Growler - Using asyncio to build a web framework

PyOhio - Aug 2, 2015

Andrew Kubera

https://github.com/pyGrowler/Growler

import asyncio

- Library introduced in Python 3.4
  - ONLY Python **3** in this talk
    - (or as I like to call it, 'python')
- Asynchronous code run in a single threaded event loop
- Introduces many new terms…

Terms

- Coroutine
- EventLoop
- Protocol
- Transport
- Server
- Task
- Future
- StreamReader/StreamWriter
- yield... from?

I will explain my interpretations of what these do and how to use them.

This is what I wish people would have told me

What is a **Coroutine**?

- Functions that get added to, scheduled, and eventually executed by the event loop
- Coroutines run ONE AT A TIME
  - Different concurrency paradigm than threads
  - can pause, another will run

What is a **Coroutine**?

- A coroutine is created by passing a function to the `asyncio.coroutine()` "constructor"

  - Usually decorated over a function…

```
@asyncio.coroutine
def spam():
    ...

coro = spam()
```

"couroutine" ambiguity!

- `spam` is now a function that returns a coroutine

  - result needs to be given to an event loop

What is a **Coroutine**?

- Coroutines (and only coroutines) have the ability to `"yield from"` other coroutines

  - This both schedules the new coroutine and blocks the current one until result is ready

    - Asynchronous code without callbacks!

    - Can catch exceptions thrown within the coroutine

- Otherwise: call coroutines manually via:

```
loop.run_until_complete(coro)
```

What is an **EventLoop**?

- Object in charge of scheduling and executing the coroutines

- Two execute options:

  - Execute a coroutine: (may call others in meantime)

```
loop.run_until_complete(coro)
```

  - Run ALL coroutines:

```
loop.run_forever()
```

At least one of these MUST be called somewhere in your code to get things started

- You can have more than one event loop at a time

- EventLoops cannot be shared among threads - must create a new one using:

```
loop = asyncio.new_event_loop()
asyncio.set_event_loop(loop)
```

What is a **Future**?

- Object that is placeholder for some *future* value
  - Other languages/libraries call this a *promise*
- Can be *yielded from* in a coroutine
- Should be used for resources that have not been initialized

```
future = asyncio.Future(loop=loop)
future.set_result(value)
```

What is a **Task**?

- From docs:

  "Schedule the execution of a coroutine: wrap it in a future."

- Important part:

  Schedule the execution of a coroutine - TASKS GET DONE!

  - Don't construct, instead build from:

    1. `task = asyncio.async(..., loop=loop)`

    2. `task = loop.create_task(...)`

  - Still must give control over to event loop:

    `loop.run_until_complete(task)` or `loop.run_forever()`

What is **run_in_executor**?

- Method which runs your function in an Executor object
  - It will wait for the function to return and return control to the calling task

- This makes non-asyncio code asynchronous!
  - If you are writing a library - this is how you wrap things with asyncio

```
loop.run_in_executor(None, foo, *args)
```

use a thread pool

function + arguments

coroutine!

What is a **Server**?

- Asyncio handles socket creation for you!

`1.loop.create_server(…)`

- Transports/Protocols

`2.asyncio.start_server(…)`

- StreamReader/StreamWriter

(Both are coroutines!)

What is a **Transport**?

- Is simply the object created when a client creates an incoming connection.

  - This basically wraps the socket.
  - Can read/write data (bytes)

- Wont deal with transports unless you're writing a TCP alternative.

What is a **Protocol**?

- This is where your server code starts.

Subclass This

- The server calls the provided factory function upon incoming connections. (no arguments)

- The server calls `` `connection_made(transport)` `` with the associated transport object.

- Incoming client data will call the protocol's `data_received(data)` member.

What is a **StreamReader**? **StreamWriter**?

- Abstracts away protocol and transport

```
@asyncio.coroutine
def foo(reader, writer):
    data = yield from reader.read(1024)
    writer.write(data)

asyncio.start_server(foo, ...)
```

- asyncio "default" protocol creates these
  - literally: `StreamReaderProtocol`
  - Transport available: `reader._transport`

What about **Clients**?

- Clients have similar functionality

`1.loop.create_connection(…)`

  - Transports/Protocols

`2.asyncio.open_connection(…)`

  - StreamReader/StreamWriter

(Both are coroutines!)

Writing a Server?
Which to use?

`loop.create_server()`

- Your object gets called upon new incoming data
- Responding to events

`asyncio.start_server()`

- Given an object which you have to `yield from` when you want more data.
- Think of as a read/wite file

My Suggestion

`loop.create_server()`

- Route incoming data to other objects - 'event handler'
- Probably have to manage your own buffers

`asyncio.start_server()`

- "Single Task" servers - e.g. read xml document from client
  - Read stream like a file, then write response
- Don't pass around the stream reader to objects - might get lost


Growler

Growler

- Web server written from ground up to use asyncio
- Written by me (and only me 😢)
- NOT STABLE! (Aug 2, 2015)
    - Yet, no one has ever gotten fired for using Growler
- Inspired by NodeJS's Express framework
    - Chain of middleware that you have total control over

Growler

- Install:
  - `pip install growler`
- Install with extras (more coming!)
  - `pip install growler[jade]`

Express Example

```
app = express();

app.use(logger('dev'));
app.use(compression());
app.use(get_user());
```

"Use" middleware functions

```
app.use(function(req, res, next) {

    if (req.user) {
        req.username = req.user.name;
    } else {
        req.username = "@" + req.ip;
    }

    res.scripts = [
        '/jquery/dist/jquery.js'
    ];

    next();
});
```

Middleware function

'Router functions'

```
app.get("/", function(req, res) { res.send_text("Hello "+req.username); });
app.post("/username", function(req, res) { req.user.name = req.body;
                                           res.send(200); });
```

```
http.listen(app, '0.0.0.0', 8080)
```

Called if HTTP request matches:

```
GET / HTTP/1.1
POST /username HTTP/1.1
```

Growler Example

```
app = growler.App("AppName")

app.use(logger('dev'))
app.use(compression())
app.use(get_user())

def page_defaults(req, res):

    if req.user:
        req.username = req.user.name
    else:
        req.username = "@%s" % req.ip

  res.scripts = [
    '/vendor/jquery/dist/jquery.js',
  ]

app.use(page_defaults)



app.get("/", lambda req, res: res.send_text("Hello %s" % req.username))
app.post("/username", lambda req, res: setattr(req.user, 'name', req.body)
                                       or res.send(200))

app.create_server_and_run_forever('0.0.0.0', 8080)
```

Growler Example

```
app = growler.App("AppName")

app.use(logger('dev'))
app.use(compression())
app.use(get_user())

def page_defaults(req, res):
    if req.user:
        req.username = req.user.name
    else:
        req.username = "@%s" % req.ip

  res.scripts = [
    '/vendor/jquery/dist/jquery.js',
  ]

def get_index(req, res):
    res.send_text("Hello %s" % (req.username))

def set_username(req, res):
    setattr(req.user, 'name', req.body)
    res.send(200)

app.use(page_defaults)
app.get('/', get_index)
app.post('/username', set_username)

app.create_server_and_run_forever('0.0.0.0', 8080)
```

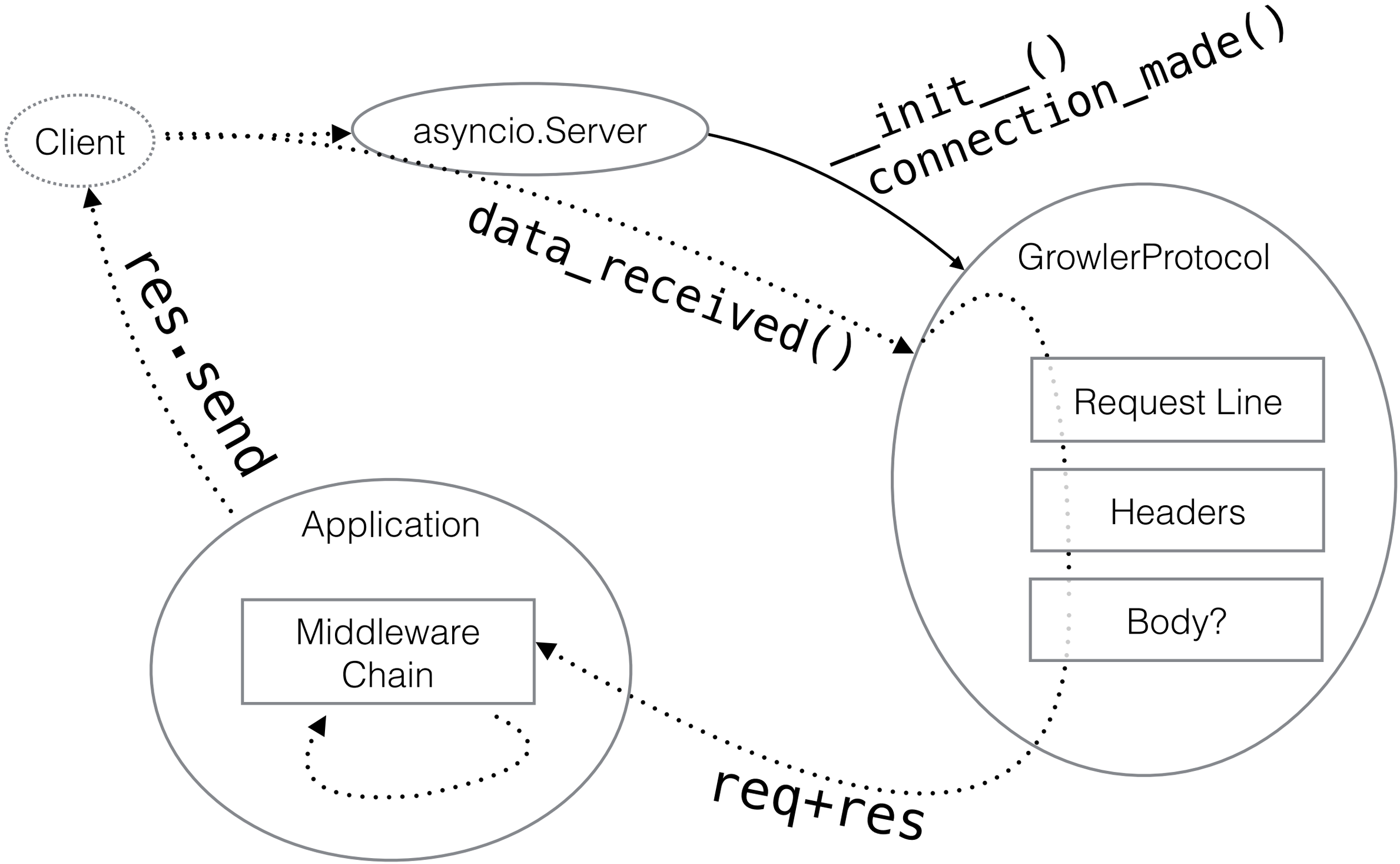
Growler Example

```
app = growler.App("AppName")

app.use(logger('dev'))
app.use(compression())
app.use(get_user())

@app.use
def page_defaults(req, res):
    if req.user:
    req.username = req.user.name
    else:
        req.username = "@%s" % req.ip

  res.scripts = [
    '/vendor/jquery/dist/jquery.js',
  ]

@ap.get('/')
def get_index(req, res):
    res.send_text("Hello %s" % (req.username))

@app.post('/username')
def set_username(req, res):
    setattr(req.user, 'name', req.body)
    res.send(200)


app.create_server_and_run_forever('0.0.0.0', 8080)
```

Which Server Method?

- Originally I used `start_server`
  - StreamReader/StreamWriter seemed perfect backends for req/res objects
- Changed to `create_server`
  - Found it difficult to manage many reader.read() calls
  - Had to implement buffers anyways

Growler Protocol



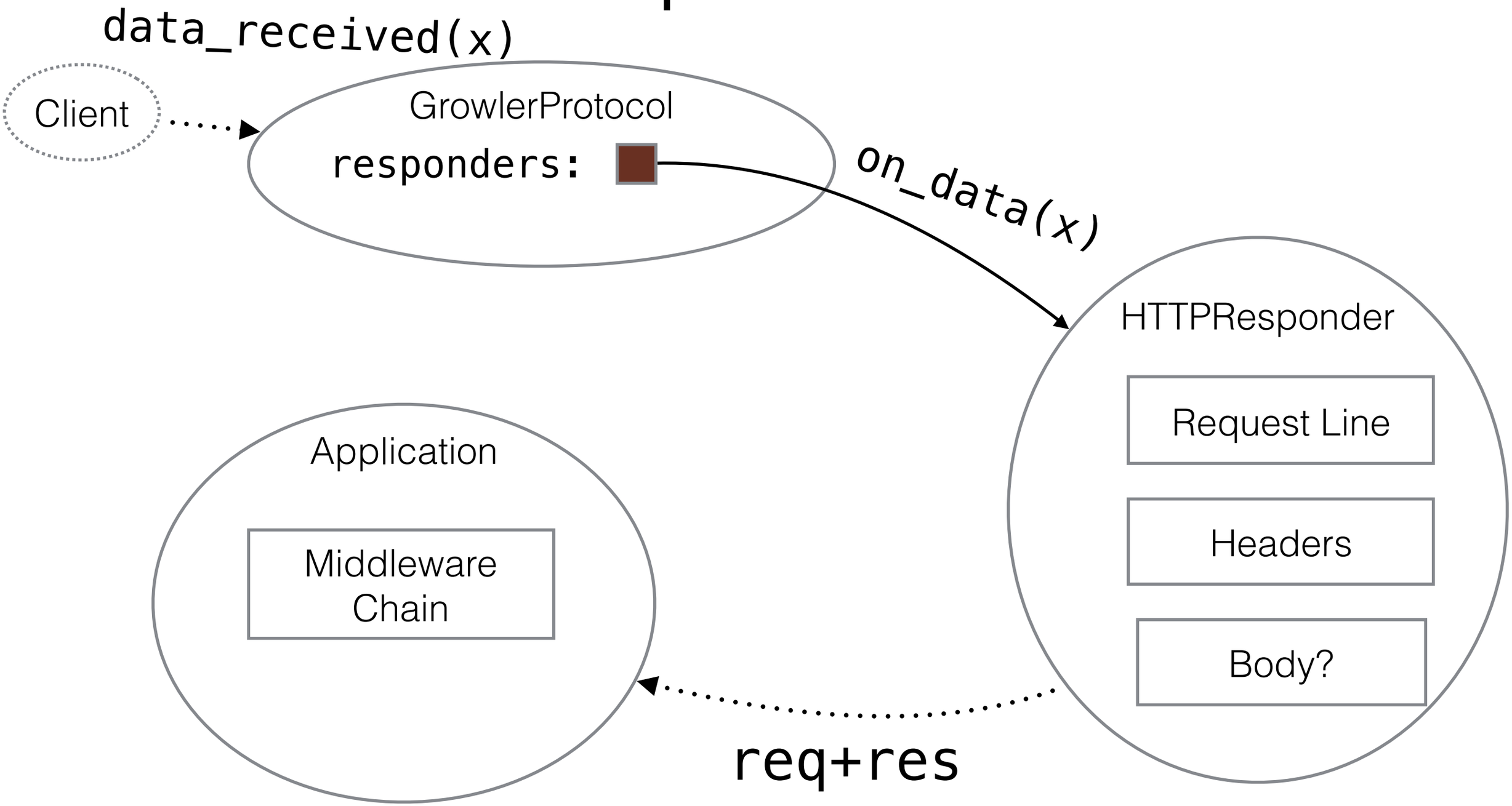
Need Flexible Protocol

- HTTP is an "extendable" protocol
- `Upgrade` header - changes how the request is processed
  - HTTP/2
  - Websocket
- `Keep-Alive`: Might need to reprocess incoming data
- Problem when client "calls" data_received() on protocol

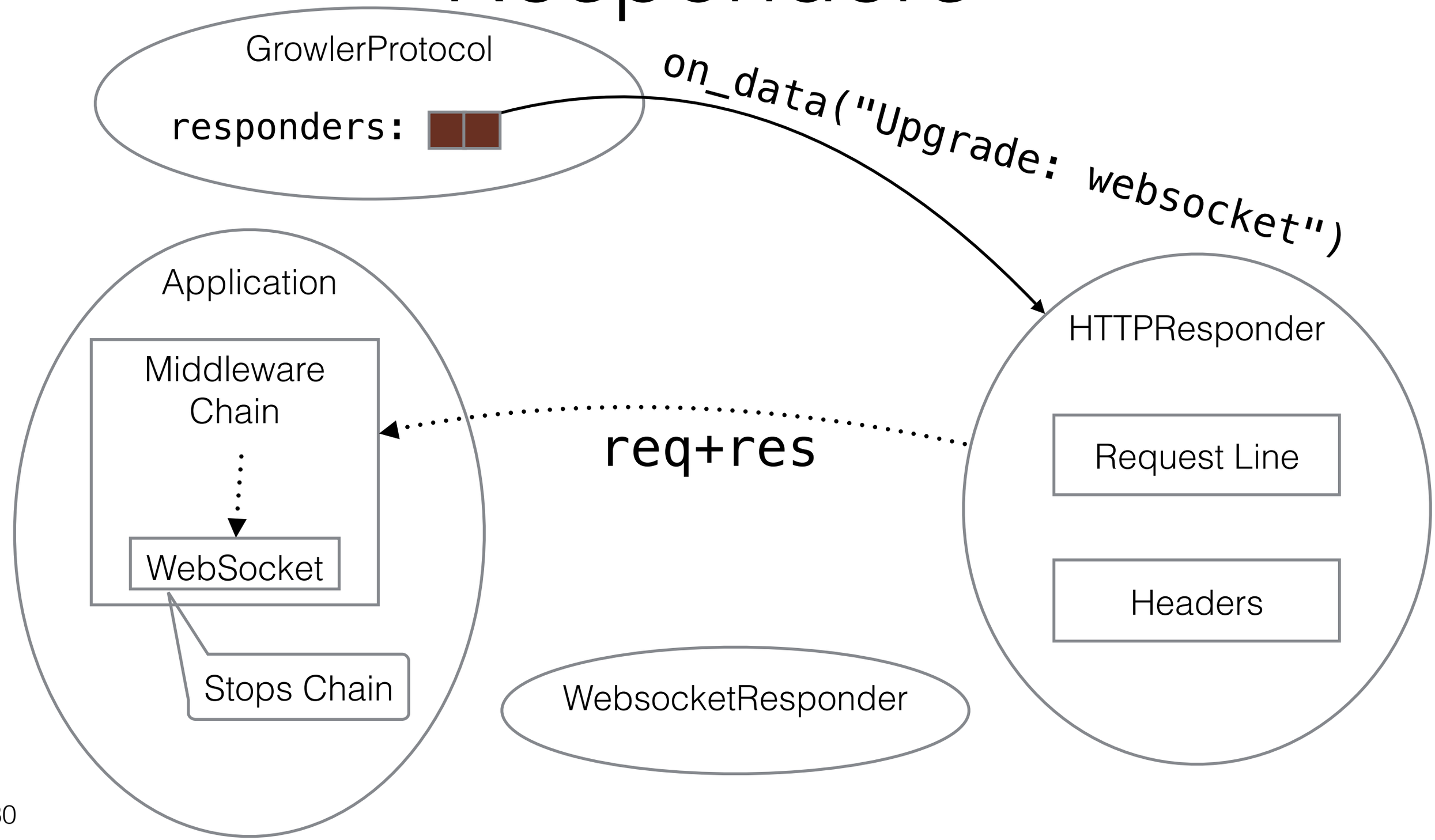
Growler Solution:
"Responder" Stack

- GrowlerProtocol forwards data to top of a stack of objects with an on_data() method.
    - Adds one more level of abstraction: 'Responders'
- Can push and pop responders to change server behavior
- can bypass the middleware chain

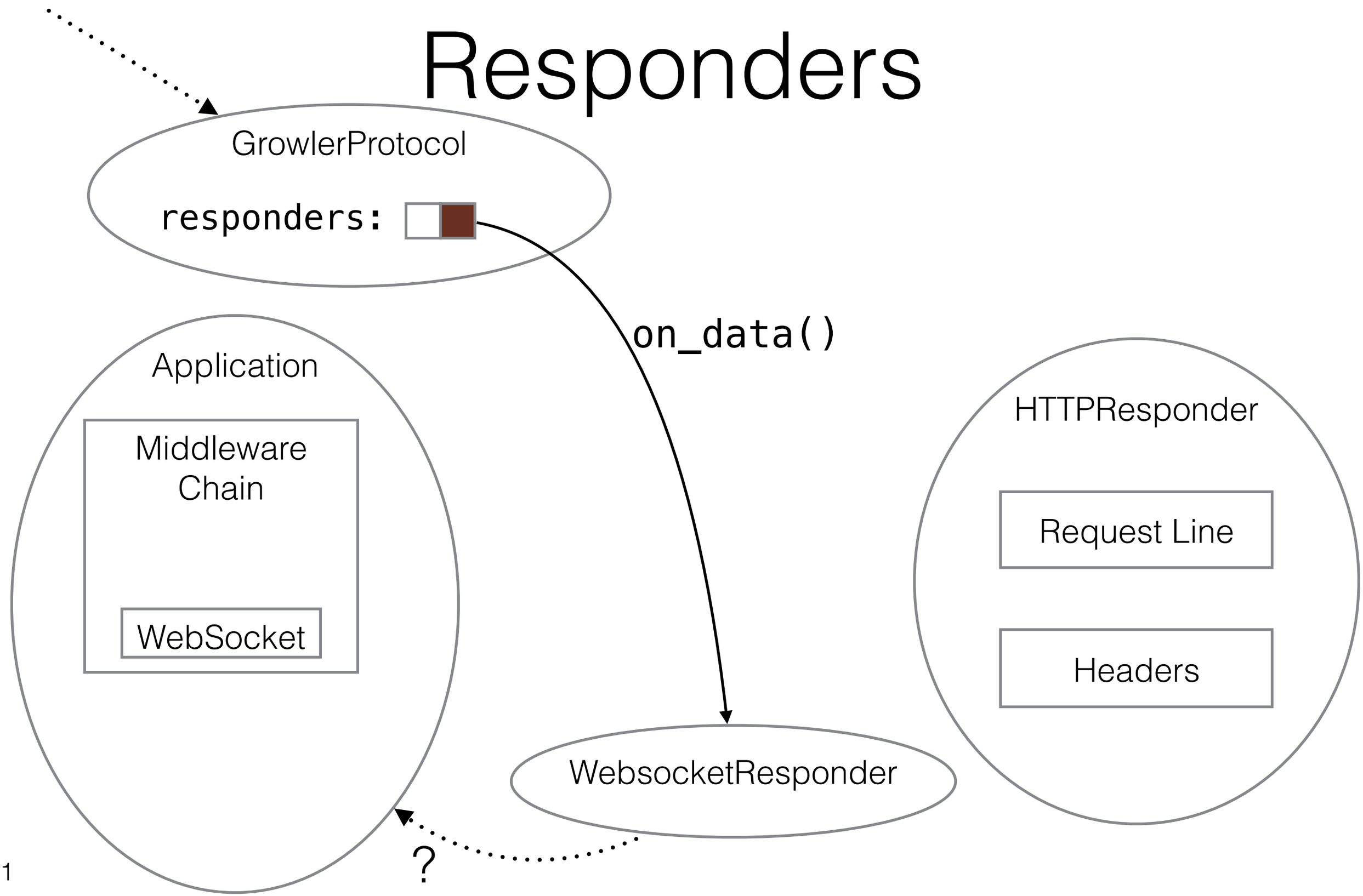
Growler Protocol & Responders



Growler Protocol & Responders



Growler Protocol & Responders



Responder Stack

- Adding middleware which creates responders gives YOU access to sockets
- Highjack HTTP(S) connection for your own purposes

¡ Total Control !

Built-In Middleware

```
from growler.middleware import (
    Renderer,
    CookieParser,
    Session,
    Static,
    Auth
)
```

Adds a 'render' member to res object.

```
res.render("homepage.mako")
```

Adds a 'cookie' member to req object.

`http.cookiejar.`**`Cookie`**

Adds a 'session' member to req object.

`Just uses dictionary as session store.`

Returns 'RAW' files found in directory

`Will call res.send_file(…) if file path matches`

(Currently unimplemented) *Want* it to enable easy user authentication via multiple methods: username/password, Persona, OpenID, OAuth, etc.

```
class Blog:

    ...

    def get_index(self, req, res):
        '''
        Get the root blog page
        '''
        pass

    def get_post_by_id(self, req, res):
         pass

    def get_posts_by_date(self, req, res):
        pass

    def post_new_entry(self, req, res):
        pass


app = App('GrowlerServer')
…

blog = Blog(db=db_cnx)

app.get('/blog', blog.get_index)
app.get('/blog/:id', blog.get_post_by_id)
app.get('/blog/:year/:mo/:day', blog.get_post_by_date)
app.post('/blog/new_post', blog.post_new_entry)
```

```
class Blog:

    ...

    def get_index(self, req, res):
        '''
        Get the root blog page
        '''
        pass

    def get_post_by_id(self, req, res):
         pass

    def get_posts_by_date(self, req, res):
        pass

    def post_new_entry(self, req, res):
        pass


app = App('GrowlerServer')
…

blog = Blog(db=db_cnx)

app.get('/blog', blog.get_index)
app.get('/blog/:id', blog.get_post_by_id)
app.get('/blog/:year/:mo/:day', blog.get_post_by_date)
app.post('/blog/new_post', blog.post_new_entry)
```

```python
from growler.router import routerify

class Blog:

    ...

    def get_index(self, req, res):
        ''' /
        Get the root blog page
        '''
        pass

    def get_post_by_id(self, req, res):
        '''/:id'''
        pass

    def get_posts_by_date(self, req, res):
        '''/:year/:mo/:day'''
        pass

    def post_new_entry(self, req, res):
        '''/blog/new_post'''
        pass


app = App('GrowlerServer')
…

blog = routerify(Blog(db=db_cnx))

app.use('/blog', blog)
```

Routerify works on anything with attributes!
(Objects, Modules)

```python
from growler.router import routerclass

@routerclass
class Blog:

    ...

    def get_index(self, req, res):
        ''' /
        Get the root blog page
        '''
        pass

    def get_post_by_id(self, req, res):
        '''/:id'''
        pass

    def get_posts_by_date(self, req, res):
        '''/:year/:mo/:day'''
        pass

    def post_new_entry(self, req, res):
        '''/blog/new_post'''
        pass


app = App('GrowlerServer')
…

blog = Blog(db=db_cnx)

app.use('/blog', blog)
```

Routerclass automatically calls routerify when object is 'app.used'

Things I've Learned

- Async is NOT Threads!
  - Your code is not actually simultaneous
  - Creating a coroutine is not enough to run it - you must give it to an event loop (via task creation)
  - Your current execution MUST stop and wait for the event loop to do "its thing"

Things I've Learned

- Separate your application from the server
  - Let python handle the socket work, setup your application and let user pass it to their own server.
  - Remember - you can get to the socket

Things I've Learned

- Only make things which need to be asynchronous coroutines
- Don't overcomplicate your code
- Don't add the overhead of scheduling a coroutine to something synchronous

Why Python?

Pandas

mock

nltk

Class-based

Friendly Community

OOP

Decorators

SQLAlchemy

argparse

Requests

SciPy

Why Not!?

py.test

Beautiful Soup

Pillow

toolz

Batteries
Included

Celery

Exception
Handling

numpy

SymPy

pyROOT

Interesting Projects

- Brython
  - Browser-based python environment

```
<script type='text/python'>
…
</script>
```

- PythonJS/Rusthon
  - "compile" python code into equivalent javascript

The Future

- Remote Growler Services
    - Server passes req/res to other processes on computer/network
    - Containment for sub applications

        i.e. "Turn off the blog for maintenance, keeping rest of the site up."

Thank You

Backup Slides

Hello Program

```
import asyncio

from growler import (App, create_http_server)
from growler.middleware import (Logger, Static, Renderer)

loop = asyncio.get_event_loop()

app = App('GrowlerServer')

@app.get('/')
def index(req, res):
    res.render("home")

@app.get('/hello')
def hello_world(req, res):
    res.send_text("Hello World!!")

srv = create_http_server(app(), host='127.0.0.1', port=8000)
asyncio.get_event_loop().run_until_complete(http.listen())
```

Hello Program

```
import growler

app = App('HelloServer')

def say_hello(req, res):
    res.send_text("Hello, Friend")

app.get("/", say_hello)

app.create_server_and_run_forever(host='127.0.0.1',
                                  port=9000)
```

Hello Program

```
import growler

app = App('HelloServer')

@app.get('/')
def say_hello(req, res):
    res.send_text("Hello, Friend")

app.create_server_and_run_forever(host='127.0.0.1',
                                  port=9000)
```

Example Program

```
import asyncio

from growler import (App, create_http_server)
from growler.middleware import (Logger, Static, Renderer)

app = App('GrowlerServer')

app.use(Logger())
app.use(Renderer("views/", "jade"))

@app.get('/')
def index(req, res):
    res.render("home")

@app.get('/hello')
def hello_world(req, res):
    res.send_text("Hello World!!")

app.create_and http = create_http_server(app(), host='127.0.0.1', port=8000)
asyncio.get_event_loop().run_until_complete(http.listen())
```