a class:

```
class checkAdmin:
    def __init__(self, message=None):
        if not message:
            message = 'Please choose another username'
        self.message = message

    def __call__(self, form, field):
        if field.data.lower() == 'admin':
            raise ValidationError(self.message)
```

Validating Forms

The FlaskForm class includes a method named `validate_on_submit()`. This will return `True` if all validators for that form have been validated successfully on submission.

This is useful for logic implementation and routing.

If `validate_on_submit()` is `False`, you can find the error messages in the Fields `errors` attributes:

```
form.username.errors
```

You can then display these on your web page by passing them through to a Jinja2 template.

Tutorial

In this tutorial, we will go through implementing a validator with a customizable message and list of forbidden usernames. To do this, we will be defining our validator as a class.

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-form-validators` and make it your current working directory:

```
mkdir flask-form-validators && 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
```

Creating the Application

First, we need to create our app structure like so:

```
.
├── app.py
└── templates
    └── home.html
```

Run these commands to create it:

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

Next, inside `app.py`, set up the app object:

```
from flask import Flask, render_template
from flask_wtf import FlaskForm
from wtforms import StringField, SubmitField # We will only use StringField and
SubmitField in our simple form.
from wtforms.validators import DataRequired, Length, ValidationError

app = Flask(__name__)
app.config['SECRET_KEY']='SOME_KEY' #Configure a secret key for CSRF protection.
```

Next, create the validator:

```
class UserCheck:
    def __init__(self, banned, message=None):# Here we set up the class to have
the banned and message attributes. banned must be passed through at declaration.
        self.banned = banned
        if not message:
            message = 'Please choose another username' # If no message chosen,
then this default message is returned.
        self.message = message

    def __call__(self, form, field):
    # Here we define the method that is ran when the class is called. If the
data in our field is in the list of words then raise a ValidationError object
with a message.
        if field.data.lower() in (word.lower() for word in self.banned):
            raise ValidationError(self.message)
```

Next, create a simple form consisting of a StringField labelled `Username`, with the validators DataRequired, Length, and our custom one UserCheck. Let's also create a button with SubmitField.

```
class myForm(FlaskForm):
    username = StringField('Username', validators=[
        DataRequired(),
        # We call our custom validator here, and pass through a message to
override the default one. We pass through the list of banned usernames as a
list.
        UserCheck(message="That username is not allowed", banned =
['root','admin','sys']),
        Length(min=2,max=15)
        ])
    submit = SubmitField('Sign up')
```

This is the app route, the function that defines the behaviour of our app. When the form is submitted, it will render the `home.html` page with the username variable set to the form submission. Otherwise, it will render it as an empty string.

```
@app.route('/', methods=['GET','POST'])
def postName():
    form = myForm()
    if form.validate_on_submit():
        username = form.username.data
        return render_template('home.html', form = form, username=username)
    else:
        return render_template('home.html', form = form, username="")
```

Now add the following so that it can be run from the command line:

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

▶ Click here to show the full code for app.py

Finally, create the `home.html` file:

```
<html>
    <head>
        <title>Simple form validator</title>
    </head>
    <body>
        <h1>Choose a username!</h1>
        <div class='form'>
            <form method='POST' action=''>
                <!--The hidden tag here is necessary for CSRF protection (always
include in forms)-->
                {{ form.hidden_tag() }}
                <!--We display the username field label and the field itself-->
                {{ form.username.label }}
                {{ form.username }}
                <!--Here, we want our page to catch and display any errors
raised. The if statement checks if a ValidationError object exists and if so,
present the message-->
                {% if form.username.errors %}
                <div class='error'>
                    {% for error in form.username.errors %}
                        <span>{{ error }}</span>
                    {% endfor %}
                </div>
                {% endif %}
                <br>
                {{ form.submit }}
            </form>
        </div>
        <div>
            <p> You chose: {{ username }} </p>
        </div>
    <body>
</html>
```

Start the Application

Run the application with the command:

```
python3 app.py
```

Now go to your application via your machine's public IP address on port `5000`.

Enter in a username that is not in the form's `banned` list to check the app is working correctly. You should see the name displayed under the text field.

Then input one of the usernames in the `banned` list provided to the `UserCheck()` class, such as 'admin`. You should see your custom error appear on the webpage.

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

Create another custom validator that stops the user adding special characters (`!@"'£$/\`, etc.) to their username.