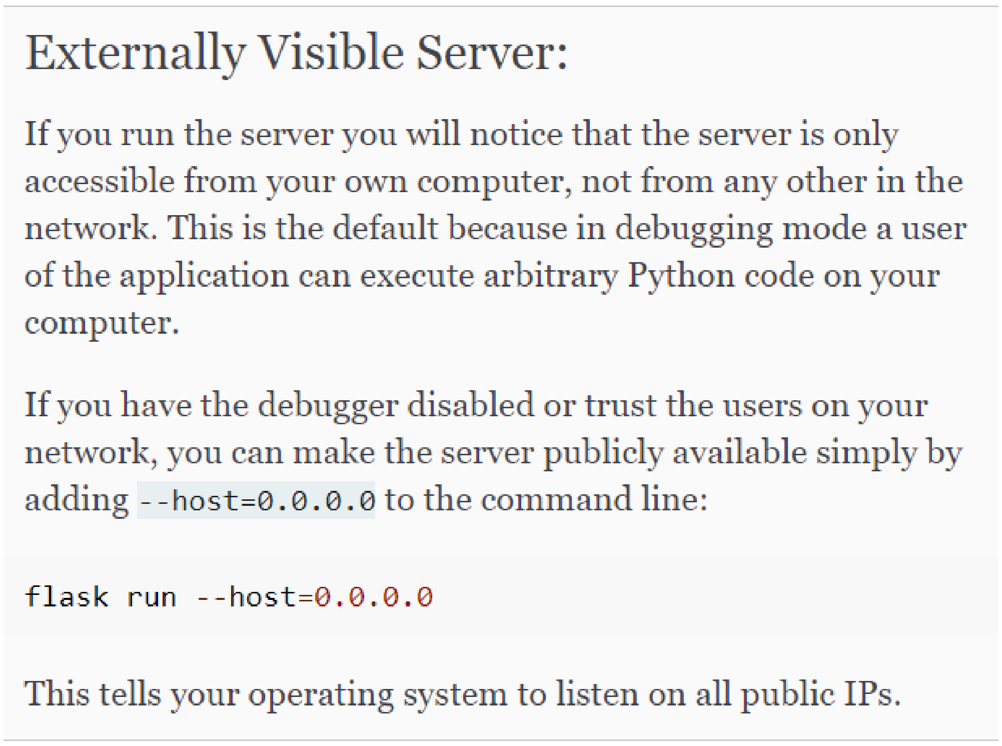
Flask

- Idea about microframeworks
- By convention, `templates` and `static` files are stored in subdirectories within the application's Python source tree, with the names `templates` and `static` respectively. While this can be changed, you usually don't have to, especially when getting started.



Externally Visible Server:

If you run the server you will notice that the server is only accessible from your own computer, not from any other in the network. This is the default because in debugging mode a user of the application can execute arbitrary Python code on your computer.

If you have the debugger disabled or trust the users on your network, you can make the server publicly available simply by adding `--host=0.0.0.0` to the command line:

```
flask run --host=0.0.0.0
```

This tells your operating system to listen on all public IPs.

Routing

- Use the `route()` decorator to bind a function to a URL.

```
@app.route('/')
def index():
    return 'Index Page'

@app.route('/hello')
def hello():
    return 'Hello, World'
```

Variable Rules

- You can add variable sections to a URL by marking sections with `<variable_name>`. Your function then receives the `<variable_name>` as a keyword argument. Optionally, you can use a converter to specify the type of the argument like `converter:variable_name`.

```
@app.route('/user/<username>')
def show_user_profile(username):
    # show the user profile for that user
    return 'User %s' % username

@app.route('/post/<int:post_id>')
def show_post(post_id):
    # show the post with the given id, the id is an integer
    return 'Post %d' % post_id

@app.route('/path/<path:subpath>')
def show_subpath(subpath):
    # show the subpath after /path/
    return 'Subpath %s' % subpath
```

URL Building

- To build a URL to a specific function, use the `url_for()` function. It accepts the name of the function as its first argument and any number of keyword arguments, each corresponding to a variable part of the URL rule. Unknown variable parts are appended to the URL as query parameters.

```
from flask import Flask, url_for

app = Flask(__name__)

@app.route('/')
def index():
    return 'index'

@app.route('/login')
def login():
    return 'login'

@app.route('/user/<username>')
def profile(username):
    return '{}\'s profile'.format(username)

with app.test_request_context():
    print(url_for('index'))
    print(url_for('login'))
    print(url_for('login', next='/'))
    print(url_for('profile', username='John Doe'))

>>> /
>>> /login
>>> /login?next=/
>>> /user/John%20Doe
```

HTTP Methods

- Web applications use different HTTP methods when accessing URLs. You should familiarize yourself with the HTTP methods as you work with Flask. By default, a route only answers to `GET` requests. You can use the `methods` argument of the `route()` decorator to handle different HTTP methods.

```
from flask import request

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        return do_the_login()
    else:
        return show_the_login_form()
```

Static Files (for CSS and JS)

- Dynamic web applications also need static files. That's usually where the CSS and JavaScript files are coming from. Ideally your web server is configured to serve them for you, but during development Flask can do that as well. Just create a folder called `static` in your package or next to your module and it will be available at `/static` on the application.
- To generate URLs for static files, use the special `'static'` endpoint name:

```
url_for('static', filename='style.css')
```

The file has to be stored on the filesystem as `static/style.css`.

Rendering Templates (HTML template file)

- To render a template you can use the `render_template()` method. All you have to do is provide the name of the template and the variables you want to pass to the template engine as keyword arguments.

```
from flask import render_template

@app.route('/hello/')
@app.route('/hello/<name>')
def hello(name=None):
    return render_template('hello.html', name=name)
```

Flask will look for templates in the `templates` folder.

Testing request context

```
from flask import request

with app.test_request_context('/hello', method='POST'):
    # now you can do something with the request until the
    # end of the with block, such as basic assertions:
    assert request.path == '/hello'
    assert request.method == 'POST'

# Passing the whoel environment
from flask import request

with app.request_context(environ):
    assert request.method == 'POST'
```

HTML with JavaScript

- To make the site dynamic
- Change element attributes of the webpage

```
"document.getElementById("xxx").xxx = xxx"
```

- Change innerHTML content <p>**xxxxx**</p>

```
<!DOCTYPE html>
<html>
<body>

<h2>My First JavaScript</h2>

<button type="button"
onclick="document.getElementById('demo').innerHTML = Date()">
```

```
Click me to display Date and Time.</button>

<p id="demo"></p>

</body>
</html>
```

- Change HTML attribute values

```
<!DOCTYPE html>
<html>
<body>

<h2>What Can JavaScript Do?</h2>

<p>JavaScript can change HTML attribute values.</p>

<p>In this case JavaScript changes the value of the src (source) attribute of an image.</p>

<button onclick="document.getElementById('myImage').src='pic_bulbon.gif'">Turn on the light</button>

<img id="myImage" src="pic_bulboff.gif" style="width:100px">

<button onclick="document.getElementById('myImage').src='pic_bulboff.gif'">Turn off the light</button>

</body>
</html>
```

- Change HTML Styles (CSS)

```
<!DOCTYPE html>
<html>
<body>

<h2>What Can JavaScript Do?</h2>

<p id="demo">JavaScript can change the style of an HTML element.</p>

<button type="button" onclick="document.getElementById('demo').style.fontSize='35px'">Click Me!</button>

</body>
</html>
```

- Hide HTML elements

```
<!DOCTYPE html>
<html>
<body>

<h2>What Can JavaScript Do?</h2>

<p id="demo">JavaScript can hide HTML elements.</p>

<button type="button" onclick="document.getElementById('demo').style.display='none'">Click Me!</button>

</body>
</html>
```

- Put functions in head and call later using `<script>`

in head

```
<!DOCTYPE html>
<html>
<head>
<script>
function myFunction() {
  document.getElementById("demo").innerHTML = "Paragraph changed.";
}
</script>
</head>
<body>

<h1>A Web Page</h1>
<p id="demo">A Paragraph</p>
<button type="button" onclick="myFunction()">Try it</button>

</body>
</html>
```

in body

> Placing scripts at the bottom of the <body> element improves the display speed, because script interpretation slows down the display.

```
<!DOCTYPE html>
<html>
<body>

<h1>A Web Page</h1>
<p id="demo">A Paragraph</p>
<button type="button" onclick="myFunction()">Try it</button>

<script>
function myFunction() {
 document.getElementById("demo").innerHTML = "Paragraph changed.";
}
</script>

</body>
</html>
```

Put as external JS file

To use an external script, put the name of the script file in the `src` (source) attribute of a `<script>` tag:

```
<script src="myScript.js"></script>
```

Even external references

```
<script src="https://www.w3schools.com/js/myScript1.js"></script>
```

use `document.write()` for testing purposes

Using document.write() after an HTML document is loaded, will delete all existing HTML:

```
<!DOCTYPE html>
<html>
<body>

<h1>My First Web Page</h1>
<p>My first paragraph.</p>

<script>
document.write(5 + 6);
</script>

</body>
</html>
```

use `window.alert()` | `condole.log()` to display data

```
<!DOCTYPE html>
<html>
<body>

<h1>My First Web Page</h1>
<p>My first paragraph.</p>

<script>
window.alert(5 + 6);
</script>

</body>
</html>
```

Template Inheritance

Base template

```
<!doctype html>
<html>
  <head>
    {% block head %}
    <link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
    <title>{% block title %}{% endblock %} - My Webpage</title>
    {% endblock %}
  </head>
  <body>
    <div id="content">{% block content %}{% endblock %}</div>
    <div id="footer">
      {% block footer %}
      &copy; Copyright 2010 by <a href="http://domain.invalid/">you</a>.
      {% endblock %}
    </div>
  </body>
</html>
```

Child template

```
{% extends "layout.html" %}
{% block title %}Index{% endblock %}
{% block head %}
  {{ super() }}
  <style type="text/css">
```

```
    .important { color: #336699; }
  </style>
{% endblock %}
{% block content %}
  <h1>Index</h1>
  <p class="important">
    Welcome on my awesome homepage.
{% endblock %}
```

Access request, session, g in template

```
{{ session['username'] }}
{{ request['method'] }}
```

Pagination

- In Flask-SQLAlchemy `paginate()` method

```
<query object>.paginate(1,20,False).items

# page number, staring from 1
# number of items per page
# error flag, if true, 404 error sent to client if out of range, if false, empty list returned
# .items is a list of items
```

- URL example for pagination

```
Page 1, implicit: http://localhost:5000/index
Page 1, explicit: http://localhost:5000/index?page=1
Page 3: http://localhost:5000/index?page=3
```

- page navigation methods

```
has_next: True if there is at least one more page after the current one
has_prev: True if there is at least one more page before the current one
next_num: page number for the next page
prev_num: page number for the previous page
```

Methods

flash

- flash() send a message which can be retrieved later
- `flash()` & `get_flashed_messages(with_categories=true)`
  - categories contain 'success' or 'error'
  - in python flask app

  ```
  flash("message"[,"category"])
  ```

- in html

```
{% with messages = get_flashed_messages([with_categories=true])%}
  ...
  {% for category, message in messages %}
    ...
  {% endfor %}
{% endwith %}
```

redirect(url_for(<function name>))

- redirect to the url of a certain function

url_for (static files)

```
url_for('static',filename='style.css')
```

request.args.get()

```
request.args.get(key [,default=None, type=None])

key of the dictionary object
default is the value return if key not found
type specifies type of value, if ValueError, return default value
```

Objects

session

- dict
- keep track on modifications

g

- request bound object for global variables

User Authentication

- Hashing is one-way function, cannot be reversed back