PYTHON CORE
FLASK

softserve

Flask

Flask is a **microframework** for **Python** based on Werkzeug (he Python WSGI Utility Library), Jinja 2 (modern and designer-friendly templating language for Python) and good intentions.

It's BSD licensed!

Flask
web development, one drop at a time

What’s in the Box?

- built-in development server and debugger
- integrated unit testing support
- RESTful request dispatching
- uses Jinja2 templating
- support for secure cookies (client side sessions)
- 100% WSGI 1.0 compliant
- Unicode based
- extensively documented

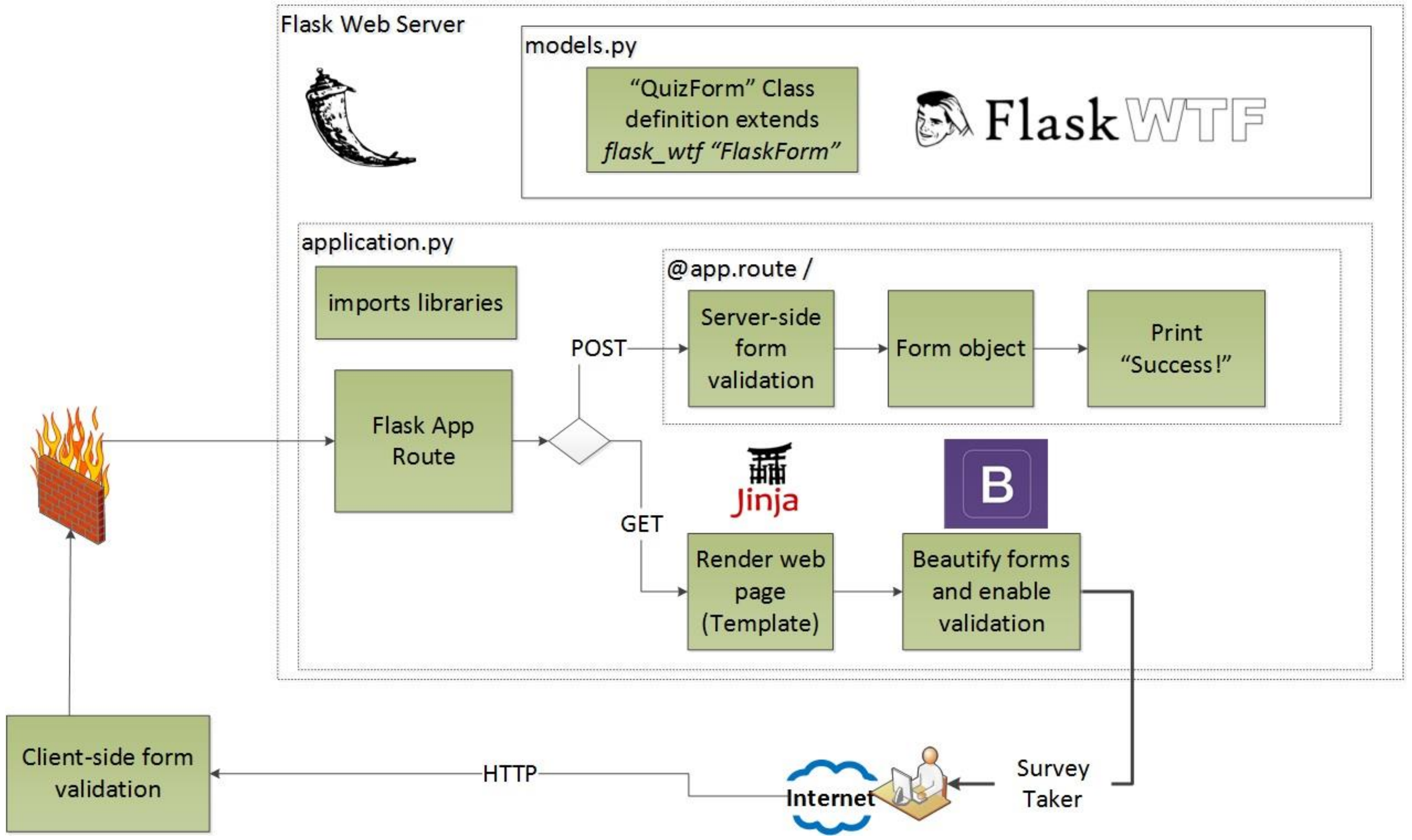
http://flask.pocoo.org/

Flask examples on github:

flaskr — a microblog
minitwit — a twitter clone
this website — static pages + mailinglist archives

softserve

Flask Architecture


Flask Web Server
models.py
"QuizForm" Class definition extends *flask_wtf "FlaskForm"*
Flask WTF
application.py
imports libraries
@app.route /
POST
Server-side form validation
Form object
Print "Success!"
Flask App Route
GET
Jinja
Render web page (Template)
B
Beautify forms and enable validation
Client-side form validation
HTTP
Internet
Survey Taker

softserve

Configuration and Conventions

Flask has many configuration values, with sensible defaults, and a few conventions when getting started.

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.

Installation

1. Install Virtualenv

2. Once you have virtualenv installed, just fire up a shell and create your own environment. Create a project folder and a venv folder within:

```
$ mkdir myproject
$ cd myproject
$ virtualenv venv
```

```
$  pip install virtualenv
$  python3 -m venv venv
```

3. Now, whenever you want to work on a project, you only have to activate the corresponding environment.

```
$ venv\Scripts\activate
```

4. Enter the following command to get Flask activated in your virtualenv:

```
pip install Flask
```

softserve

A Minimal Application

Just save next code as **hello.py** or something similar. !Make sure to not call your application flask.py because this would conflict with Flask itself.

```
from flask import Flask
app = Flask(__name__)

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

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

```
Terminal: $ python hello.py
```

1. We **imported the Flask class**. An instance of this class will be our **WSGI application**.
2. Next we **create an instance of this class**. The first argument is the name of the application's module or package. If you are using a single module (as in this example), you should use __name__ because depending on if it's started as application or imported as module the name will be different ('__main__' versus the actual import name). This is needed so that Flask knows where to look for templates, static files, and so on
3. We then use **the route() decorator** to tell Flask what URL should trigger our function.
4. The function is given a name which is also used to **generate URLs** for that particular function, and **returns the message** we want to display in the user's browser.

softserve

Routing

Modern web applications have beautiful URLs. This helps people remember the URLs.

As you have seen above, the **route()** decorator is used to bind a function to a URL.

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

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

Variable Rules

To add variable parts to a URL you can mark these special sections as **<variable_name>**. Such a part is then passed as a keyword argument to your function. Optionally a converter can be used by specifying a rule with **<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
```

string	accepts any text without a slash
int	accepts integers
float	like int but for floating point values
path	like the default but also accepts slashes
any	matches one of the items provided
uuid	accepts UUID strings

softserve

URL Building

```
>>> from flask import Flask, url_for
>>> app = Flask(__name__)
>>> @app.route('/')
... def index(): pass

>>> @app.route('/login')
... def login(): pass

>>> @app.route('/user/<username>')
... def profile(username): pass

>>> 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
```

To build a URL to a specific function you can use the **url_for()** function.

first argument - the **name of the function**

thecond argument - number of **keyword arguments**

softserve

HTTP Methods

HTTP (the protocol web applications are speaking) knows different methods for accessing URLs. By default, a route only answers to GET requests, but that can be changed by providing the methods argument to the route() decorator.

! You need import **request**

```
from flask import request

@app.route("/")
def index():
    return "Method used: %s" % request.method



# By default only GET allowed, but you can change that using
the methods argument

@app.route("/check_method", methods=['GET', 'POST'])
def check_method():
    if request.method == 'POST':
        return "You are using POST"
    else:
        return "You are probably using GET"
```

softserve

Rendering Templates

To render a template you can use the **render_template()** method.

```
from flask import render_template

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

Template file **hello.html**

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

Case 1: a module:

```
/application.py
/templates
    /hello.html
```

Case 2: a package:

```
/application
    /__init__.py
    /templates
        /hello.html
```

softserve

More questions ?

softserve

TO
THE
FUTURE
softserve