Make more responsive web applications with SocketIO and gevent

Luke Sneeringer
@lukesneeringer

Part I

State of the Web Address

lukesn.me/py2013-socketio

@lukesneeringer

State of the Web

- Traditional HTTP is:
  - stateless
  - client-driven

State[less] of the Web

- HTTP is designed around a series of mostly-independent requests and responses.
- Not completely independent: cookies, for instance, offer some state storage.
- Contrast with: online multiplayer gaming.


BUNGIE
amatbe rc meaty
You beat down tingamer
You infected tingamer
tingamer has become a zombie
25m
Infection

Client-Driven

- HTTP is, at a fundamental level, client-driven.
- The client asks for something, which starts the process. The server then complies with the request (or doesn't) and returns a response.
- Important takeaway: in traditional HTTP, the client is always the instigator.

AJAX: Client-Driven

- The AJAX model is still built around this fundamental order.
- At its most basic, polling a server with AJAX is a client asking for material on a cadence.
- “Are we there yet? Are we there yet?”

ARE WE THERE YET !?!

State of the Web

- We've been working around these restrictions for a decade or more.
- Flash, AJAX, Comet, long-polling -- these are all ways of disguising or circumventing this fundamental paradigm.
- *cue announcer voice* "And now..."

socket.io

- socket.io is a JavaScript client and server library.
- socket.io is a solution to the “last mile problem”: getting information to browsers without the browser having to repeatedly ask for it.
- socket.io is the newest, browser-compatible way to maintain a stateful connection between browser and server.

But wait, there's Python!

- The socket.io server is also written in JavaScript (Node.js).
- There is also a Python socket.io server built on top of gevent. This is our ultimate focus.

lukesn.me/py2013-socketio @lukesneeringer

Part II

The Client

lukesn.me/py2013-socketio

@lukesneeringer

Getting the Client

Getting the Client

- Sadly, you can't download the client directly from the socket.io website.
- I download it from the examples found in the gevent-socketio repository on GitHub.
  - github.com/abourget/gevent-socketio
- My tutorial repo also has it.

The Client

- The socket.io client is really about doing pretty basic things:
  - Opening and holding a connection at the server.
  - Sending data to the server.
  - Listening for certain data from the server.
- In socket.io, these are spelled `connect`, `emit`, and `on`.

The Client: connect

```
var socket = io.connect('http://serv.er/namespace')
```

- If the browser supports it, socket.io uses websocket to open and hold a connection to the server.
- Not all browsers support websocket, so socket.io will intelligently fall back on the best possible way to get (or mimic) the desired behavior: websocket, Flash, AJAX...

The Client: connect

```
var socket = io.connect('http://serv.er/namespace')
```

- Namespaces are a SocketIO term. It's basically SocketIO's internal URI routing.
- Except not, because it doesn't map to true URIs at all. It's a dirty lie.
- We'll come back to this.

The Client: Events

- The `emit` and `on` methods are about sending and receiving events.
- `emit` sends, `on` listens for receipt.
- Events comprise:
  - the event name (string)
  - zero or more pieces of data (JSON)

The Client: Events

- Receiving events just involves setting up listeners: “I want to know when X happens.”
- The server can (and often does) send more events than the client requests.

The Client: Events

```
var socket = io.connect('http://serv.er/namespace')

socket.on('connect', function() {
    socket.emit('hello', 'Luke Sneeringer')
})
```

- `on` waits for an event to come in. When an event with the proper name does come in, the function runs.
- N.B. The server emits “connect” automatically (with no data).

The Client: Events

```
var socket = io.connect('http://serv.er/namespace')

socket.on('connect', function() {
    socket.emit('hello', 'Luke Sneeringer')
})
```

- `emit` sends data back. This is a “hello” event with a JSON string (my name) as an argument.
- We’ll come back to that server-side.

Part III

The Server

lukesn.me/py2013-socketio

@lukesneeringer

What's Available

- SocketIO was originally designed around using Node.js as the server platform.
- In fact, the SocketIO library assumes that everyone does this.
- Of course, this is PyCon.

What's Available

- Using gevent and the gevent-socketio library with Python offers some advantages over JavaScript:
  - Much cleaner debugging.
  - More straightforward code in general. No need to nest anonymous functions ad infinitum.

The Server

- We're using the gevent-socketio library, available on PyPI.
- For this example, we'll integrate it with Django. Any Python framework will work; it's not Django specific.

The Server: Making it Work

- Basic Steps:
  - Install the SocketIO server. (`pip install gevent-socketio`)
  - Run the SocketIO server.
  - Route requests at one endpoint (usually /socket.io/) to the SocketIO server, and everything else to the traditional web server.

The Server: Making it Work

- Running the SocketIO server is actually pretty straightforward.
- Since environments and frameworks and such differ, you'll probably have to do some scripting:

```python
from gevent import monkey, spawn; monkey.patch_all()
from socketio.server import SocketIOServer
import sys

socket_io_server = SocketIOServer(
    (host, int(port)),
    wsgi_application,
    resource='socket.io',
    policy_server=True,
)

try:
    socket_io_server.serve_forever()
except KeyboardInterrupt:
    print '\r\n\n'
    sys.exit(0)
```

The Server: Making it Work

- `wsgi_application` is the handler. It's the same object that Django and most (all?) other Python web frameworks use.
- `resource` is the actual REST URI.
- `policy_server` is whether to run the Flash policy server.
- Note: The SocketIO server can handle "normal" requests. Useful for dev.

The Server: URI Routing

- Assuming the Django-bundled WSGI application is used, we now go through Django's usual routing process:

```
from __future__ import unicode_literals
from django.conf.urls import patterns, url

urlpatterns = patterns('my_app.views',
    url(r'^/?$', 'home'),
    url(r'^plain/$', 'regular_view'),
    url(r'^socket\.io/', 'socketio_view'),
)
```

The Server: The URI

- A note: The URI here is `socket.io/`.
- This maps to the `resource` argument we looked at earlier.
- It’s easiest to leave this at the default. You can change it, but it has to be done everywhere (e.g. also in `io.connect`), and it’s a pain.

The Server: The View

- A boilerplate Django view moves us off to SocketIO's internal routing.
- SocketIO's rough equivalent to Django views are called namespaces.
- Each namespace is a class, and maps to something that looks like a URI (but isn't; it's a lie).

```python
from socketio import socketio_manage

def socketio_view(request):
    # If this isn't a socketio environment, fail now.
    if 'socketio' not in request.environ:
        return HttpResponseBadRequest('Not SocketIO')

    # Route the request within SocketIO.
    try:
        socketio_manage(request.environ, {
            '/namespace': MyNamespace,
            '/other': MyOtherNamespace,
        }, request)
    except:
        logging.getLogger('socketio').error(
            'Exception while handling socketio connection',
            exc_info=True,
        )

    # We're done; send down an empty response.
    return HttpResponse('')
```

The Server: Namespaces

- The third argument to socketio_manage is `request`. Whatever is sent to that will be available to all namespaces at `self.request`.
- This is not Django-specific. It works with any framework.
- The namespaces' internals don't use `self.request` at all, and it can be omitted if you don't need it.

The Server: Namespaces

- Namespaces are, at their basic, a collection of `on_%s` methods that are called upon receipt of events:

```
socket.emit('hello', 'Luke Sneeringer')
```

```
from socketio.namespace import BaseNamespace

class MyNamespace(BaseNamespace):
    def on_hello(self, name):
        self.name = name
        self.emit('hello_ack', 'Oh, hi!')
```

The Server: Notes

- Data passed back and forth doesn't have to be strings. Anything that JSON understands works.
  - so basically: str, int, float, bool, list, dict
- Also, more than one argument can be used. Arguments retain positional order.
- JavaScript does not support kwargs.

The Last Mile

- Once a SocketIO connection is in place, your namespace can emit an event that the client will pick up immediately.
- The on-methods only give us functionality we already have (but faster).
- More interesting problem: How do we send data without being asked?

The Last Mile

- You're holding a connection now, so there are multiple ways to solve this problem.
- An easy one: Redis.
  - Implements pub/sub.
  - Easy to add to a namespace.

```python
from redis import redis
from socketio.namespace import BaseNamespace
import json

class MyNamespace(BaseNamespace):
    def initialize(self):
        self.redis = Redis(host='localhost', port=6379)
        self.redis_pubsub = redis.pubsub()
        self._greenlet = self.spawn(self._listen)

    def _listen(self):
        for block in self.redis_pubsub.listen():
            if not block or 'data' not in block:
                continue

            data = json.loads(block['data'])
            self.emit(data['event'], data)
```

The Last Mile

- Subscribe to Redis broadcasts in your namespace's on methods
  - redis_pubsub.subscribe([
    "channel",
    ])
- Events will be sent down to the client automatically while the connection holds.

The Rest of It

- Not everything can be covered here.
- Want to know more? Clone my repo:
  - github.com/lukesneeringer/pycon2013_socketio/
- Thanks for attending!

lukesn.me/py2013-socketio @lukesneeringer

Contact

- Luke Sneeringer
  - E-mail: luke@sneeringer.com
  - Twitter: @lukesneeringer
- Example repository:
  - github.com/lukesneeringer/pycon2013_socketio/