Laughing With The Hyenas

Building Your First Website using bottle.py

hound@marginhound.com

PyATL

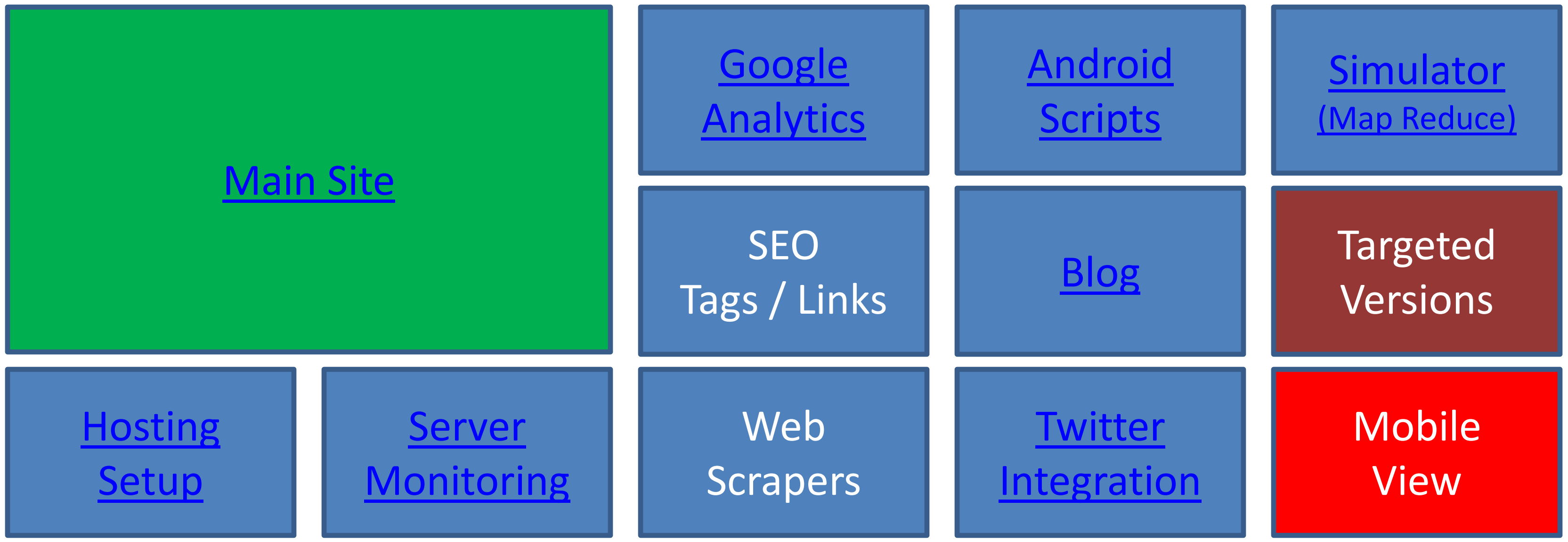
March 8th, 2012

What I'm Here To Talk About

- A small family project
  - Started as a code doodle (anagram solver)
  - Wrapped a web framework around it
  - And shared it with rest of the world
  - Initial focus: hangman solver, hanging with friends helper
- What we learned building it:
  - Bottle.py – micro web-frameworks
  - Site Architecture, Deployment
  - Managing the total package…

Build a Site in A Week!

The Site Took Less Than a Week…



Kicking off about three months of tinkering…

The Core

- Wrote a word game solver using Python:
  - Send it several pieces of data about a puzzle
  - It returns a dictionary object with a list of suggestions
- Encapsulated it within a class method
  - Hides significant internal complexity
  - Can be plugged into other programs (eg. strategy simulator)
  - Modular design, configuration options for new games
- Could have invoked it from the command line
- Instead, the arguments come via HTTP….
- Pattern works for calculations, database lookups, etc.

The Front End

- Not much to see here:
  - Static HTML files
  - Jquery / CSS enhancement
  - Loads page & does AJAX calls to get word ideas
  - Most common calls are cached on server
- Which is the point:
  - Few moving parts – serve content from Apache/nginx
  - Can swap content w/o restarting the server
  - Lots of people & tools available to create HTML.
  - Can swap out server side components fairly easily…

How can we link them?

- Python has some good options:
  - Django: full web framework, many features
  - Others – cherrypy, Web2py, etc.
- But I don’t want to rebuild my application:
  - I just want to wrap my analytics program
  - Handle the details of composing a response
  - And expose it to the web…
- Which is why we have micro-frameworks….

What are Micro Frameworks?

- Minimalist approach to python websites
- Examples – Bottle.py, Flask, many others
- Maps URL routes to Python function calls:
  - Request Routing (URL definition)
  - Request Parsing & Validation
  - Returning Content (files, errors, cookies, etc)
- Can be extended with plug-ins…
  - HTML Templates, Validation, Databases, Sessions

When are Micro Frameworks relevant?

Several areas come to mind:

- Simple or portable applications
  - Data focused web services which don't need a full framework
  - Simplifies process of spinning up a new machine
- Google App Engine (bottle has a special adapter)
- Entry Point for developers from other web languages
- Best of Breed Model (experienced developers)
  - For when the framework doesn't match the way you think
  - Extend framework using plug-ins and custom modules
  - Easier to see what is going on under the hood

When To Think Twice…

Some cautionary notes:

- Don’t reinvent Django
  - If it looks like large CMS / framework, quacks like a…
  - Don’t use bottle if you want a ready-made solution
  - Ideally – seek simplicity or to address a mindset gap
- Usually need a front-end server:
  - Bottle & Flask have development servers
  - Will need to run a “production grade” server in front
  - Both have “adapters” to simplify this process

Introducing Bottle.py

- Been around several years
- Entire framework fits in a single file!
  - No dependencies outside the standard library
  - But…works better with a good server (cherrypy)
- Addresses core web server functions
- Includes "SimpleTemplateEngine" markup language
  - Supports others (Jinja2, Mako)
- Plugins for many common production servers

Routes

- Route Decorator
- URL => Python Function => Returns Result

```
from bottle import route, run

@route('/')
def hello():
    return 'Hello World'

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

- Produces a familiar looking result….

A richer example…

```
from bottle import route, run, validate, static_file

def calc(inputval):
   return {'result':42}
```

Complicated Analytics Function
Must Return a Dictionary
(we're ignoring the inputval
for some cheap humor)

```
@route('/')
def serve_homepage():
    return static_file('home.html', root='static/')
```

Serve the Home Page
(could also use a template)

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

Serves Static Assets (js, css, art)
(in production – move this to
front end server, S3, CDN)

```
@route('/calculate/:inputval', method='GET')
def run_calc(inputval):
   return calc(inputval)

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

Accepts value from browser,
runs "calc" function,
returns dict from calc
as a JSON object

Client side code…

Basic HTML Page

```
<html><head>
<script type="text/javascript" src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js"></script>
<script type="text/javascript">
        $(document).ready(function() { $('#get_answer').click(
                function(){
                        $('#the_answer').empty()
                        $.ajax({ url: '/calculate/50‘,
                        cache:false, type: 'GET',
                        success: function(data) {
                                $('#the_answer‘).append("The Answer IS:" + data.result);}
                        });
                })
        });
</script>
</head><body>

<img src="static/180px-Heinz_Doofenshmirtz.png"><br><br>

Think of a Question and I'll give you the answer <br><br>

<button id='get_answer'>Get the answer!</button> <br><br>

<div id='the_answer'></div>
</body>
</html>
```

Jquery Executes AJAX Call To Server

Grab The Static Image

Trigger The AJAX

Write Out Results

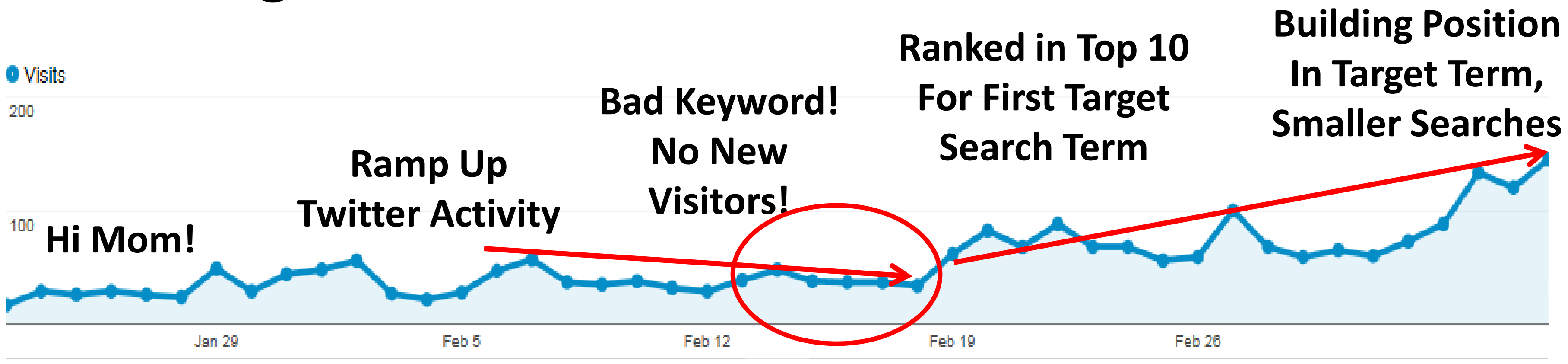
Building Up The Server…

- Dynamic URL’s
  - Regular expressions, @validate decorator, custom validation functions
- Request Object
  - Parse forms, POSTS, handle file uploads
- Other basics
  - Cookies, HTTP error codes, HTTP redirects
- Simple Template Engine
  - Dynamic HTML generation, @view decorator
- Sessions / Caching
  - recommend using beaker
- Databases
  - Plugins for SQLAlchemy, Mongo, sqlite, redis, memcache, others…

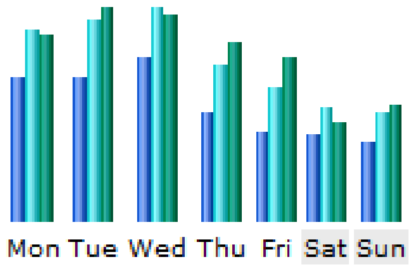
Other Lessons Learned

- Process, Process, Process….
  - Script everything, use automated triggers where possible
  - Pays big dividends – everything is repetitive
- Server Monitoring
  - Minimum – cron job to monitor, report, restart
  - For a more serious site, look at packages/services
- Use Revision Control Religiously
  - Especially for SEO rewrites – helps you see what hit you
- Set up a Staging Environment
  - Internet accessible but in “dark space” (no search engines)
  - Crawl yourself (free tools), load test yourself, live test browsers
- Clean deployment / restart process

Getting Out There



- Product / Audience
  - Your assumptions are wrong. But that's ok…
  - Know where you can actually get users (Scrabble vs. Hangman)
  - Blogging / Twitter helps by forcing you to simplify your message



- SEO – it's worth investing some time to learn this…
  - Knew NOTHING at launch – our design wasn't SEO friendly
  - Ranking on Google takes time – seed critical searches early….
- Learn your traffic patterns, schedule accordingly
  - Twitter & Release new content 12 – 24 hours before peaks (SEO Boost)
  - Release content slowly, so there's always something relatively new…
- Most Important: Have Faith. If you keep trying & learning you'll eventually get it.

Conclusion

- Would I do it again?
  - Absolutely!
- Did it for fun (for now) but..
  - Forced exposure to many areas, measurable competency
  - Got the confidence to pursue more ambitious projects
  - Already using the lessons in my day job
- Idea doesn't have to be great…
  - We figured out the really good stuff (features, promotions, design elements) after we launched!
  - Measurable outcomes (visits, sales, quality) facilitate progress
  - Hardest part is getting started…