Bottle

Jack Rosenthal

2017-03-23

Mines Linux Users Group

HTTP Crash Course

- GET – Your typical “downloads from a web page” HTTP request
  - Has no support for sending data to the server other than the URL requested
- POST – Your typical “form submission” HTTP request
  - Sends multipart form data along with the request
  - Web browsers ask you to “resubmit” when refreshed
- PUT – Ask to put a file at a location
  - Not needed as the same data can be sent using POST
  - Not supported in older browsers
  - Simple to use when there's no web browser involved

HTTP Crash Course

- GET – Your typical “downloads from a web page” HTTP request
  - Has no support for sending data to the server other than the URL requested
- POST – Your typical “form submission” HTTP request
  - Sends multipart form data along with the request
  - Web browsers ask you to “resubmit” when refreshed
- PUT – Ask to put a file at a location
  - Not needed as the same data can be sent using POST
  - Not supported in older browsers
  - Simple to use when there's no web browser involved

HTTP Crash Course

- GET – Your typical “downloads from a web page” HTTP request
  - Has no support for sending data to the server other than the URL requested
- POST – Your typical “form submission” HTTP request
  - Sends multipart form data along with the request
  - Web browsers ask you to “resubmit” when refreshed
- PUT – Ask to put a file at a location
  - Not needed as the same data can be sent using POST
  - Not supported in older browsers
  - Simple to use when there's no web browser involved

HTTP Crash Course

- GET – Your typical “downloads from a web page” HTTP request
  - Has no support for sending data to the server other than the URL requested
- POST – Your typical “form submission” HTTP request
  - Sends multipart form data along with the request
  - Web browsers ask you to “resubmit” when refreshed
- PUT – Ask to put a file at a location
  - Not needed as the same data can be sent using POST
  - Not supported in older browsers
  - Simple to use when there's no web browser involved

HTTP Crash Course

- GET – Your typical “downloads from a web page” HTTP request
  - Has no support for sending data to the server other than the URL requested
- POST – Your typical “form submission” HTTP request
  - Sends multipart form data along with the request
  - Web browsers ask you to “resubmit” when refreshed
- PUT – Ask to put a file at a location
  - Not needed as the same data can be sent using POST
  - No support in older browsers
  - Simple to use when there's no web browser involved

HTTP Crash Course

- GET – Your typical “downloads from a web page” HTTP request
  - Has no support for sending data to the server other than the URL requested
- POST – Your typical “form submission” HTTP request
  - Sends multipart form data along with the request
  - Web browsers ask you to “resubmit” when refreshed
- PUT – Ask to put a file at a location
  - Not needed as the same data can be sent using POST
  - No support in older browsers
  - Simple to use when there’s no web browser involved

HTTP Crash Course

- GET – Your typical “downloads from a web page” HTTP request
  - Has no support for sending data to the server other than the URL requested
- POST – Your typical “form submission” HTTP request
  - Sends multipart form data along with the request
  - Web browsers ask you to “resubmit” when refreshed
- PUT – Ask to put a file at a location
  - Not needed as the same data can be sent using POST
  - No support in older browsers
  - Simple to use when there’s no web browser involved

HTTP Crash Course

- GET – Your typical “downloads from a web page” HTTP request
  - Has no support for sending data to the server other than the URL requested
- POST – Your typical “form submission” HTTP request
  - Sends multipart form data along with the request
  - Web browsers ask you to “resubmit” when refreshed
- PUT – Ask to put a file at a location
  - Not needed as the same data can be sent using POST
  - No support in older browsers
  - Simple to use when there’s no web browser involved

HTTP Crash Course

- GET – Your typical “downloads from a web page” HTTP request
  - Has no support for sending data to the server other than the URL requested
- POST – Your typical “form submission” HTTP request
  - Sends multipart form data along with the request
  - Web browsers ask you to “resubmit” when refreshed
- PUT – Ask to put a file at a location
  - Not needed as the same data can be sent using POST
  - No support in older browsers
  - Simple to use when there’s no web browser involved

Decorators

Decorators are a pretty way to wrap functions using functions that return functions.

Both the following are equivalent:

```
@logging
def foo(bar, baz):
    return bar + baz - 42


# equivalent to...
def foo(bar, baz):
    return bar + baz - 42
foo = logging(foo)
```

Bottle makes heavy use of decorators to bind into routing.

Decorators

Decorators are a pretty way to wrap functions using functions that return functions.

Both the following are equivalent:

```
@logging
def foo(bar, baz):
    return bar + baz - 42

# equivalent to...
def foo(bar, baz):
    return bar + baz - 42
foo = logging(foo)
```

Bottle makes heavy use of decorators to bind into routing.

Bottle: Really Simple Web Framework

- Provides routing and convenient access to data
- Built in HTTP server, or use any WSGI-compatible web server
- Very lightweight, only a couple thousand lines of code

A *Hello, World!* App

```
from bottle import route, run, template

@route('/hello/<name>')
def hello(name):
    return '<h1>Hello {name}!</h1>'.format(name)

run(host='localhost', port=8080)
```

Bottle: Really Simple Web Framework

- Provides routing and convenient access to data
- Built in HTTP server, or use any WSGI-compatible web server
- Very lightweight, only a couple thousand lines of code

A *Hello, World!* App

```
from bottle import route, run, template

@route('/hello/<name>')
def hello(name):
    return '<h1>Hello {name}!</h1>'.format(name)

run(host='localhost', port=8080)
```

Bottle: Really Simple Web Framework

- Provides routing and convenient access to data
- Built in HTTP server, or use any WSGI-compatible web server
- Very lightweight, only a couple thousand lines of code

A *Hello, World!* App

```
from bottle import route, run, template

@route('/hello/<name>')
def hello(name):
    return '<h1>Hello {name}!</h1>'.format(name)

run(host='localhost', port=8080)
```

Bottle: Really Simple Web Framework

- Provides routing and convenient access to data
- Built in HTTP server, or use any WSGI-compatible web server
- Very lightweight, only a couple thousand lines of code

A *Hello, World!* App

```
from bottle import route, run, template

@route('/hello/<name>')
def hello(name):
    return '<h1>Hello {name}!</h1>'.format(name)

run(host='localhost', port=8080)
```

Where should I use Bottle?

What Bottle Is

- Bottle is a *micro-framework*
- Bottle is only a library
- Bottle really small and fast

What Bottle Is Not

- Bottle is not a MVC framework
- Bottle will not generate files for you
- *No magic included!* :)

Where should I use Bottle?

What Bottle Is

- Bottle is a *micro-framework*
- Bottle is only a library
- Bottle really small and fast

What Bottle Is Not

- Bottle is not a MVC framework
- Bottle will not generate files for you
- *No magic included!*

Where should I use Bottle?

What Bottle Is

- Bottle is a *micro-framework*
- Bottle is only a library
- Bottle really small and fast

What Bottle Is Not

- Bottle is not a MVC framework
- Bottle will not generate files for you
- *No magic included!*

Where should I use Bottle?

What Bottle Is

- Bottle is a *micro-framework*
- Bottle is only a library
- Bottle really small and fast

What Bottle Is Not

- Bottle is not a MVC framework
- Bottle will not generate files for you
- *No magic included!*

Where should I use Bottle?

What Bottle Is

- Bottle is a *micro-framework*
- Bottle is only a library
- Bottle really small and fast

What Bottle Is Not

- Bottle is not a MVC framework
- Bottle will not generate files for you
- *No magic included!*

Where should I use Bottle?

What Bottle Is

- Bottle is a *micro-framework*
- Bottle is only a library
- Bottle really small and fast

What Bottle Is Not

- Bottle is not a MVC framework
- Bottle will not generate files for you
- *No magic included!*

Routing

Routing is how you let Bottle know which URLs map to which functions. Bottle uses decorators to specify this.

```
@route('/hello/<name>')
def hello(name):
    return '<h1>Hello {name}!</h1>'.format(name)
```

You can specify multiple routes per function, even using default arguments.

```
@route('/')
@route('/page/<pagename>')
def wiki_page(pagename='FrontPage'):
    # ... load page from database and return it
```

Routing

Routing is how you let Bottle know which URLs map to which functions. Bottle uses decorators to specify this.

```
@route('/hello/<name>')
def hello(name):
    return '<h1>Hello {name}!</h1>'.format(name)
```

You can specify multiple routes per function, even using default arguments.

```
@route('/')
@route('/page/<pagename>')
def wiki_page(pagename='FrontPage'):
    # ... load page from database and return it
```

More Routing

URLs match more specific routes before more generalized.

```
@route('/page/<pagename>')
def wiki_page(pagename='FrontPage'):
    # ... load up a wiki page

@route('/<username>/<pagename>')
def user_page(username, pagename):
    # ... load up a user's personal page
```

So `/page/WikiPage` will use `wiki_page`, but `/jrosenth/home` will use `user_page`.

Routing Modifiers

You can modify parameters to only match certain types.

- `:int` – Only match integers
- `:float` – Only match real numbers
- `:path` – Match path to end of URL (including slashes)
- `:re` – Match a regular expression

```
@route('/user/<uid:int>')
def user_by_id(uid):
    # ... load up a user by id

@route('/user/<username:re:[a-z]+>')
def user_by_name(username):
    # ... load up a user by name
```

Routing Modifiers

You can modify parameters to only match certain types.

- `:int` – Only match integers
- `:float` – Only match real numbers
- `:path` – Match path to end of URL (including slashes)
- `:re` – Match a regular expression

```
@route('/user/<uid:int>')
def user_by_id(uid):
    # ... load up a user by id


@route('/user/<username:re:[a-z]+>')
def user_by_name(username):
    # ... load up a user by name
```

Routing Modifiers

You can modify parameters to only match certain types.

- `:int` – Only match integers
- `:float` – Only match real numbers
- `:path` – Match path to end of URL (including slashes)
- `:re` – Match a regular expression

```
@route('/user/<uid:int>')
def user_by_id(uid):
    # ... load up a user by id

@route('/user/<username:re:[a-z]+>')
def user_by_name(username):
    # ... load up a user by name
```

Routing Modifiers

You can modify parameters to only match certain types.

- `:int` – Only match integers
- `:float` – Only match real numbers
- `:path` – Match path to end of URL (including slashes)
- `:re` – Match a regular expression

```
@route('/user/<uid:int>')
def user_by_id(uid):
    # ... load up a user by id

@route('/user/<username:re:[a-z]+>')
def user_by_name(username):
    # ... load up a user by name
```

Routing Modifiers

You can modify parameters to only match certain types.

- `:int` – Only match integers
- `:float` – Only match real numbers
- `:path` – Match path to end of URL (including slashes)
- `:re` – Match a regular expression

```
@route('/user/<uid:int>')
def user_by_id(uid):
    # ... load up a user by id

@route('/user/<username:re:[a-z]+>')
def user_by_name(username):
    # ... load up a user by name
```

Routing Methods

The `route` decorator accepts all HTTP methods by default, you can use any of the alternate decorators to accept specific methods.

```
@get('/login')
def login_page():
    # ... display the login form

@post('/login')
def login_user():
    # ... process the submitted form
```

Error Routes

The `error` decorator matches certain HTTP errors.

```
@error(403)
def error403(error):
    return "You're not allowed in here."

@error(404)
def error404(error):
    return "It's not my fault."

@error(500)
def error500(error):
    return "I made a mistake."
```

Abort! Abort!

Need to cause an error? **abort** is your amigo.

```
@route('/pages/<pagename>')
def load_page(pagename):
    if pagename not in pages:
        abort(404, "This page does not exist")
    # ...
```

Redirect

Perhaps you wanted to redirect them instead?

```
@route('/pages/<pagename>')
def load_page(pagename):
    if pagename not in pages:
        redirect('/newpage')
    # ...
```

Serving up a static directory

```
from bottle import route, static_file

@route('/static/<filename:path>')
def serve_static(filename):
    return static_file(filename, root='./static')
```

Accessing POST data

The `request` object gives you access to information about the request, including a dictionary contating form data called `forms`.

```
@post('/login')
def login_user():
    username = request.forms.get('username')
    password = request.forms.get('password')
    if login_ok(username, password):
        return 'Congrats! You broke in!'
    else:
        return 'Better luck next time.'
```

Mmm! Cookies!

```
@route('/introduce/<name>')
def introduce(name):
    response.set_cookie('name', name)
    return 'Nice to meet you!'

@route('/welcomeback')
def welcomeback():
    name = request.cookies.get('name', 'Stranger')
    return 'Welcome back, {}!'.format(name)
```

So what can you `return`?

- Returning a string will simply show the string for you
- You can return a dictionary and it will dump JSON for you
- Returning file objects (or anything with `.read()`) will show the file contents for you
- You can even return an iterable or `yield` in your function and it will continually pull and show

So what can you `return`?

- Returning a string will simply show the string for you
- You can return a dictionary and it will dump JSON for you
- Returning file objects (or anything with `.read()`) will show the file contents for you
- You can even return an iterable or `yield` in your function and it will continually pull and show

So what can you `return`?

- Returning a string will simply show the string for you
- You can return a dictionary and it will dump JSON for you
- Returning file objects (or anything with `.read()`) will show the file contents for you
- You can even return an iterable or `yield` in your function and it will continually pull and show

So what can you `return`?

- Returning a string will simply show the string for you
- You can return a dictionary and it will dump JSON for you
- Returning file objects (or anything with `.read()`) will show the file contents for you
- You can even return an iterable or `yield` in your function and it will continually pull and show

Templates

Bottle includes a templating engine called `SimpleTemplate`. You can use it like this:

```
@route('/hello')
@route('/hello/<name>')
@view('hello_template')
def hello(name='World'):
    return dict(name=name)
```

Templates

```
@route('/hello')
@route('/hello/<name>')
@view('hello_template')
def hello(name='World'):
    return dict(name=name)
```

views/hello_template.tpl

```
%if name == 'World':
    <h1>Hello {{name}}!</h1>
    <p>This is a test.</p>
%else:
    <h1>Hello {{name.title()}}!</h1>
    <p>How are you?</p>
%end
```

Applications and Sub-applications

```
@route('/')
def home():
    # ...

# make another bottle app
blog = Bottle()

@blog.route('/')
def blog_home():
    # ...

# /blog accesses all in the blog app
# you can use any WSGI-compatible app here
default_app().mount('/blog', blog)
```

Deploying your application

Bottle has many ways to deploy, here's the two most common:

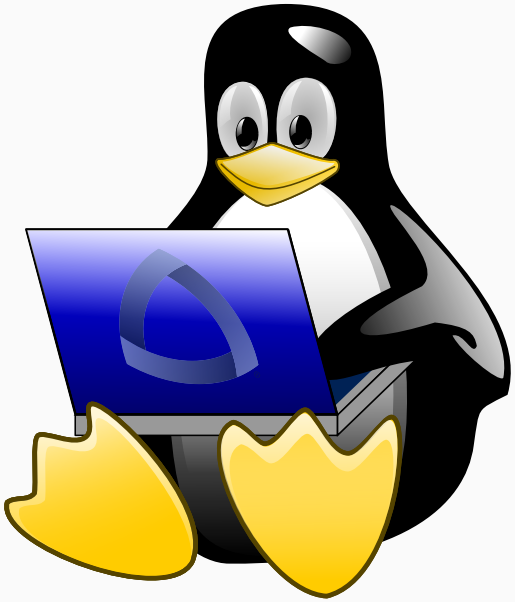
1. Using WSGI, simply create an `app.wsgi` file that imports your bottle app as `application`
2. Using CGI (compatible with nearly any web server), just put `run(server='cgi')` in your CGI script

Questions?

Copyright Notice

This presentation was from the **Mines Linux Users Group**. A mostly-complete archive of our presentations can be found online at `https://lug.mines.edu`.

Individual authors may have certain copyright or licensing restrictions on their presentations. Please be certain to contact the original author to obtain permission to reuse or distribute these slides.



Colorado School of Mines
Linux Users Group