

Web Development with Flask and the Raspberry Pi

Leading by Example



CUAUHTEMOC CARBAJAL

ITESM CEM

22/04/2014



Flask

web development,
one drop at a time



Jinja



Introduction



- Flask: lightweight web application framework written in Python and based on the Werkzeug WSGI toolkit and Jinja2 template engine
 - Web application framework (WAF) :
 - ✦ software framework designed to support the development of dynamic websites, web applications, web services and web resources.
 - ✦ Aims to alleviate the overhead associated with common activities performed in web development.
 - ✦ For example, many frameworks provide libraries for database access, templating frameworks and session management, and they often promote code reuse.

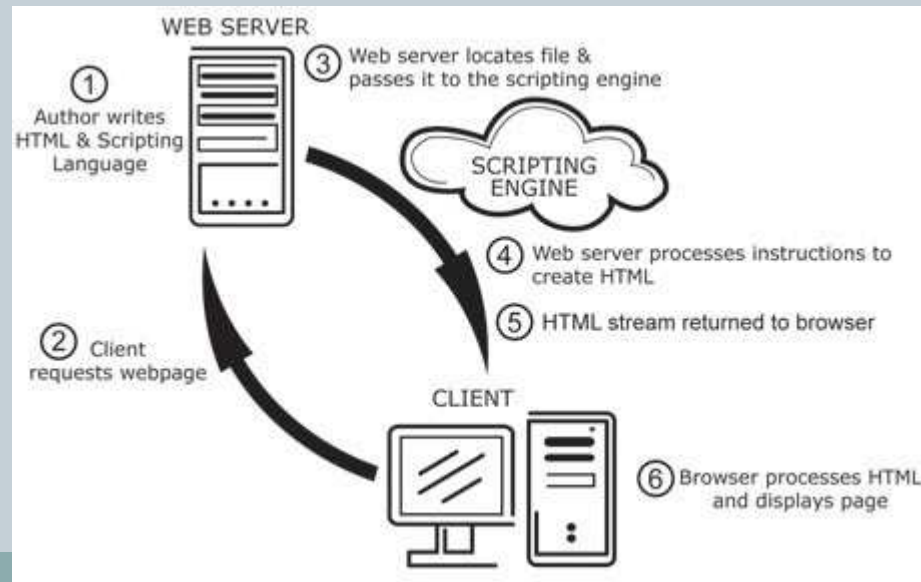
Introduction



- **Web Server Gateway Interface:** specification for simple and universal interface between web servers and web applications or frameworks for the Python programming language.
- **Werkzeug WSGI toolkit:** started as a simple collection of various utilities for WSGI applications and has become one of the most advanced WSGI utility modules.
 - ✦ It includes a powerful debugger, fully featured request and response objects, HTTP utilities to handle entity tags, cache control headers, HTTP dates, cookie handling, file uploads, a powerful URL routing system and a bunch of community contributed addon modules.
- **Jinja:** template engine for the Python programming language

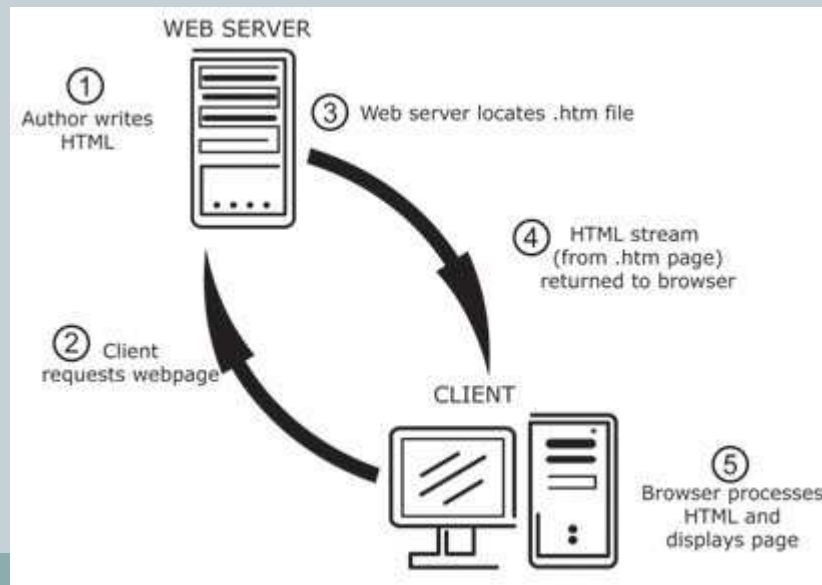
Dynamic web page generation

- Pages are assembled “on the fly” as and when they are requested. Most server side languages as PHP, JSP and ASP powered sites do this technology by actively encourages dynamic content creation. Generating pages dynamically allows for all sorts of clever applications, from e-commerce, random quote generators to full on web applications such as Hotmail.



Static web page generation

- HTML pages are pre-generated by the publishing software and stored as flat files on the web server, ready to be served. This approach is less flexible than dynamic generation in many ways and is often ignored as an option as a result, but in fact the vast majority of content sites consist of primarily static pages and could be powered by static content generation without any loss of functionality to the end user.

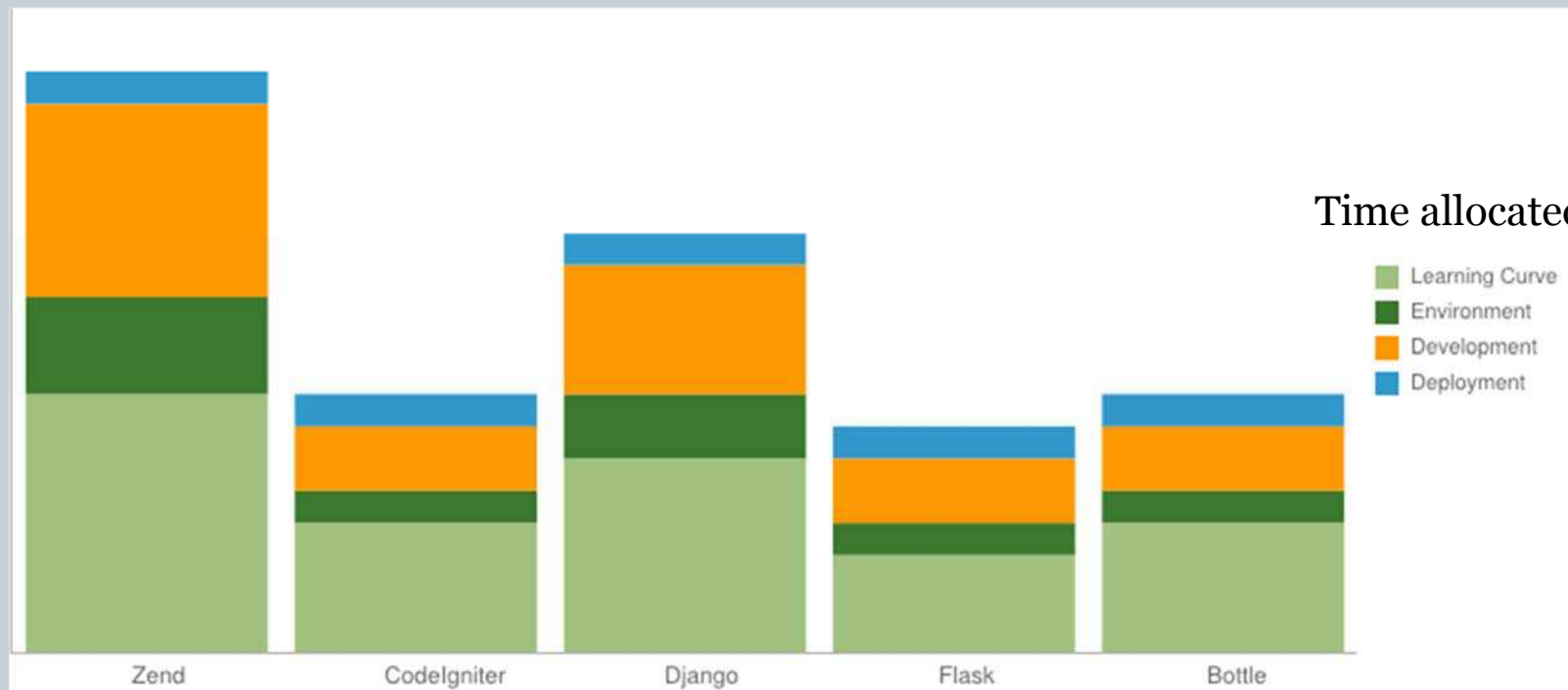


Flask microframework



- **Flask: keeps the core simple but extensible:**
 - There is no database abstraction layer, form validation, or any other components where third-party libraries already exist to provide common functionality.
 - However, Flask supports extensions, which can add such functionality into an application as if it was implemented in Flask itself.
 - There are extensions for object-relational mappers, form validation, upload handling, various open authentication technologies, and more.

Comparison to other frameworks



PHP-based

Python-based

Handling HTTP methods



- The most common keyword argument to `app.route` is `methods`, giving Flask a list of HTTP methods to accept when routing (default: **GET**)
 - **GET**: used to reply with information on resource
 - **POST**: used to receive from browser/client updated information for resource
 - **PUT**: like **POST**, but repeat **PUT** calls on a resource should have no effect
 - **DELETE**: removes the resource
 - **HEAD**: like **GET**, but replies only with HTTP headers and not content
 - **OPTIONS**: used to determine which methods are available for resource

Installation



- **RPi**

- In order to install Flask, you'll need to have pip installed. If you haven't already installed pip, it's easy to do:
- `pi@raspberrypi ~ $ sudo apt-get install python-pip`
- After pip is installed, you can use it to install flask and its dependencies:
- `pi@raspberrypi ~ $ sudo pip install flask`

Flask is fun!



```
# hello-flask.py
from flask import Flask
app = Flask(__name__)
@app.route('/')
def hello_world():
    return "Hello World!"

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

Note: don't call your file **flask.py** if you are interested in avoiding problems.

TERMINAL 1

```
$ python hello-flask.py
* Running on http://127.0.0.1:5000/
* Restarting with reloader
```

TERMINAL 2

```
$ midori &
$ Gtk-Message: Failed to load module "canberra-gtk-module"
$ sudo apt-get install libcanberra-gtk3-module
$ midori &
```

hello-flask



hello-flask



```
app = Flask(__name__)
```

- When we create an instance of the Flask class, the first argument is the name of the application's module or package
- When using a single module, use `__name__` because this will work regardless of whether `__name__` equals `'__main__'` or the actual import name

hello-flask



```
app.run()
```

- The `app.run()` function runs the application on a local server
- This will only be visible on your own computer! We will talk about deployment later

Debugging



- When testing, use `app.run(debug=True)`
 - Now the server will reload itself on code changes
 - Additionally, you will see error messages in the browser
 - But never leave this on in production!

hello-flask



```
@app.route('/')  
def hello_world():  
    return "Hello World!"
```

- The `app.route('/')` decorator tells Flask to call the `hello_world()` function when the relative URL `'/'` is accessed.
- The `hello_world()` function returns the web page (in this case, a simple string) to be displayed.

More routing



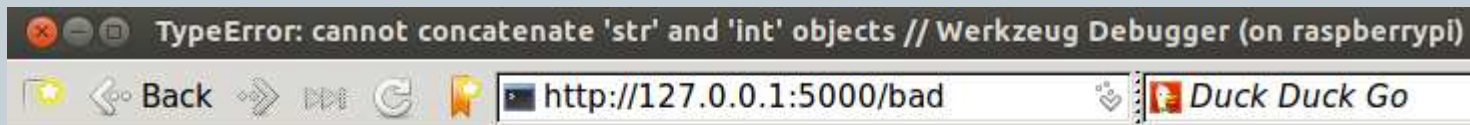
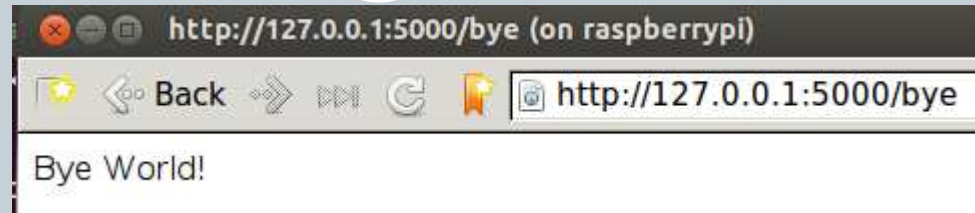
```
#hello-flask2.py
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello World!"

@app.route('/bad')
def bad():
    return 'hi' + 4

@app.route('/bye')
def bye_world():
    return "Bye World!"

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



TypeError

TypeError: cannot concatenate 'str' and 'int' objects

Variable Rules



- To add variable parts to a URL, use `<variable_name>`
- The variables are passed as arguments to the function

```
@app.route('/user/<username>')  
def greet_user(username):  
    return "Hello %s!" % username
```

hello-flask3



```
#hello-flask3.py
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello World!"

@app.route('/user/<username>')
def greet_user(username):
    return "Hello %s!" % username

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



Variable Rules



- Multiple URLs can route to the same function:

```
@app.route('/name/<first>')
@app.route('/name/<first>/<last>')
def greet_name(first, last=None):
    name = first + ' ' + last if last else first
    return "Hello %s!" % name
```

hello-flask4



```
pi@raspberrypi: ~/work/python/flask/hello-flask
GNU nano 2.2.6 File: hello-flask4.py

#hello-flask4.py
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello World!"

@app.route('/name/<first>')
@app.route('/name/<first>/<last>')
def greet_name(first, last=None):
    name = first + ' ' + last if last else first
    return "Hello %s!" % name

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

```
http://127.0.0.1:5000/ (on raspberrypi)
Back http://127.0.0.1:5000/
Hello World!
```

```
http://127.0.0.1:5000/name/Sergio (on raspberrypi)
Back http://127.0.0.1:5000/name/Sergio
Hello Sergio!
```

```
http://127.0.0.1:5000/name/Sergio/Carbajal (on raspberrypi)
Back http://127.0.0.1:5000/name/Sergio/Carbajal
Hello Sergio Carbajal!
```

Why we need templates?



- Let's consider we want the home page of our app to have a heading that welcomes the logged in user.
- An easy option to output a nice and big heading would be to change our view function to output HTML, maybe something like this:

```
from app import app

@app.route('/')
@app.route('/index')
def index():
    user = { 'nickname': 'Miguel' } # fake user
    return ''

<html>
  <head>
    <title>Home Page</title>
  </head>
  <body>
    <h1>Hello, '' + user['nickname'] + ''</h1>
  </body>
</html>
''
```

Why we need templates?



```
pi@raspberrypi: ~/work/python/flask/hello-flask
GNU nano 2.2.6 File: hello-flask0.py

from flask import Flask
app = Flask(__name__)
@app.route("/")
def hello():
    return ''' <h1> Flask on Raspberry Pi </h1> '''

if __name__ == "__main__":
    app.run(host='0.0.0.0',port=5000, debug=True)
```



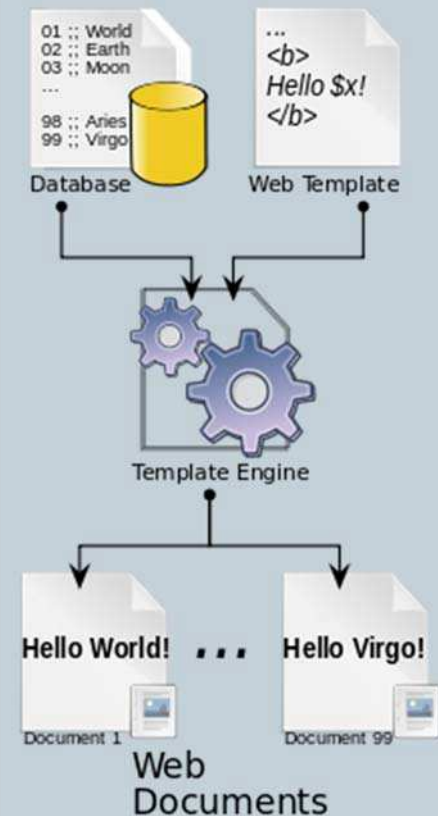
Templating



- Generating HTML from within Python is not fun, and actually pretty cumbersome because you have to do the HTML escaping on your own to keep the application secure.
- Consider how complex the code will become if you have to return a large and complex HTML page with lots of dynamic content.
- And what if you need to change the layout of your web site in a large app that has dozens of views, each returning HTML directly?
 - This is clearly not a scalable option.

Templating

- Templates to the rescue...
 - A web template system uses a template engine to combine web templates to form finished web pages, possibly using some data source to customize the pages or present a large amount of content on similar-looking pages.
 - Flask uses a templating system called Jinja.



Jinja is beautiful!



- Philosophy: keep the logic of your application separate from the layout or presentation of your web pages
- Jinja2 allows you to use most of the Python syntax that you are used to, inside of your templates, helping you generate either text or code in a powerful, yet flexible way.
- We just wrote a mostly standard HTML page, with the only difference that there are some placeholders for the dynamic content enclosed in `{{ ... }}` sections.

```
<html>
  <head>
    <title>{{title}} - microblog</title>
  </head>
  <body>
    <h1>Hello, {{user.nickname}}!</h1>
  </body>
</html>
```

template

```
from flask import render_template
from app import app

@app.route('/')
@app.route('/index')
def index():
    user = { 'nickname': 'Miguel' } # fake user
    return render_template("index.html",
                           title = 'Home',
                           user = user)
```

view function

Rendering Templates



- 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.
- Here's a simple example of how to render a template:

```
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.

Control statements in templates



- The Jinja2 templates also support control statements, given inside `{% ... %}` blocks.

```
<!doctype html>
<title>Hello from Flask</title>
{% if name %}
    <h1>Hello {{ name }}!</h1>
{% else %}
    <h1>Hello World!</h1>
{% endif %}
```

template-name



```
pi@raspberrypi: ~/work/python/flask/template-name
GNU nano 2.2.6 File: template-name.py

from flask import Flask, render_template

app = Flask(__name__)

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

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

```
Hello from Flask (on raspberrypi)
Back http://127.0.0.1:5000/
```

Hello World!

```
Hello from Flask (on raspberrypi)
Back http://127.0.0.1:5000/Sergio
```

Hello Sergio!

```
pi@raspberrypi: ~/work/python/flask/template-name/templates
GNU nano 2.2.6 File: index.html

<!doctype html>
<title>Hello from Flask</title>
{% if name %}
    <h1>Hello {{ name }}!</h1>
{% else %}
    <h1>Hello World!</h1>
{% endif %}
```

hello-template



```
#hello-template.py
from flask import Flask, render_template
import datetime
app = Flask(__name__)
@app.route("/")
def hello():
    now = datetime.datetime.now()
    timeString = now.strftime("%Y-%m-%d %H:%M")
    templateData = {
        'title' : 'HELLO!',
        'time': timeString
    }
    return render_template('main.html', **templateData)

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80, debug=True)
```

hello-template

```
pi@raspberrypi: ~/work/python/flask/hello-template
GNU nano 2.2.6 File: hello-template.py

from flask import Flask, render_template
import datetime
app = Flask(__name__)
@app.route("/")
def hello():
    now = datetime.datetime.now()
    timeString = now.strftime("%Y-%m-%d %H:%M")
    templateData = {
        'title' : 'HELLO!',
        'time': timeString
    }
    return render_template('main.html', **templateData)

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80, debug=True)
```

```
pi@raspberrypi: ~/work/python/flask/hello-template/templates
GNU nano 2.2.6 File: main.html

<!DOCTYPE html>
<head>
<title>{{ title }}</title>
</head>
<body>
<h1>Hello, World!</h1>
<h2>The date and time on the server is: {{ time }}</h2>
</body>
</html>
```

RPi

```
$ sudo python hello-template.py
```

PC

Hello, World!

The date and time on the server is: 2014-04-18 14:31

Adding a favicon



- A “favicon” is an icon used by browsers for tabs and bookmarks. This helps to distinguish your website and to give it a unique brand.
- How to add a favicon to a flask application?
 - First, of course, you need an icon.
 - It should be 16 × 16 pixels and in the ICO file format. This is not a requirement but a de-facto standard supported by all relevant browsers.
 - Put the icon in your static directory as favicon.ico.
 - Now, to get browsers to find your icon, the correct way is to add a link tag in your HTML. So, for example:

```
<link rel="shortcut icon" href="{{ url_for('static',  
filename='favicon.ico') }}">
```

- That’s all you need for most browsers.

Adding a favicon



- main.html modified from hello-template

```
GNU nano 2.2.6      File: main.html

<!DOCTYPE html>
<head>
<title>{{ title }}</title>
</head>
<body>
<link rel="shortcut icon" href="{{ url_for('static', filename='favicon.ico') }}">
<h1>Hello, World!</h1>
<h2>The date and time on the server is: {{ time }}</h2>
</body>
</html>
```

- Python hello-template.py



Template Inheritance



- The most powerful part of Jinja is template inheritance. Template inheritance allows you to build a base “skeleton” template that contains all the common elements of your site and defines blocks that child templates can override.
- Sounds complicated but is very basic. It’s easiest to understand it by starting with an example.

Base Template



- This template, which we'll call `template.html`, defines a simple HTML skeleton document. It's the job of “child” templates to fill the empty blocks with content.

```
pi@raspberrypi: ~/work/python/flask/inheritance/templates
GNU nano 2.2.6 File: template.html

<!DOCTYPE html>
<html>
  <head>
    <title>Flask Tutorial 1</title>
  </head>
  <body>
    <div class="container">
      {% block content %}
      {% endblock %}
    </div>
  </body>
</html>
```

Child templates



```
pi@raspberrypi: ~/work/python/flask/inheritance/templates
GNU nano 2.2.6 File: home.html

{% extends "template.html" %}
{% block content %}
  <div class="jumbo">
    <h2>Welcome to Flask!</h2>
    <p>Click <a href="/welcome">here</a> to go to the welcome page</p>
  </div>
{% endblock %}
```

```
pi@raspberrypi: ~/work/python/flask/inheritance/templates
GNU nano 2.2.6 File: welcome.html

{% extends "template.html" %}
{% block content %}
  <h2>Sample</h2>
  <p>Lorem ipsum ad his scripta blandit partiendo...</p>
{% endblock %}
```

view file



```
pi@raspberrypi: ~/work/python/flask/inheritance
GNU nano 2.2.6 File: inheritance.py

# inheritance.py
from flask import Flask, render_template
app = Flask(__name__)

@app.route('/')
def home():
    return render_template('home.html')

@app.route('/welcome')
def welcome():
    return render_template('welcome.html')

if __name__ == '__main__':
    #app.run(debug=True)
    app.run(host='0.0.0.0', port=5000, debug=True)
```

```
$ python inheritance.py
```

