FLASK & WEBFORMS

INTRODUCING WTFORMS

- HTML form input handling and validation
- Many web-application frameworks will associate forms with database models
- But sometimes you will have a form that is not associated with a database

- WTForms allows you to create form fields for HTML, but allows customization using templates
- Allows separation of presentation and validation code

KEY CONCEPTS OF WTFORMS

- Forms are the core container of WTForms. Forms represent a collection of fields, which can be accessed on the form dictionary-style or attribute style

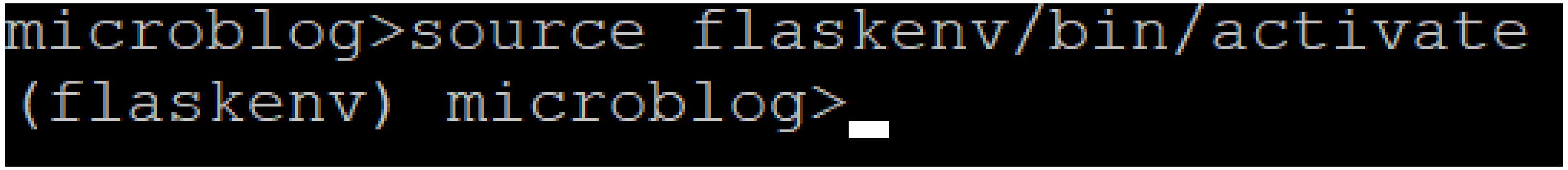
- Most of the work is done using Fields. Each field represents a *data type* and the field handles coercing form input to that datatype. For example, *IntegerField* and *StringField* represent two different data types.

- Every field has a Widget instance. The widget's job is to render the field in HTML.

- Fields contain a list of Validators. These specify the validation rules you want to apply

INTEGRATING WTFORMS WITH FLASK

- Go into your `venv` for microblog using the command

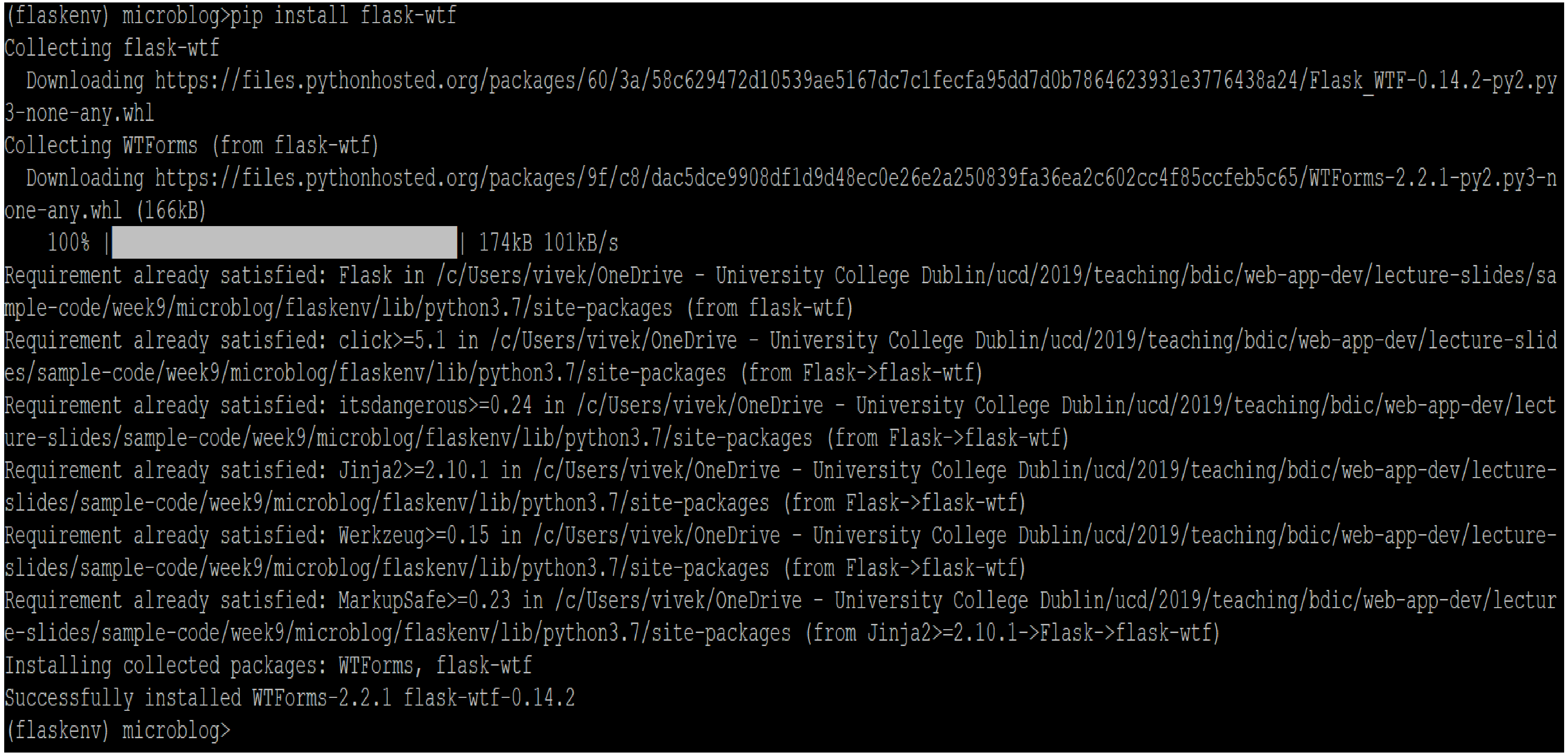


```
microblog>source flaskenv/bin/activate
(flaskenv) microblog>_
```

- install the Flask-WTF extension, using the command

```
pip install flask-wtf
```

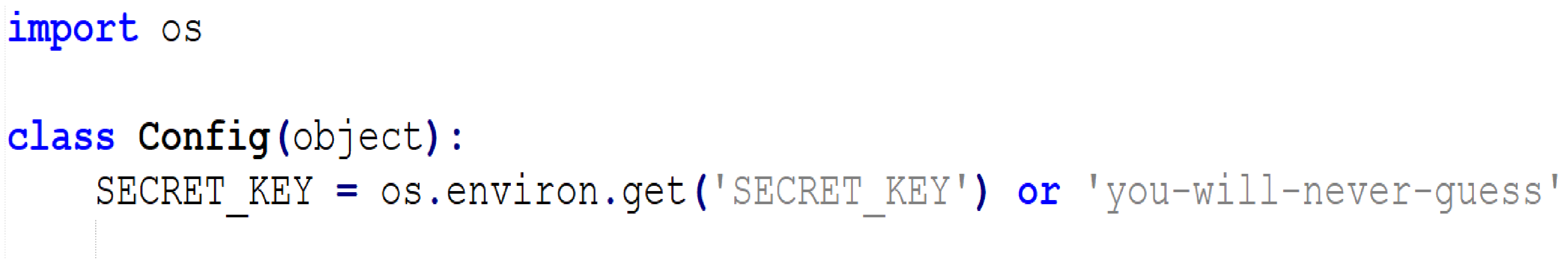
YOU SHOULD SEE THIS



```
(flaskenv) microblog>pip install flask-wtf
Collecting flask-wtf
  Downloading https://files.pythonhosted.org/packages/60/3a/58c629472d10539ae5167dc7c1fecfa95dd7d0b7864623931e3776438a24/Flask_WTF-0.14.2-py2.py
3-none-any.whl
Collecting WTForms (from flask-wtf)
  Downloading https://files.pythonhosted.org/packages/9f/c8/dac5dce9908df1d9d48ec0e26e2a250839fa36ea2c602cc4f85ccfeb5c65/WTForms-2.2.1-py2.py3-n
one-any.whl (166kB)
    100% |████████████████████████████████| 174kB 101kB/s
Requirement already satisfied: Flask in /c/Users/vivek/OneDrive - University College Dublin/ucd/2019/teaching/bdic/web-app-dev/lecture-slides/sa
mple-code/week9/microblog/flaskenv/lib/python3.7/site-packages (from flask-wtf)
Requirement already satisfied: click>=5.1 in /c/Users/vivek/OneDrive - University College Dublin/ucd/2019/teaching/bdic/web-app-dev/lecture-slid
es/sample-code/week9/microblog/flaskenv/lib/python3.7/site-packages (from Flask->flask-wtf)
Requirement already satisfied: itsdangerous>=0.24 in /c/Users/vivek/OneDrive - University College Dublin/ucd/2019/teaching/bdic/web-app-dev/lect
ure-slides/sample-code/week9/microblog/flaskenv/lib/python3.7/site-packages (from Flask->flask-wtf)
Requirement already satisfied: Jinja2>=2.10.1 in /c/Users/vivek/OneDrive - University College Dublin/ucd/2019/teaching/bdic/web-app-dev/lecture-
slides/sample-code/week9/microblog/flaskenv/lib/python3.7/site-packages (from Flask->flask-wtf)
Requirement already satisfied: Werkzeug>=0.15 in /c/Users/vivek/OneDrive - University College Dublin/ucd/2019/teaching/bdic/web-app-dev/lecture-
slides/sample-code/week9/microblog/flaskenv/lib/python3.7/site-packages (from Flask->flask-wtf)
Requirement already satisfied: MarkupSafe>=0.23 in /c/Users/vivek/OneDrive - University College Dublin/ucd/2019/teaching/bdic/web-app-dev/lectur
e-slides/sample-code/week9/microblog/flaskenv/lib/python3.7/site-packages (from Jinja2>=2.10.1->Flask->flask-wtf)
Installing collected packages: WTForms, flask-wtf
Successfully installed WTForms-2.2.1 flask-wtf-0.14.2
(flaskenv) microblog>
```

NOW WE CONFIGURE IT

- In the top-level directory (`blogapp`), create a new module called `config.py`
- Add the following code



```
import os

class Config(object):
    SECRET_KEY = os.environ.get('SECRET_KEY') or 'you-will-never-guess'
```

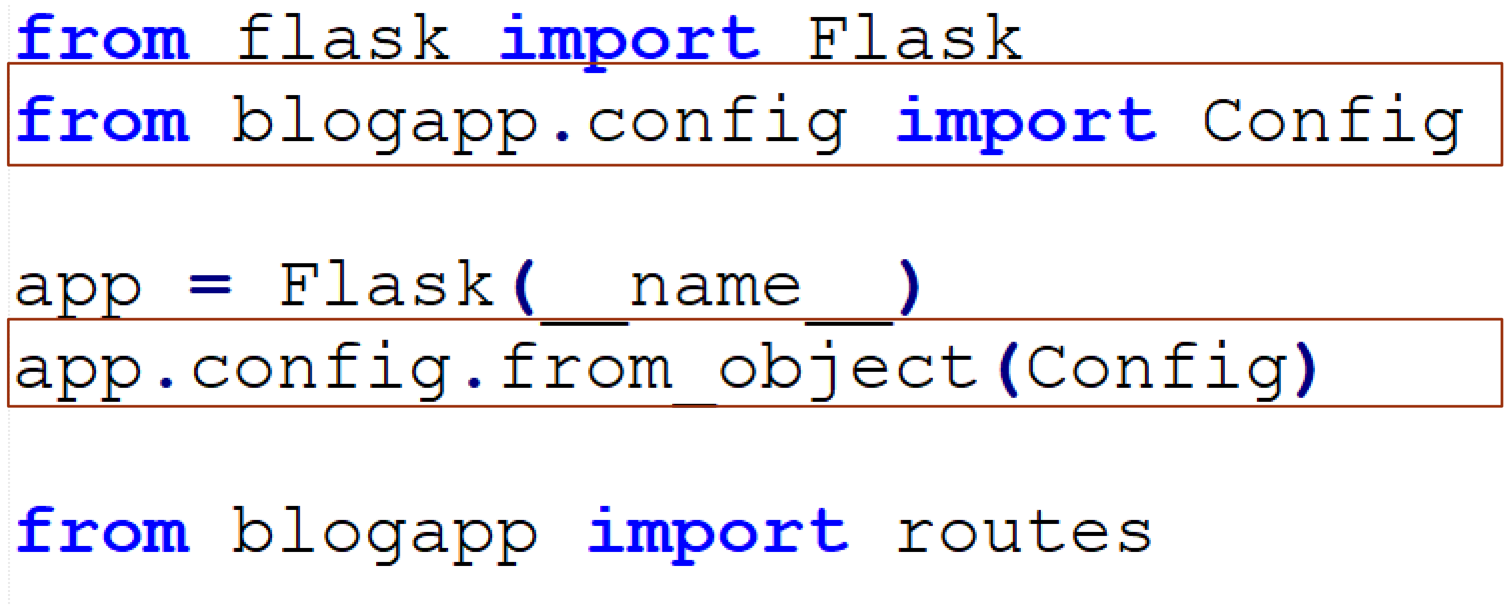
WHAT DOES THAT DO?

- Flask and some extensions use SECRET_KEY as a cryptographic key to generate signatures or tokens
- Flask-WTF extension uses it to protect against `Cross-Site Request Forgery` attacks
- So, we put it in an environment variable so that it is not visible in code. So, the server has a secure key that no one else knows

- The ‘`or`’ operator is used while coding so that it gets a value, even if the environment variable is empty

MAKE YOUR APP READ THE CONFIG

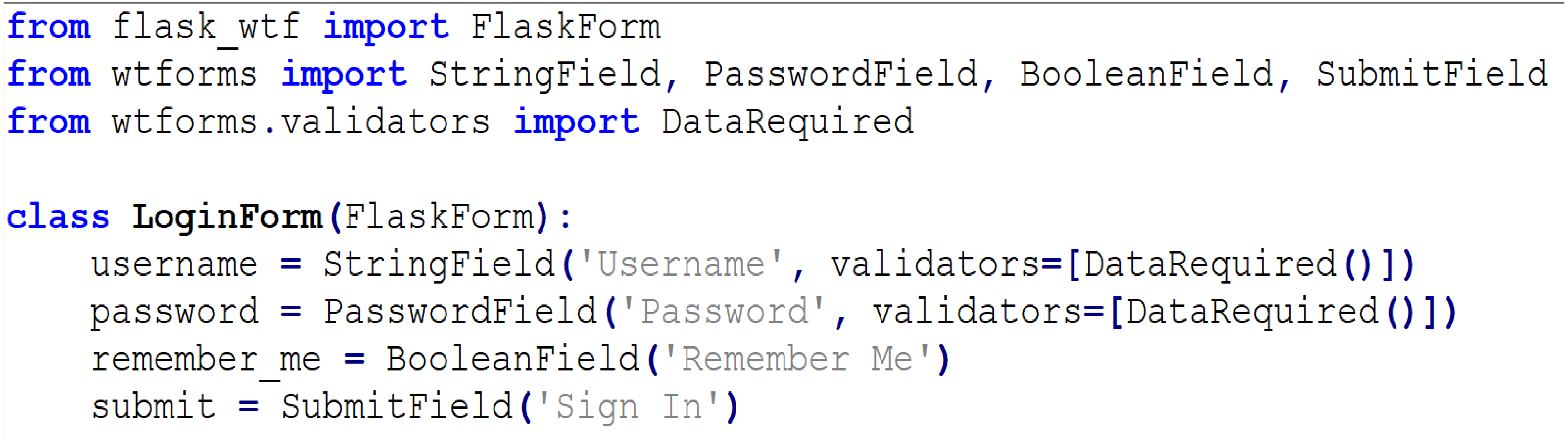
- Open the __init__.py file, and add code to read the configuration file



```python
from flask import Flask
from blogapp.config import Config

app = Flask(__name__)
app.config.from_object(Config)

from blogapp import routes
```

CREATING A FORM

- Flask-WTF uses python classes to represent web forms
- A form class simply defines the fields of the form as class variables
- In the `blogapp` directory, we create a login form by writing a `forms.py` file



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

class LoginForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired()])
    password = PasswordField('Password', validators=[DataRequired()])
    remember_me = BooleanField('Remember Me')
    submit = SubmitField('Sign In')
```

EXPLANATION – FORMS.PY

- Flask-WTF has all its symbols in the `flask_wtf` module
- The FlaskForm class is defined in the `flask_wtf` module
- From `WTForms`, we import field types directly
- For each field, an object is created as a class variable in the `LoginForm` class
- Each field is given a description or label as an argument
- The optional `validators` argument is used to attach validators to each field

ADD A TEMPLATE TO RENDER IT

- All fields in `LoginForm` class know how to render themselves
- We just need to create a template which calls them
- In the `templates` directory, create a file called: `login.html` [code in next slide]
- Then, create a new route called `login` in `routes.py`
- In `login,` we call the template we just created

LOGIN.HTML

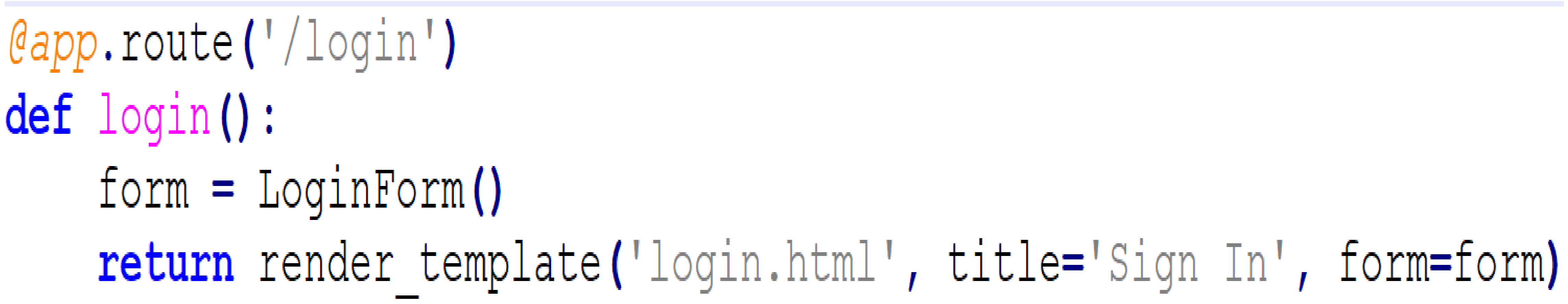
```
{% extends "base.html" %}

{% block content %}
    <h1>Sign In</h1>
    <form action="" method="post" novalidate>
        {{ form.hidden_tag() }}
        <p>
            {{ form.username.label }}<br>
            {{ form.username(size=32) }}
        </p>
        <p>
            {{ form.password.label }}<br>
            {{ form.password(size=32) }}
        </p>
        <p>{{ form.remember_me() }} {{ form.remember_me.label }}</p>
        <p>{{ form.submit() }}</p>
    </form>
{% endblock %}
```

Used to prevent CSRF attacks

- In `routes.py`, first import the `LoginForm` class

```
from blogapp import app
from blogapp.forms import LoginForm
```

- Add a decorator, and define a function called `login`



```
@app.route('/login')
def login():
    form = LoginForm()
    return render_template('login.html', title='Sign In', form=form)
```

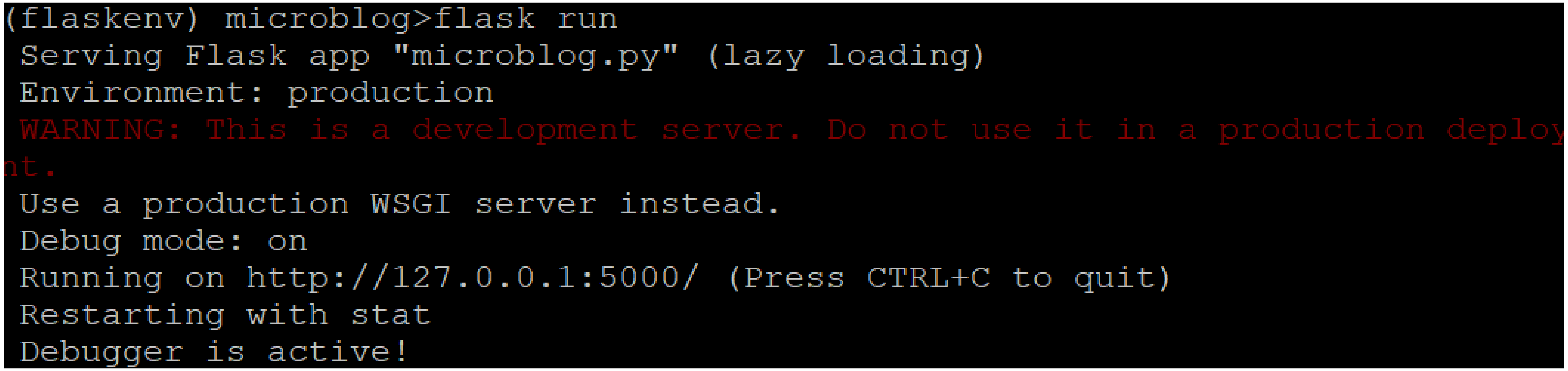
FINALLY, CHANGE THE BASE.HTML

- To make it easy to access the form from anywhere, add it to the base template

```
<html>
    <head>
      {% if title %}
      <title>{{ title }} - Microblog</title>
      {% else %}
      <title>Don't be lazy! Set a title.</title>
      {% endif %}
    </head>
    <body>
        <div>Microblog:
            <a href="/index">Home</a>
            <a href="/login">Login</a>
        </div>
        <hr>
        {% block content %}{% endblock %}
    </body>
</html>
```

ACCESS THE WEBSITE

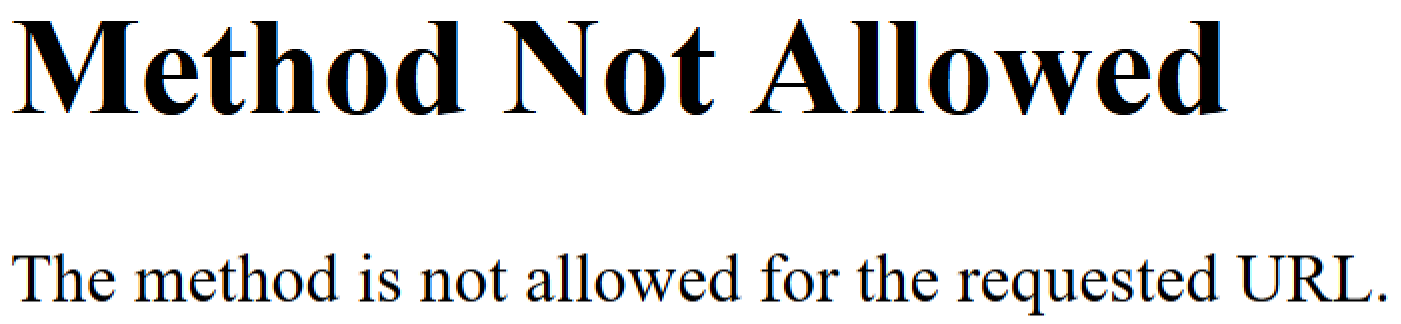
- On the command line, use: `flask run` to run the server



```
(flaskenv) microblog>flask run
 Serving Flask app "microblog.py" (lazy loading)
 Environment: production
 WARNING: This is a development server. Do not use it in a production deploy
nt.
 Use a production WSGI server instead.
 Debug mode: on
 Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 Restarting with stat
 Debugger is active!
```

SENDING DATA

- If you press the submit button, you will get a "Method Not Allowed" error



```
Method Not Allowed

The method is not allowed for the requested URL.
```

WHY?

- Our code has no logic to process the data yet
- Modify the login function to accept and validate user data

```python
@app.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        flash('Login requested for user {}, remember_me={}'.format(
            form.username.data, form.remember_me.data))
        return redirect('/login')
    return render_template('login.html', title='Sign In', form=form)
```

EXPLANATION

- The methods argument in the route decorator tells Flask that it accepts both `GET` and `POST` requests
- Default is only `GET`.
- So, when the form sent a `POST` request, Flask returned "`Method Not Allowed`"
- `form.validate_on_submit()` does all the processing work.
- When the browser sends the first `GET` request to receive the form, the function returns `False`, so the code skips to the `render_template` on the last line
- When it returns `True`, we call two new functions: `flash` and `redirect`
- To show the `flash` function, we change the `base` template

NEW BASE TEMPLATE

```
<body>
    <div>
        Microblog:
        <a href="/index">Home</a>
        <a href="/login">Login</a>
    </div>
    <hr>
    {% with messages = get_flashed_messages() %}
    {% if messages %}
    <ul>
        {% for message in messages %}
        <li>{{ message }}</li>
        {% endfor %}
    </ul>
    {% endif %}
    {% endwith %}
    {% block content %}{% endblock %}
</body>
```

Move newbase.html to base.html

FLASHED MESSAGES

- `get_flashed_messages()` is a method that comes from Flask
- It returns a list of all messages that have been registered using `flash()` previously
- In our template, we use a conditional to check if there are messages and show each message using an `<li>`

ADDING VALIDATION

- The validators at the back end work correctly
- However, the user is not given any indication of what is wrong. The user simply gets the form back
- Good design states that the user should be informed using meaningful error messages next to each field that failed validation
- Thankfully, Flask already generates error messages for the validators. We just need to show them in our template
- So, we modify the `login.html` template to show validation messages for the `username` and `password` fields.
- Typically, fields with validators will have error messages under `form.<field_name>.errors`

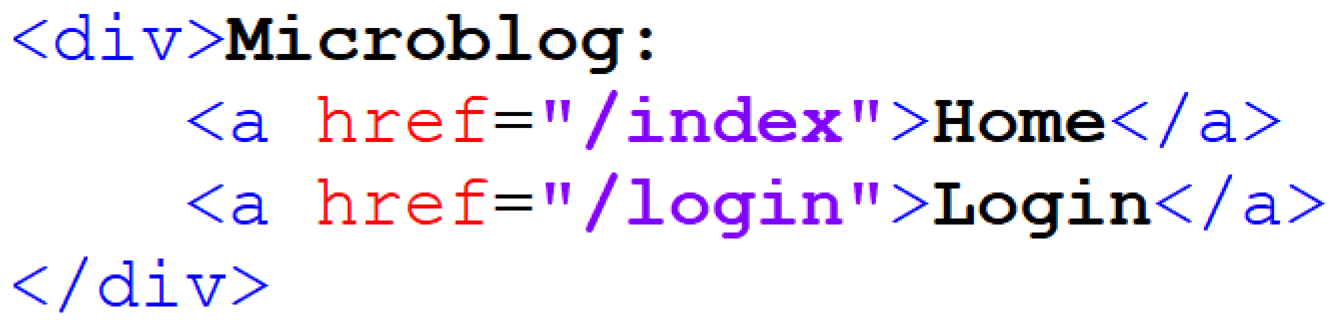
ADD LINES TO LOGIN.HTML

```
<p>
    {{ form.username.label }}<br>
    {{ form.username(size=32) }}

    {% for error in form.username.errors %}
    <span style="color: red;">[{{ error }}]</span>
    {% endfor %}

</p>
<p>
    {{ form.password.label }}<br>
    {{ form.password(size=32) }}

    {% for error in form.password.errors %}
    <span style="color: red;">[{{ error }}]</span>
    {% endfor %}
```

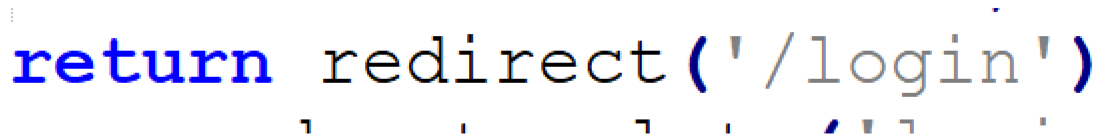
Move newlogin.html to login.html

GENERATING LINKS

- So far, we have been adding links to different parts of the application directly.



```
<div>Microblog:
    <a href="/index">Home</a>
    <a href="/login">Login</a>
</div>
```



```
return redirect('/login')
```

- If we re-organize our application, these links will break

THE `url_for()` FUNCTION

- The `url_for()` function generates urls, using its internal mappings of views to functions
- For example:
  - `url_for ('login')` **returns** `'/login'`
  - `url_for('index')` **returns** `'/index'`

- This is generally more robust because internal function names change less frequently than urls

MODIFY base.html

```
<div>
    Microblog:
    <a href="{{ url_for('index') }}">Home</a>
    <a href="{{ url_for('login') }}">Login</a>
</div>
```

Move newbase2.html to base.html

Modify routes.py

```
from flask import render_template, flash, redirect, url_for
```

```
return redirect(url_for('index'))
```

Move newroutes.py to routes.py

TO-DO

- Follow the steps in the lecture, to get your microblog application working