Django in the Real World

Jacob Kaplan-Moss

OSCON 2009

http://jacobian.org/TN

Jacob Kaplan-Moss

http://jacobian.org / jacob@jacobian.org / @jacobian

Lead Developer, Django

Partner, Revolution Systems

Shameless plug:

http://revsys.com/

Hat tip:

James Bennett (http://b-list.org)

So you've written a Django site…

… now what?

- API Metering
- Backups & Snapshots
- Counters
- Cloud/Cluster Management Tools
  - Instrumentation/Monitoring
  - Failover
  - Node addition/removal and hashing
  - Auto-scaling for cloud resources
- CSRF/XSS Protection
- Data Retention/Archival
- Deployment Tools
  - Multiple Devs, Staging, Prod
  - Data model upgrades
  - Rolling deployments
  - Multiple versions (selective beta)
  - Bucket Testing
  - Rollbacks
  - CDN Management
- Distributed File Storage
- Distributed Log storage, analysis
- Graphing
- HTTP Caching
- Input/Output Filtering
- Memory Caching
- Non-relational Key Stores
- Rate Limiting
- Relational Storage
- Queues
- Rate Limiting
- Real-time messaging (XMPP)
- Search
  - Ranging
  - Geo
- Sharding
- Smart Caching
  - Dirty-table management

http://randomfoo.net/2009/01/28/infrastructure-for-modern-web-sites

The bare minimum:

- Test.
- Structure for deployment.
- Use deployment tools.
- Design a production environment.
- Monitor.
- Tune.

Testing

Tests are the Programmer's stone, transmuting fear into boredom.

— Kent Beck

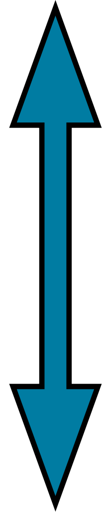
Hardcore TDD

I don’t do test driven development. I do stupidity driven testing… I wait until I do something stupid, and then write tests to avoid doing it again.

— Titus Brown

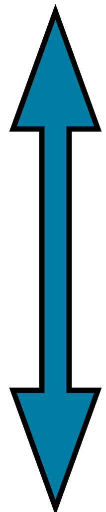
Whatever happens, don’t let your test suite break thinking, “I’ll go back and fix this later.”

Unit testing unittest



doctest

Functional/behavior testing



django.test.Client, Twill

Browser testing Windmill, Selenium

You need them all.

Testing Django

- Unit tests (`unittest`)
- Doctests (`doctest`)
- Fixtures
- Test client
- Email capture

Unit tests

- "Whitebox" testing
- Verify the small functional units of your app
- Very fine-grained
- Familier to most programmers (JUnit, NUnit, etc.)
- Provided in Python by `unittest`

django.test.TestCase

- Fixtures.
- Test client.
- Email capture.
- Database management.
- Slower than unittest.TestCase.

```
class StoryAddViewTests(TestCase):
    fixtures = ['authtestdata', 'newsbudget_test_data']
    urls = 'newsbudget.urls'

    def test_story_add_get(self):
        r = self.client.get('/budget/stories/add/')
        self.assertEqual(r.status_code, 200)
        …

    def test_story_add_post(self):
        data = {
            'title': 'Hungry cat is hungry',
            'date': '2009-01-01',
        }
        r = self.client.post('/budget/stories/add/', data)
        self.assertEqual(r.status_code, 302)
        …
```

Doctests

- Easy to write & read.
- Produces self-documenting code.
- Great for cases that only use assertEquals.
- Somewhere between unit tests and functional tests.
- Difficult to debug.
- Don’t always provide useful test failures.

```
class Choices(object):
    """
    Easy declarative "choices" tool::

        >>> STATUSES = Choices("Live", "Draft")

        # Acts like a choices list:
        >>> list(STATUSES)
        [(1, 'Live'), (2, 'Draft')]

        # Easily convert from code to verbose:
        >>> STATUSES.verbose(1)
        'Live'

        # ... and vice versa:
        >>> STATUSES.code("Draft")
        2

    """
    …
```

```
*********************************************
File "utils.py", line 150, in __main__.Choices
Failed example:
    STATUSES.verbose(1)
Expected:
    'Live'
Got:
    'Draft'
*********************************************
```

Functional tests

- a.k.a "Behavior Driven Development."
- "Blackbox," holistic testing.
- All the hardcore TDD folks look down on functional tests.
- But they keep your boss happy.
- Easy to find problems; harder to find the actual bug.

Functional testing tools

- django.test.Client
- webunit
- Twill
- ...

django.test.Client

- Test the whole request path without running a web server.
- Responses provide extra information about templates and their contexts.

```
class StoryAddViewTests(TestCase):
    fixtures = ['authtestdata', 'newsbudget_test_data']
    urls = 'newsbudget.urls'

    def test_story_add_get(self):
        r = self.client.get('/budget/stories/add/')
        self.assertEqual(r.status_code, 200)
     …

    def test_story_add_post(self):
        data = {
            'title': 'Hungry cat is hungry',
            'date': '2009-01-01',
        }
        r = self.client.post('/budget/stories/add/', data)
        self.assertEqual(r.status_code, 302)
        …
```

Web browser testing

- The ultimate in functional testing for web applications.
- Run test in a web browser.
- Can verify JavaScript, AJAX; even CSS.
- Test your site across supported browsers.

Browser testing tools

- Selenium
- Windmill

“Exotic” testing

- Static source analysis.
- Smoke testing (crawlers and spiders).
- Monkey testing.
- Load testing.
- ...

cockecounty

FAILED
Test MP Frontpage run at 2:49pm

semomarketplace

PASSED
Test MP Frontpage run at 2:49pm

ogden

PASSED
Test MP Frontpage run at 2:49pm

PAS
Test

gatehouse

PASSED
Test MP Frontpage run at 2:49pm

everythingmidmo

PASSED
Test MP Frontpage run at 2:49pm

marketplacedemo

FAILED
Test MP Frontpage run at 2:49pm

FAILED
Test MP Frontpage run at 2:49pm

PAS
Test

semoindiana

PASSED
Test MP Frontpage run at 2:49pm

postregistermarketplace

PASSED
Test MP Frontpage run at 2:49pm

ozark

PASSED
Test MP Frontpage run at 2:49pm

PAS
Test

gazlo

PASSED
Test MP Frontpage run at 2:49pm

amarillo

PASSED
Test MP Frontpage run at 2:49pm

salinafyi

PASSED
Test MP Frontpage run at 2:49pm

PAS
Test

wenatchee

PASSED
Test MP Frontpage run at 2:49pm

nea

PASSED
Test MP Frontpage run at 2:49pm

marketplacetraining

PASSED
Test MP Frontpage run at 2:49pm

PAS
Test

lancaster

PASSED
Test MP Frontpage run at 2:49pm

wonderstate

FAILED
Test MP Frontpage run at 2:49pm

mcminn

PASSED
Test MP Frontpage run at 2:49pm

PAS
Test

Further resources

- Windmill talk here at OSCON
  http://bit.ly/14tkrd
- Django testing documentation
  http://bit.ly/django-testing
- Python Testing Tools Taxonomy
  http://bit.ly/py-testing-tools

Structuring applications for reuse

Designing for reuse

- Do one thing, and do it well.
- Don't be afraid of multiple apps.
- Write for flexibility.
- Build to distribute.
- Extend carefully.

1.

Do one thing, and do it well.

Application == encapsulation

Focus

- Ask yourself: “What does this application do?”
- Answer should be one or two *short* sentences.

Good focus

- “Handle storage of users and authentication of their identities.”
- “Allow content to be tagged, del.icio.us style, with querying by tags.”
- “Handle entries in a weblog.”

Bad focus

- “Handle entries in a weblog, and users who post them, and their authentication, and tagging and categorization, and some flat pages for static content, and...”

Warning signs

- Lots of files.
- Lots of modules.
- Lots of models.
- Lots of code.

Small is good

- Many great Django apps are very small.
- Even a lot of “simple” Django sites commonly have a dozen or more applications in INSTALLED_APPS.
- If you’ve got a complex site and a short application list, something’s probably wrong.

Approach features skeptically

- What does the application do?
- Does this feature have anything to do with that?
- No? Don't add it.

2.

Don’t be afraid of many apps.

The monolith anti-pattern

- The “application” is the whole site.
- Re-use? YAGNI.
- Plugins that hook into the “main” application.
- Heavy use of middleware-like concepts.

(I blame Rails)

The Django mindset

- Application: some bit of functionality.
- Site: several applications.
- Spin off new “apps” liberally.
- Develop a suite of apps ready for when they’re needed.

Django encourages this

- INSTALLED_APPS
- Applications are just Python packages, not some Django-specific “app” or “plugin.”
- Abstractions like `django.contrib.sites` make you think about this as you develop.

Spin off a new app?

- Is this feature unrelated to the app's focus?
- Is it orthogonal to the rest of the app?
- Will I need similar functionality again?

The ideal:

I need a contact form

```
urlpatterns = ('',
    …
    (r'^contact/', include('contact_form.urls')),
    …
)
```

Done.

(http://bitbucket.org/ubernostrum/django-contact-form/)

But… what about…

- Site A wants a contact form that just collects a message.
- Site B’s marketing department wants a bunch of info.
- Site C wants to use Akismet to filter automated spam.

3.

Write for flexibility.

Common sense

- Sane defaults.
- Easy overrides.
- Don’t set anything in stone.

Forms

- Supply a form class.
- Let users specify their own.

Templates

- Specify a default template.
- Let users specify their own.

Form processing

- You want to redirect after successful submission.
- Supply a default URL.
    - (Preferably by using reverse resolution).
- Let users override the default.

```
def edit_entry(request, entry_id):
    form = EntryForm(request.POST or None)
    if form.is_valid():
        form.save()
        return redirect('entry_detail', entry_id)
    return render_to_response('entry/form.html', {…})
```

```
def edit_entry(request, entry_id,
               form_class=EntryForm,
               template_name='entry/form.html',
               post_save_redirect=None):

    form = form_class(request.POST or None)
    if form.is_valid():
        form.save()
        if post_save_redirect:
            return redirect(post_save_redirect)
        else:
            return redirect('entry_detail', entry_id)

    return render_to_response([template_name, 'entry/form.html'], {…})
```

URLs

- Provide a URLConf with all views.
- Use named URL patterns.
- Use reverse lookups (by name).

4.

Build to distribute (even private code).

What the tutorial teaches

```
myproject/
    settings.py
    urls.py

    myapp/
        models.py

    mysecondapp/
        views.py

    …
```

```
from myproject.myapp.models import …
from myproject. myapp.models import …

…

myproject.settings
myproject.urls
```

Project coupling kills re-use

Projects in real life.

- A settings module.
- A root URLConf.
- *Maybe* a `manage.py` (but…)
- And that’s it.

Advantages

- No assumptions about where things live.
- No `PYTHONPATH` magic.
- Reminds you that “projects” are just a Python module.

You don’t even need a project

ljworld.com:

- `worldonline.settings.ljworld`
- `worldonline.urls.ljworld`
- And a whole bunch of apps.

Where apps really live

- Single module directly on Python path (`registration`, `tagging`, etc.).
- Related modules under a top-level package (`ellington.events`, `ellington.podcasts`, etc.)
- No projects (`ellington.settings` doesn't exist).

Want to distribute?

- Build a package with distutils/setuptools.
- Put it on PyPI (or a private package server).
- Now it works with easy_install, pip, buildout, …

General best practices

- Establish dependancy rules.
- Establish a minimum Python version (suggestion: Python 2.5).
- Establish a minimum Django version (suggestion: Django 1.0).
- Test frequently against new versions of dependancies.

Document obsessively.

5.

Embrace and extend.

Don’t touch!

- Good applications are extensible without patching.
- Take advantage of every extensibility point an application gives you.
- You may end up doing something that deserves a new application anyway.

But this application wasn't meant to be extended!

Python Power!

Extending a view

- Wrap the view with your own code.
- Doing it repetitively? Write a decorator.

Extending a model

- Relate other models to it.
- Subclass it.
- Proxy subclasses (Django 1.1).

Extending a form

- Subclass it.
- There is no step 2.

Other tricks

- Signals lets you fire off customized behavior when certain events happen.
- Middleware offers full control over request/response handling.
- Context processors can make additional information available if a view doesn't.

If you must make changes to external code…

Keep changes to a minimum

- If possible, instead of adding a feature, add extensibility.
- Keep as much changed code as you can out of the original app.

Stay up-to-date

- Don’t want to get out of sync with the original version of the code!
- You might miss bugfixes.
- You might even miss the feature you needed.

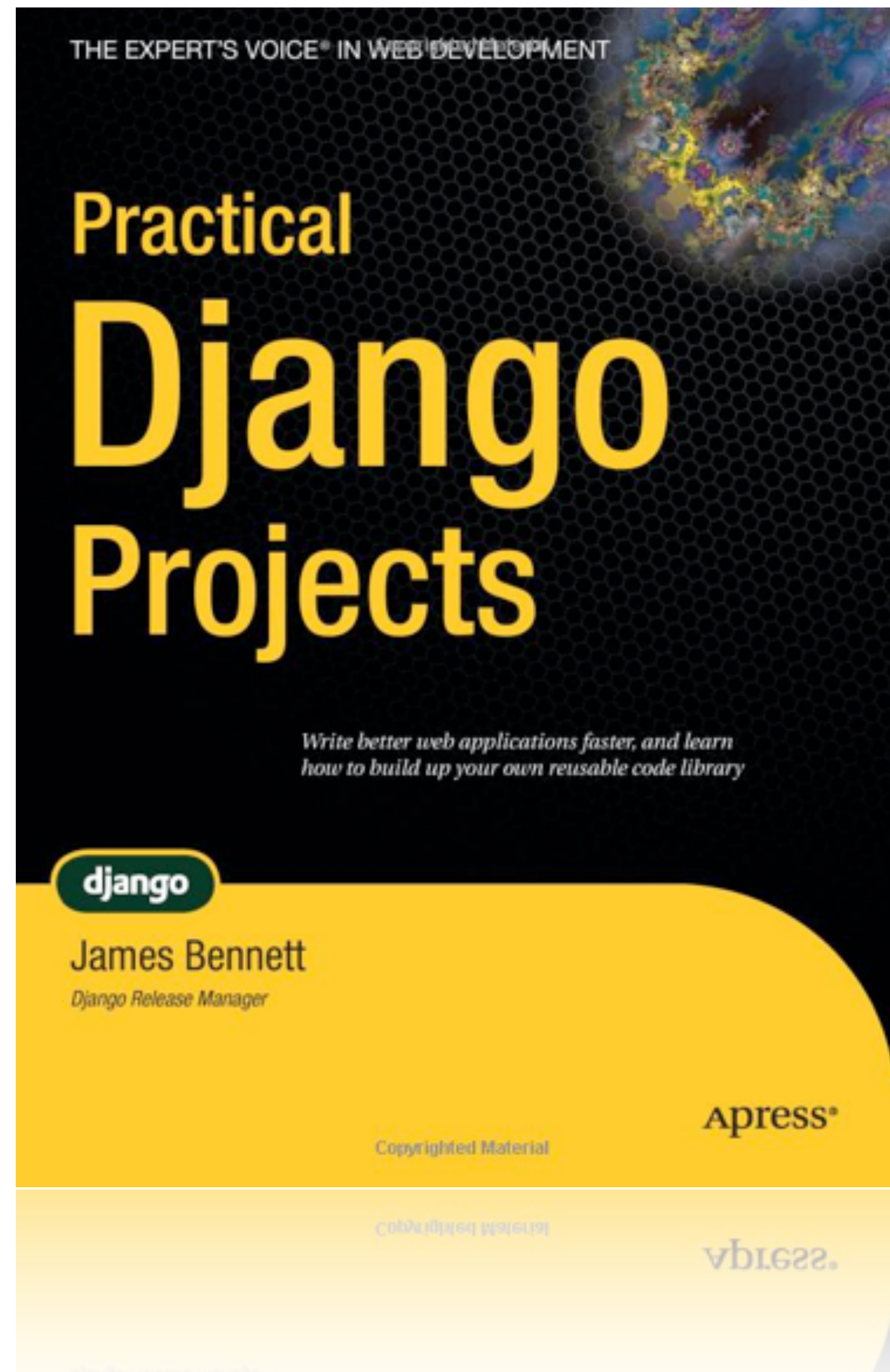
Use a good VCS

- Subversion vendor branches don’t cut it.
- DVCSes are *perfect* for this:
  - Mercurial queues.
  - Git rebasing.
- At the very least, maintain a patch queue by hand.

Be a good citizen

- If you change someone else's code, let them know.
- Maybe they'll merge your changes in and you won't have to fork anymore.

Further reading



Deployment

Deployment should...

- Be automated.
- Automatically manage dependencies.
- Be isolated.
- Be repeatable.
- Be identical in staging and in production.
- Work the same for everyone.

Dependency management	**Isolation**	**Automation**
apt/yum/...	virtualenv	Capistrano
easy_install	zc.buildout	Fabric
pip		Puppet/Chef/…
zc.buildout		

Dependancy management

- The Python ecosystem rocks!
- Python package management doesn't.
- Installing packages — and dependancies — correctly is a lot harder than it should be; most defaults are wrong.
- Here be dragons.

Vendor packages

- APT, Yum, …
- The good: familiar tools; stability; handles dependancies not on PyPI.
- The bad: small selection; not (very) portable; hard to supply user packages.
- The ugly: **installs packages system-wide**.

easy_install

- The good: multi-version packages.
- The bad: requires ‘net connection; can’t uninstall; can’t handle non-PyPI packages; multi-version packages barely work.
- The ugly: stale; unsupported; defaults almost totally wrong; **installs system-wide**.

pip

http://pip.openplans.org/

- “Pip Installs Packages”
- The good: Just Works™; handles non-PyPI packages (including direct from SCM); repeatable dependancies; integrates with virtualenv for isolation.
- The bad: still young; not yet bundled.
- The ugly: haven’t found it yet.

zc.buildout

http://buildout.org/

- The good: incredibly flexible; handles any sort of dependancy; repeatable builds; reusable “recipes;” good ecosystem; handles isolation, too.
- The bad: often cryptic, INI-style configuration file; confusing duplication of recipes; sometimes *too* flexible.
- The ugly: nearly completely undocumented.

Package isolation

- Why?
  - Site A requires Foo v1.0; site B requires Foo v2.0.
  - You need to develop against multiple versions of dependancies.

Package isolation tools

- Virtual machines (Xen, VMWare, EC2, …)
- Multiple Python installations.
- “Virtual” Python installations.
  - virtualenv
    http://pypi.python.org/pypi/virtualenv
  - zc.buildout
    http://buildout.org/

Why automate?

- “I can’t push this fix to the servers until Alex gets back from lunch.”
- “Sorry, I can’t fix that. I’m new here.”
- “Oops, I just made the wrong version of our site live.”
- “It’s broken! What’d you do!?”

Automation basics

- SSH is right out.
- Don’t futz with the server. Write a recipe.
- Deploys should be idempotent.

Capistrano

http://capify.org/

- The good: lots of features; good documentation; active community.
- The bad: stale development; very “opinionated” and Rails-oriented.

Fabric

http://fabfile.org/

- The good: very simple; flexible; actively developed; Python.
- The bad: no high-level commands; in flux.

Configuration management

- CFEngine, Puppet, Chef, …
- Will handle a *lot* more than code deployment!
- I only know a little about these.

Recommendations

- ● Pip, Virtualenv, and Fabric
- ■ Buildout and Fabric.
- ◆ Buildout and Puppet/Chef/….
- ◆◆ Utility computing and Puppet/Chef/….

Production environments

net.

LiveJournal Backend: Today

(Roughly.)

BIG-IP
- bigip1
- bigip2

djabberd
- djabberd
- djabberd

perlbal (httpd/proxy)
- proxy1
- proxy2
- proxy3
- proxy4
- proxy5

mod_perl
- web1
- web2
- web3
- web4
- ...
- webN

Memcached
- mc1
- mc2
- mc3
- mc4
- ...
- mcN

Global Database
- master_a
- master_b
- slave1
- slave2
- ...
- slave5

User DB Cluster 1
- uc1a ↔ uc1b

User DB Cluster 2
- uc2a ↔ uc2b

User DB Cluster 3
- uc3a ↔ uc3b

User DB Cluster N
- ucNa ↔ ucNb

Job Queues (xN)
- jqNa ↔ jqNb

gearmand
- gearmand1
- gearmandN

“workers”
- gearwrkN
- theschwkN

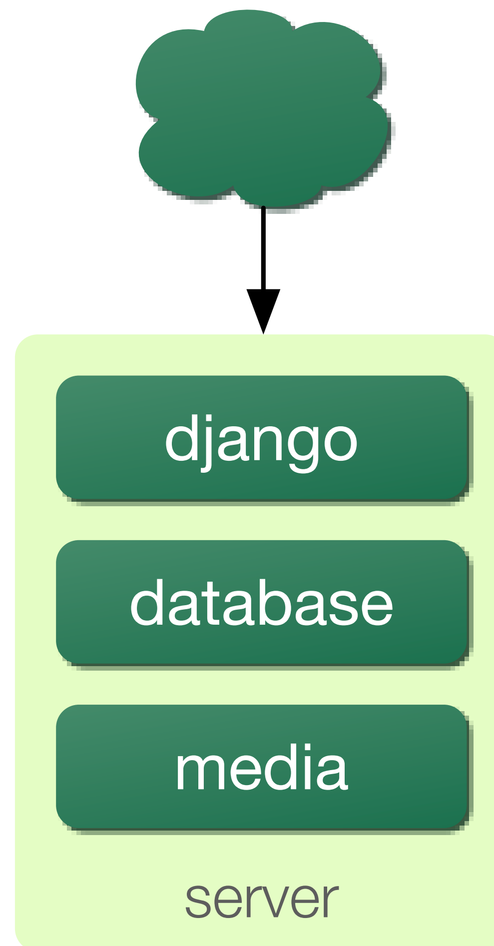
Mogile Storage Nodes
- sto1
- sto2
- ...
- sto8

Mogile Trackers
- tracker1
- tracker3

MogileFS Database
- mog_a
- mog_b
- slave1
- slaveN

Brad Fitzpatrik, http://danga.com/words/2007_06_usenix/

REVSYS revolution systems


django
database
media
server

Application servers

- Apache + mod_python
- Apache + mod_wsgi
- Apache/lighttpd + FastCGI
- SCGI, AJP, nginx/mod_wsgi, ...

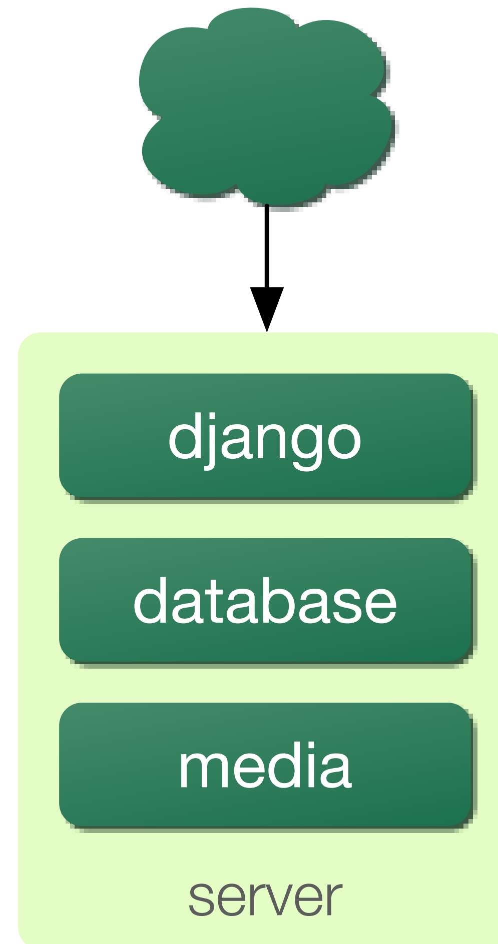
Use mod_wsgi

```
WSGIScriptAlias / /home/mysite/mysite.wsgi
```

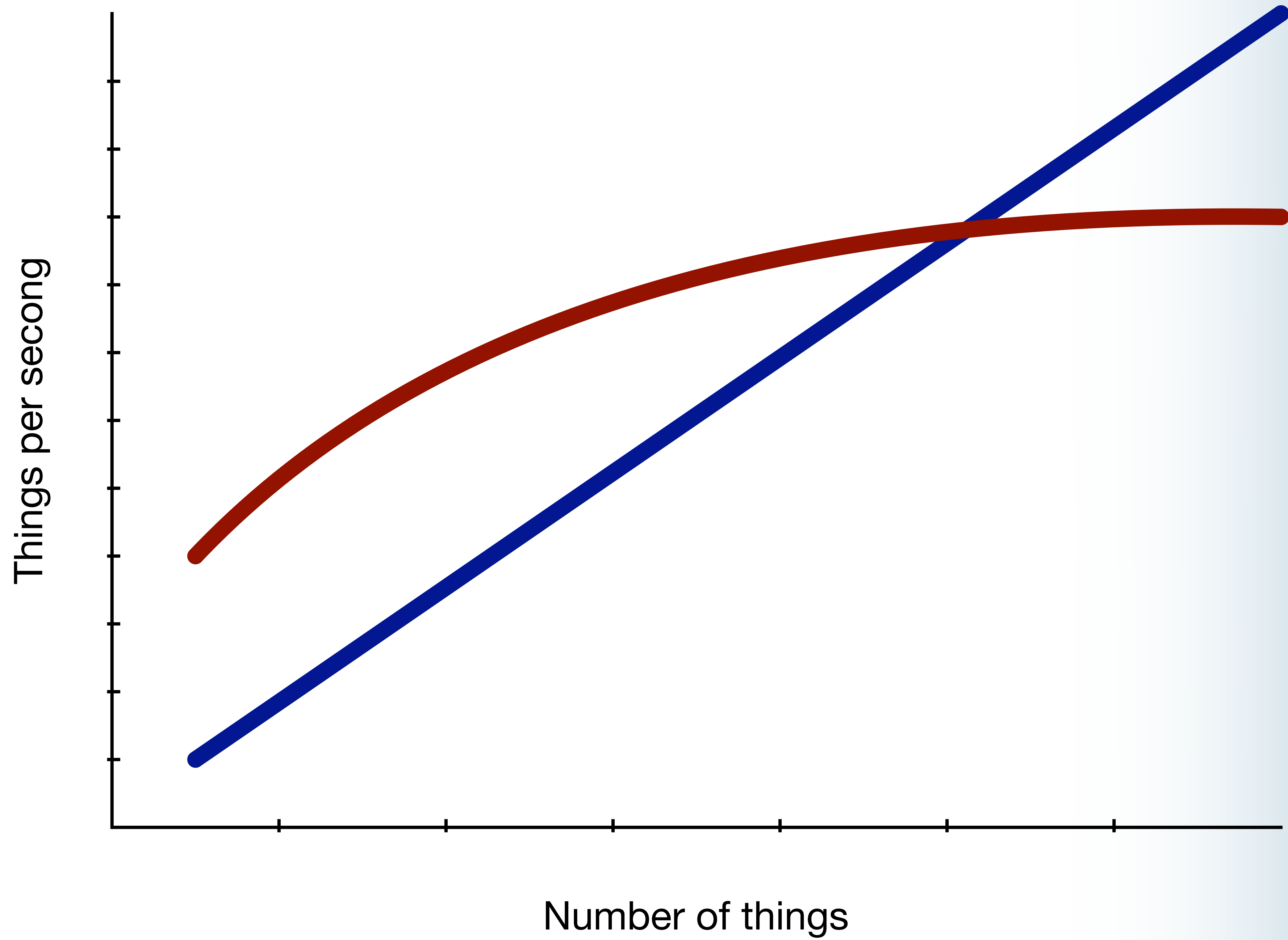
```
import os, sys

# Add to PYTHONPATH whatever you need
sys.path.append('/usr/local/django')

# Set DJANGO_SETTINGS_MODULE
os.environ['DJANGO_SETTINGS_MODULE'] = 'mysite.settings'

# Create the application for mod_wsgi
import django.core.handlers.wsgi
application = django.core.handlers.wsgi.WSGIHandler()
```

“Scale”

Does this scale?



Maybe!


Things per secong
Number of things

Real-world example

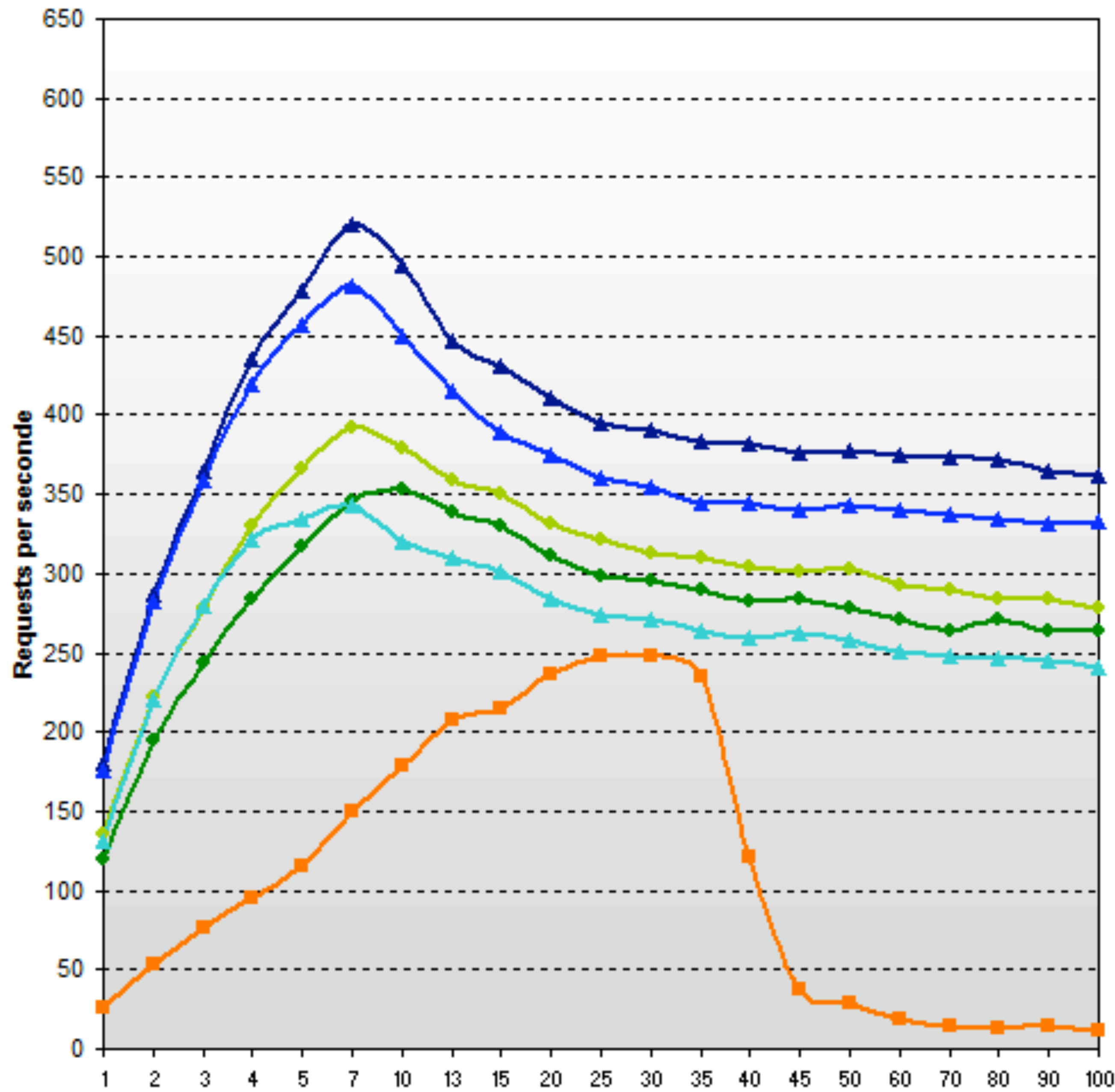
Database A

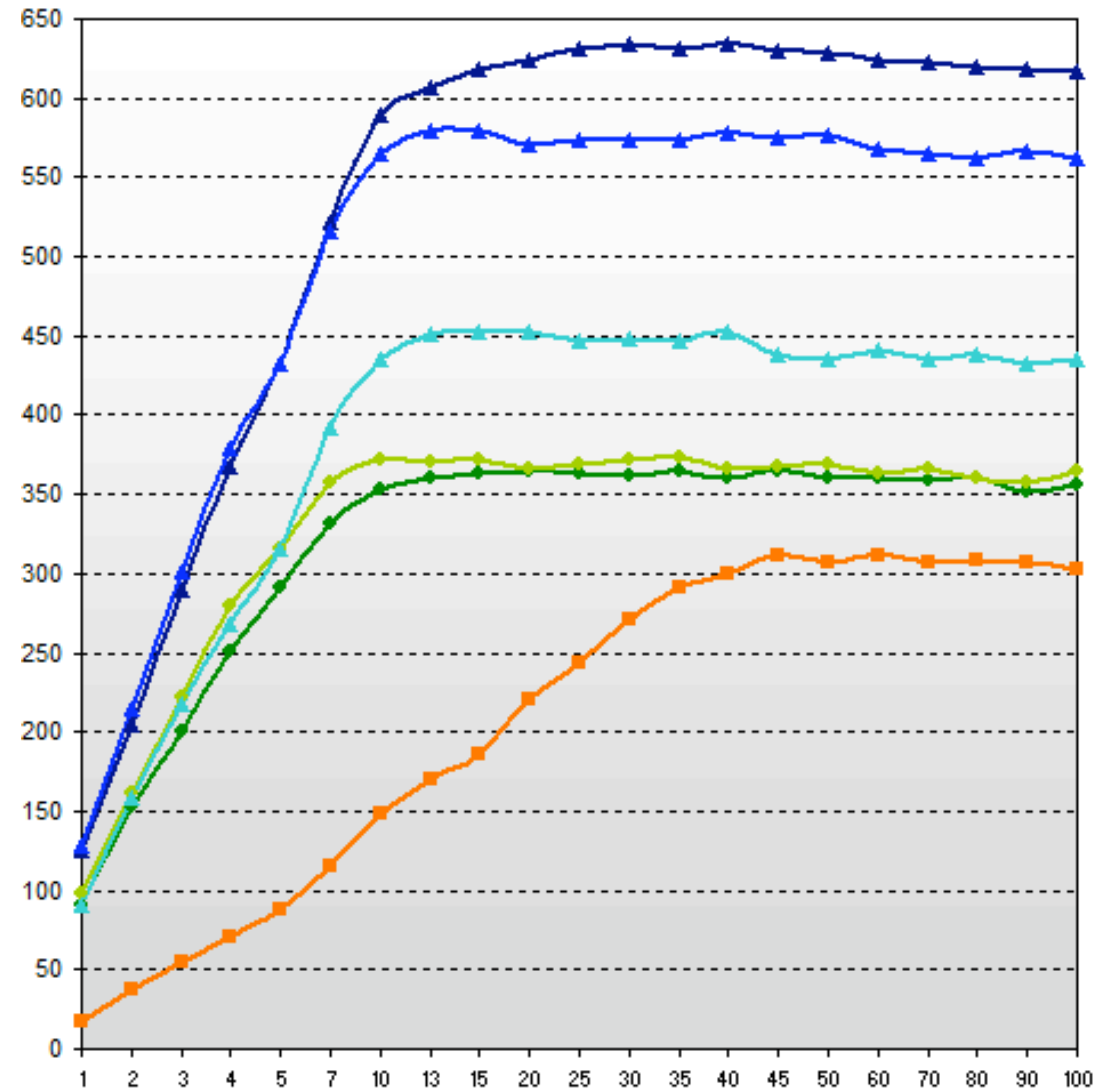
175 req/s

Database B

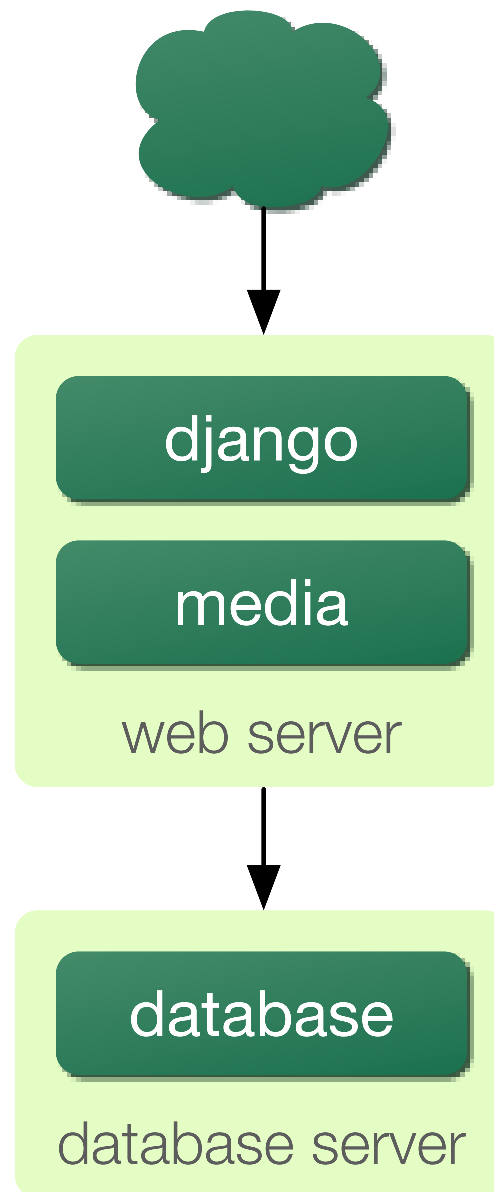
75 req/s

Real-world example





http://tweakers.net/reviews/657/6


django
media
web server
database
database server

Why separate hardware?

- Resource contention
- Separate performance concerns
- 0 → 1 is much harder than 1 → N

```
DATABASE_HOST = '10.0.0.100'
```

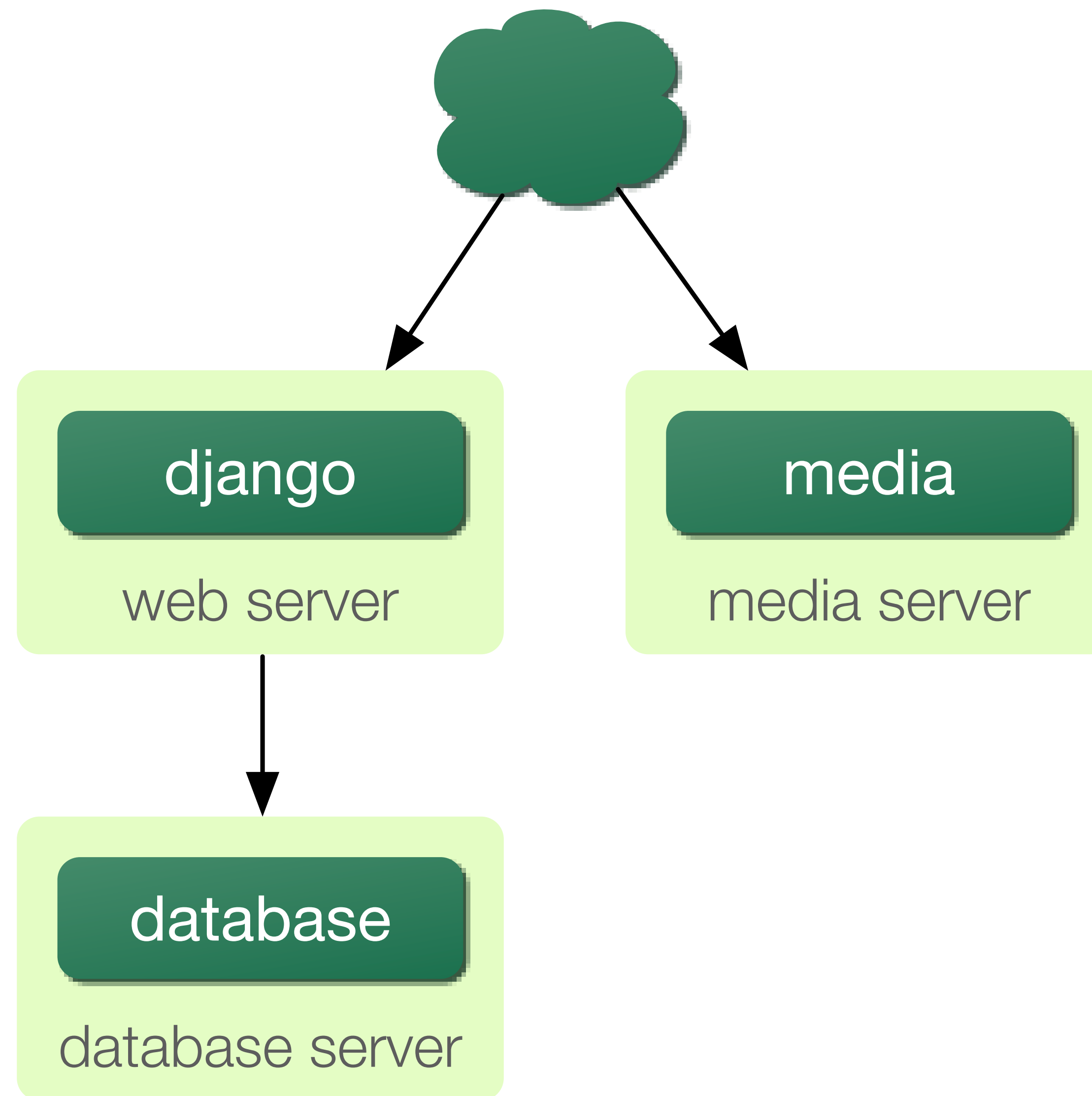
FAIL

Connection middleware

- Proxy between web and database layers
- Most implement hot fallover and connection pooling
  - Some also provide replication, load balancing, parallel queries, connection limiting, &c
- `DATABASE_HOST = '127.0.0.1'`

Connection middleware

- PostgreSQL: pgpool
- MySQL: MySQL Proxy
- Database-agnostic: sqlrelay
- Oracle: ?


django
web server
media
media server
database
database server

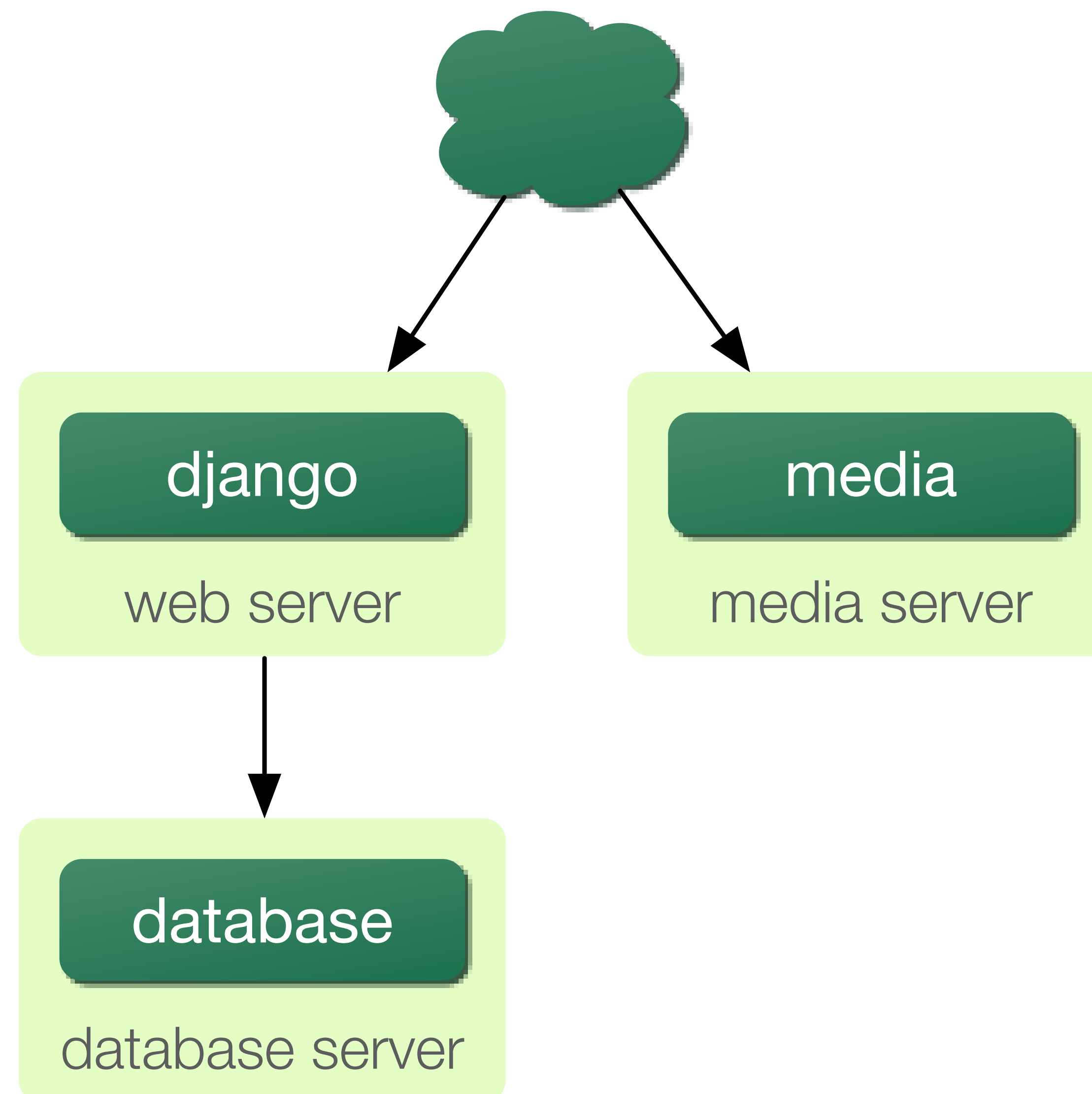
Media server traits

- Fast
- Lightweight
- Optimized for high concurrency
- Low memory overhead
- Good HTTP citizen

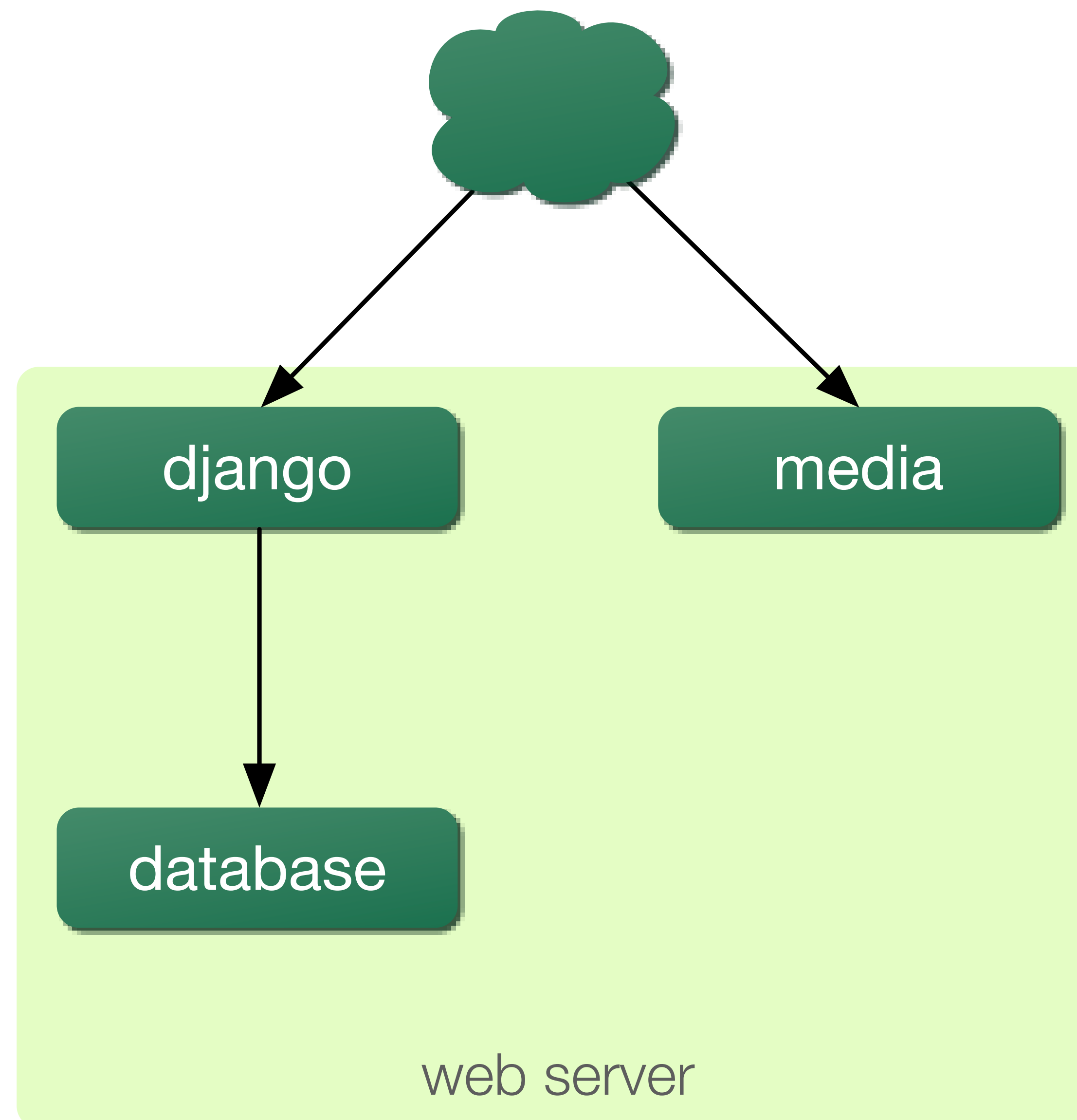
Media servers

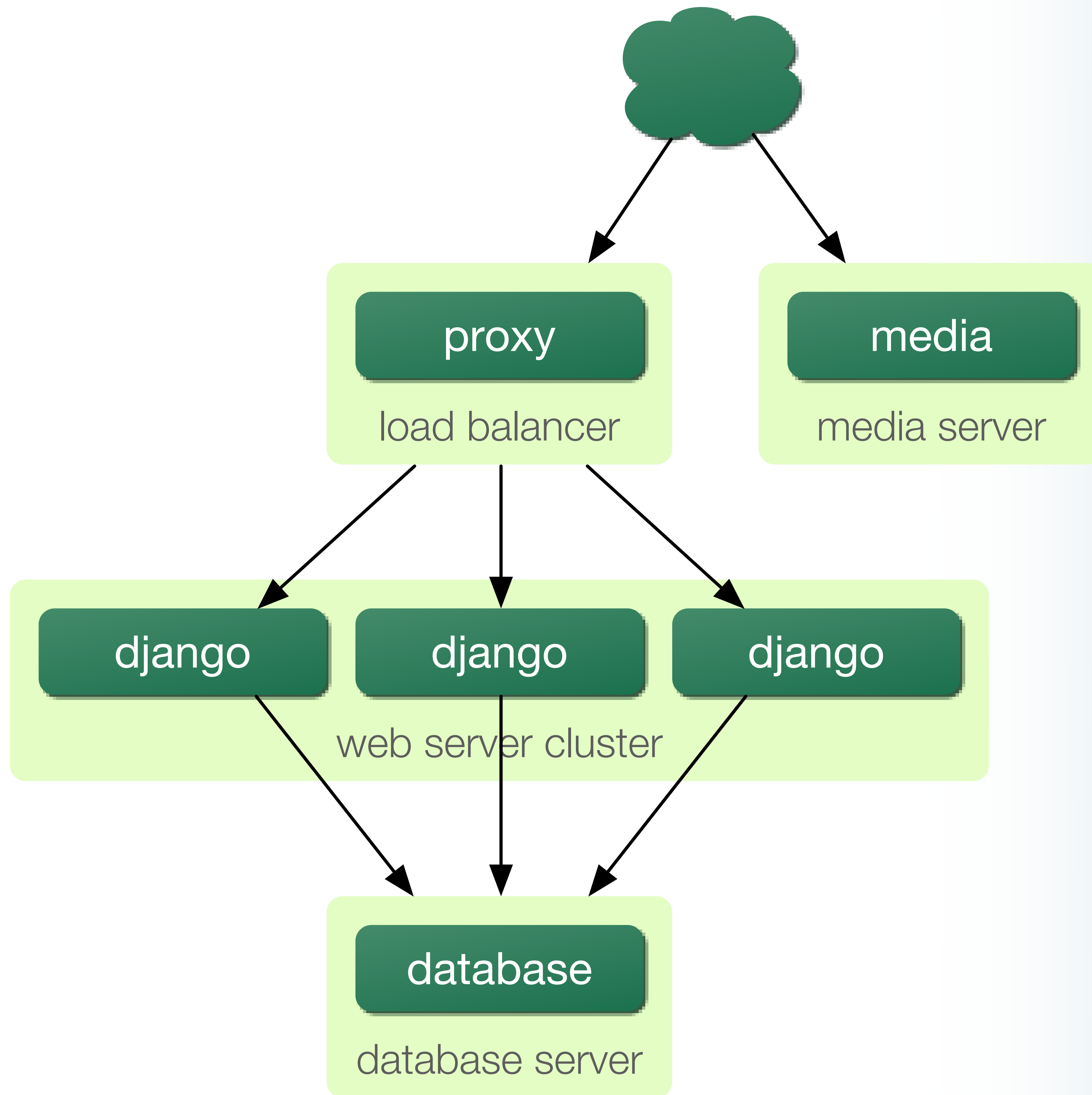
- Apache?
- lighttpd
- nginx
- S3

The absolute minimum



The absolute minimum




proxy
load balancer
media
media server
django
django
django
web server cluster
database
database server

Why load balancers?

Load balancer traits

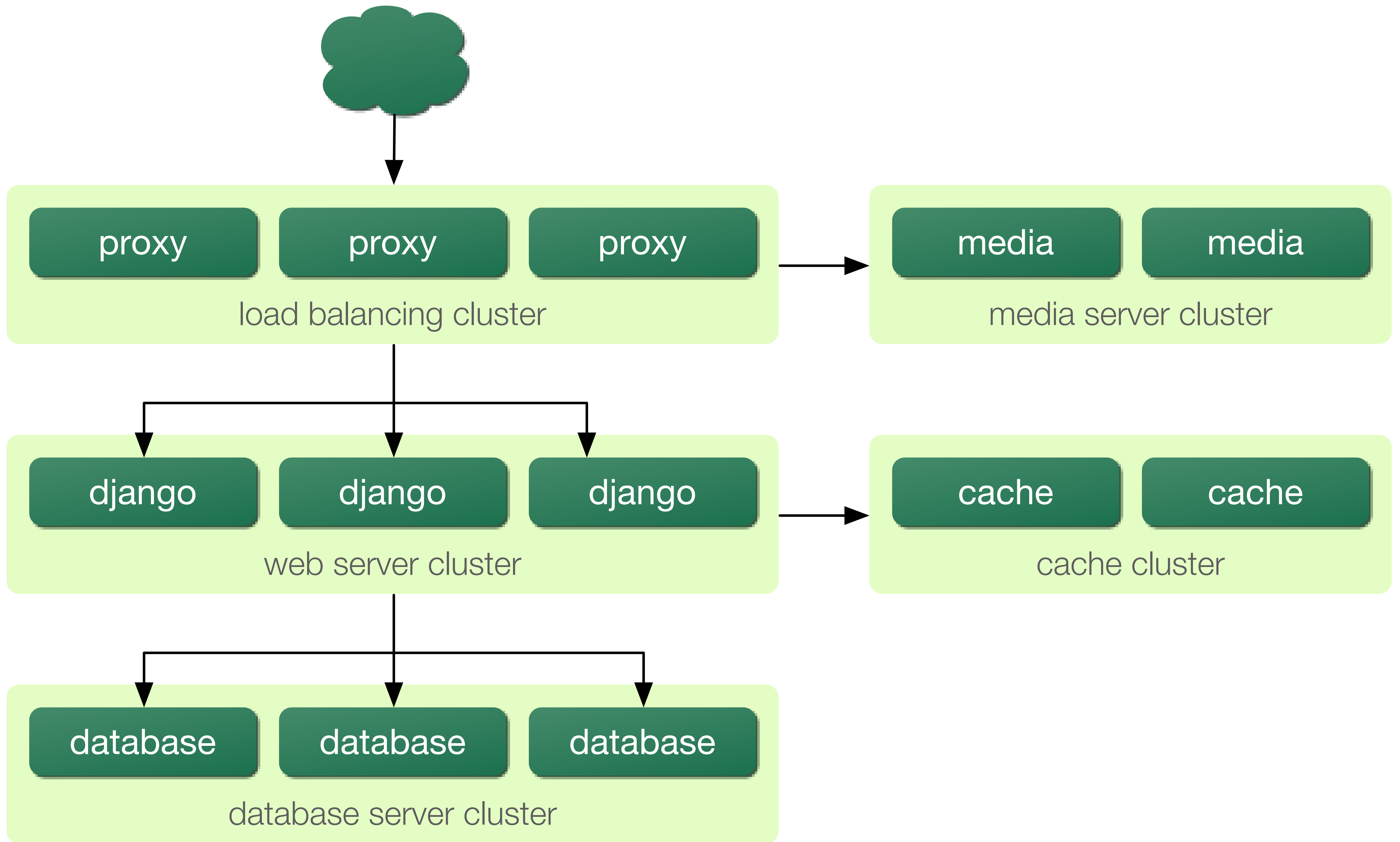
- Low memory overhead
- High concurrency
- Hot fallover
- Other nifty features...

Load balancers

- Apache + mod_proxy
- perlbal
- nginx
- Varnish
- Squid

```
CREATE POOL mypool
    POOL mypool ADD 10.0.0.100
    POOL mypool ADD 10.0.0.101

CREATE SERVICE mysite
    SET listen = my.public.ip
    SET role = reverse_proxy
    SET pool = mypool
    SET verify_backend = on
    SET buffer_size = 120k
ENABLE mysite
```

```
you@yourserver:~$ telnet localhost 60000

pool mysite add 10.0.0.102
OK

nodes 10.0.0.101
10.0.0.101 lastresponse 1237987449
10.0.0.101 requests 97554563
10.0.0.101 connects 129242435
10.0.0.101 lastconnect 1237987449
10.0.0.101 attempts 129244743
10.0.0.101 responsecodes 200 358
10.0.0.101 responsecodes 302 14
10.0.0.101 responsecodes 207 99
10.0.0.101 responsecodes 301 11
10.0.0.101 responsecodes 404 18
10.0.0.101 lastattempt 1237987449
```


proxy
proxy
proxy
load balancing cluster
media
media
media server cluster
django
django
django
web server cluster
cache
cache
cache cluster
database
database
database
database server cluster

“Shared nothing”

```
BALANCE = None

def balance_sheet(request):
    global BALANCE
    if not BALANCE:
        bank = Bank.objects.get(...)
        BALANCE = bank.total_balance()
    ...
```

FAIL

Global variables are right out

```
from django.cache import cache

def balance_sheet(request):
    balance = cache.get('bank_balance')
    if not balance:
        bank = Bank.objects.get(...)
        balance = bank.total_balance()
        cache.set('bank_balance', balance)

    ...
```

WIN

```
def generate_report(request):
    report = get_the_report()
    open('/tmp/report.txt', 'w').write(report)
    return redirect(view_report)

def view_report(request):
    report = open('/tmp/report.txt').read()
    return HttpResponse(report)
```

FAIL

Filesystem?
What filesystem?

Further reading

- Cal Henderson, *Building Scalable Web Sites*
- John Allspaw, *The Art of Capacity Planning*
- http://kitchensoap.com/
- http://highscalability.com/

Monitoring

Goals

- When the site goes down, know it immediately.
- Automatically handle common sources of downtime.
- Ideally, handle downtime before it even happens.
- Monitor hardware usage to identify hotspots and plan for future growth.
- Aid in postmortem analysis.
- Generate pretty graphs.

Availability monitoring principles

- Check services for availability.
- More then just “ping yoursite.com.”
- Have some understanding of dependancies.
- Notify the “right” people using the “right” methods, and don’t stop until it’s fixed.
- Minimize false positives.
- Automatically take action against common sources of downtime.

Availability monitoring tools

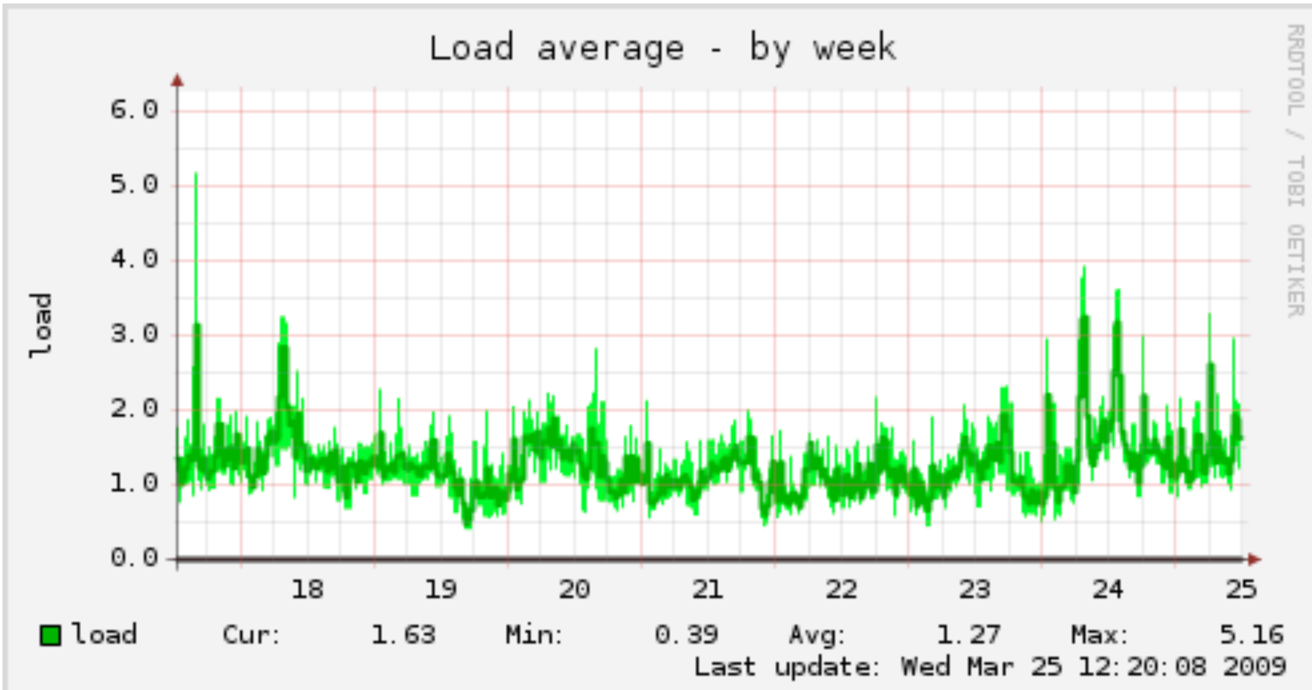
- Internal tools
  - Nagios
  - Monit
  - Zenoss
  - ...
- External monitoring tools

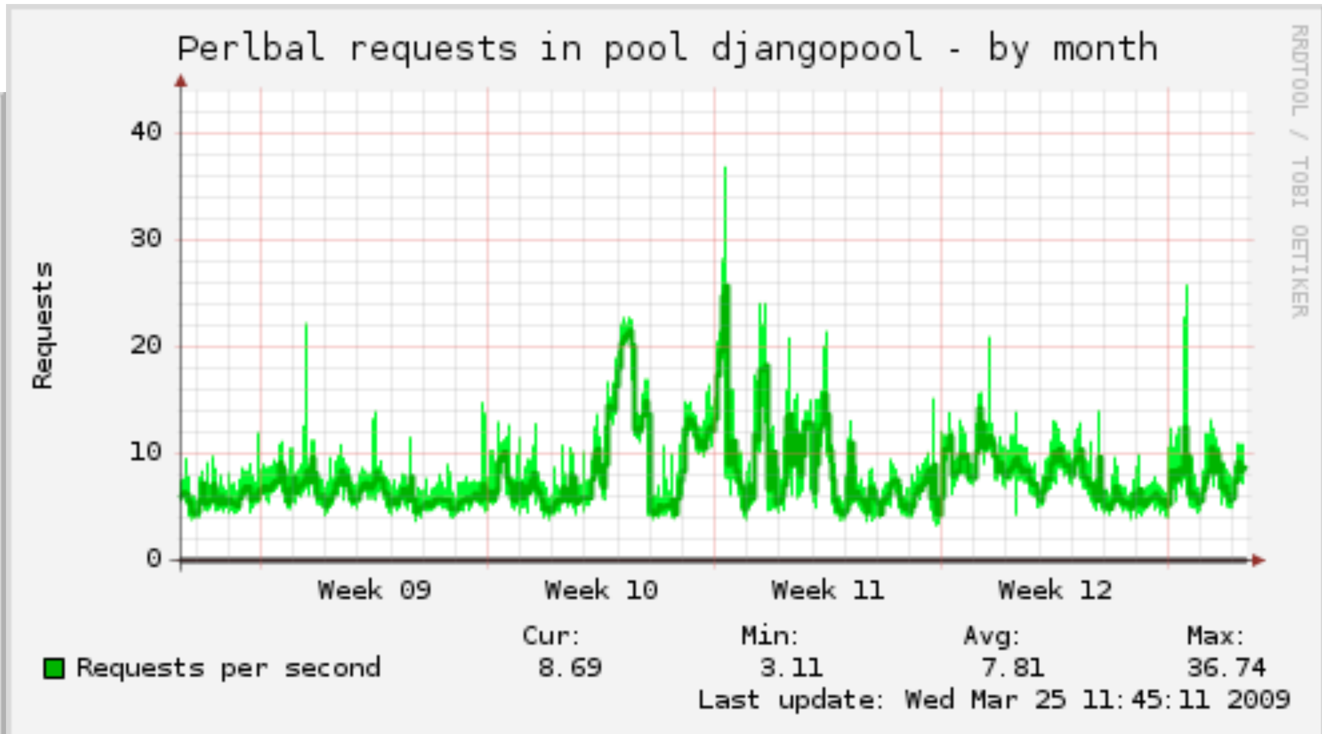
Usage monitoring

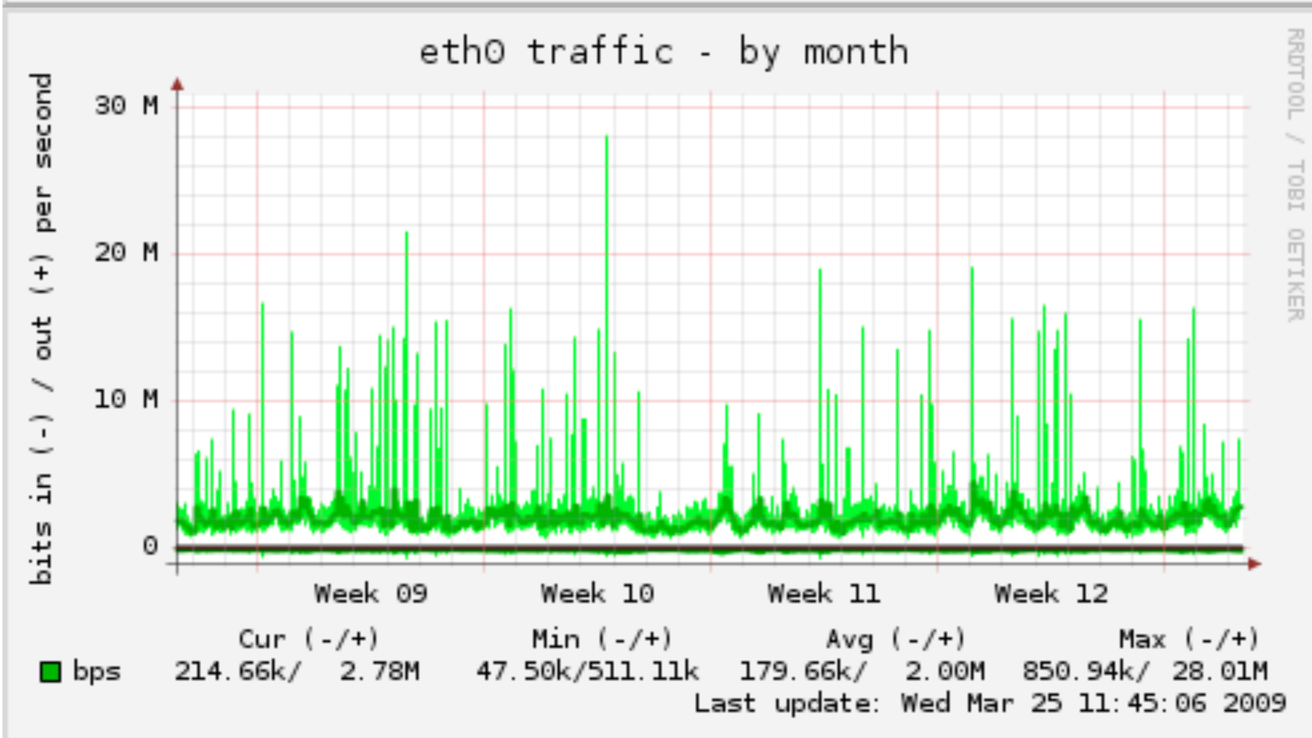
- Keep track of resource usage over time.
- Spot and identify trends.
- Aid in capacity planning and management.
- Look good in reports to your boss.

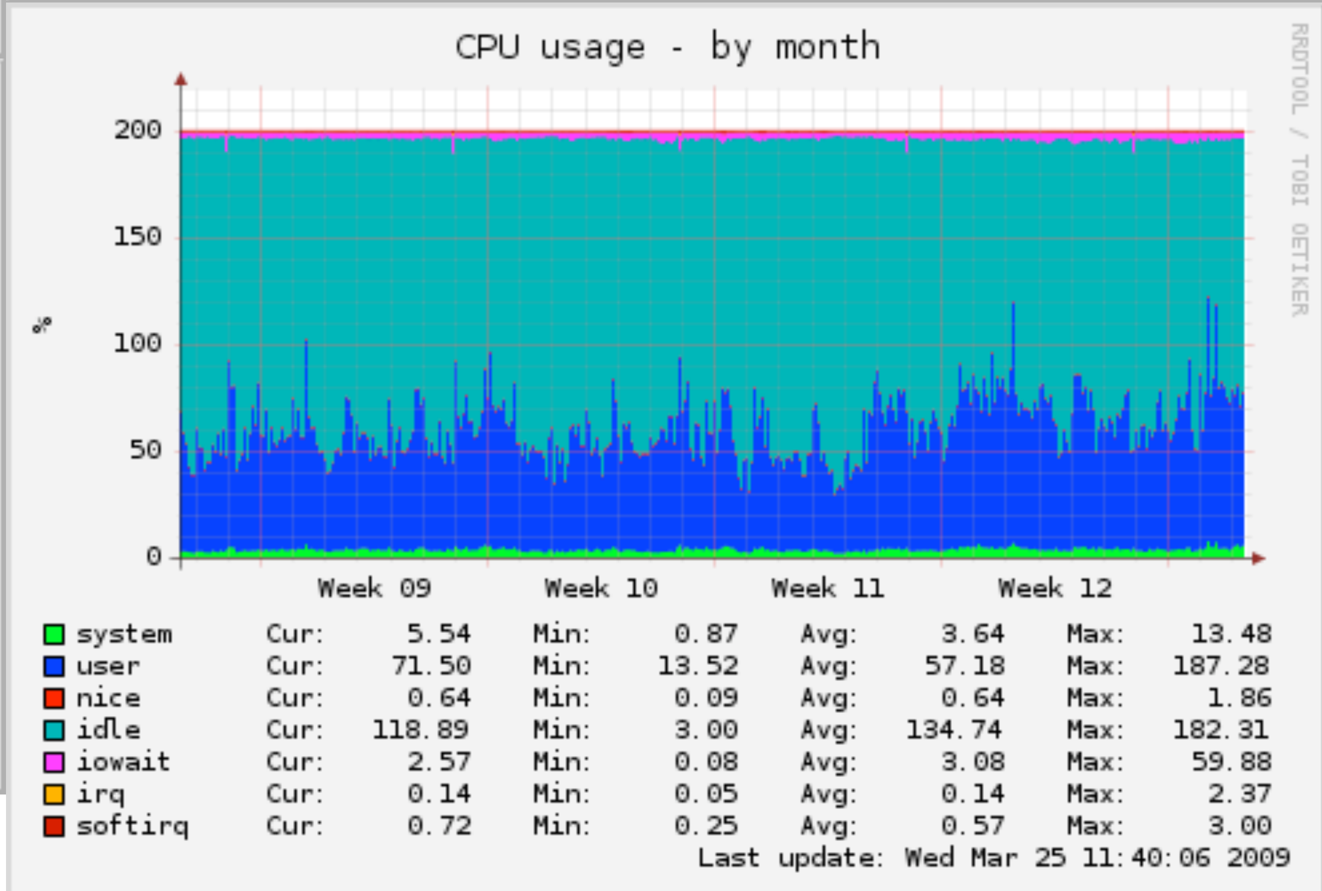
Usage monitoring tools

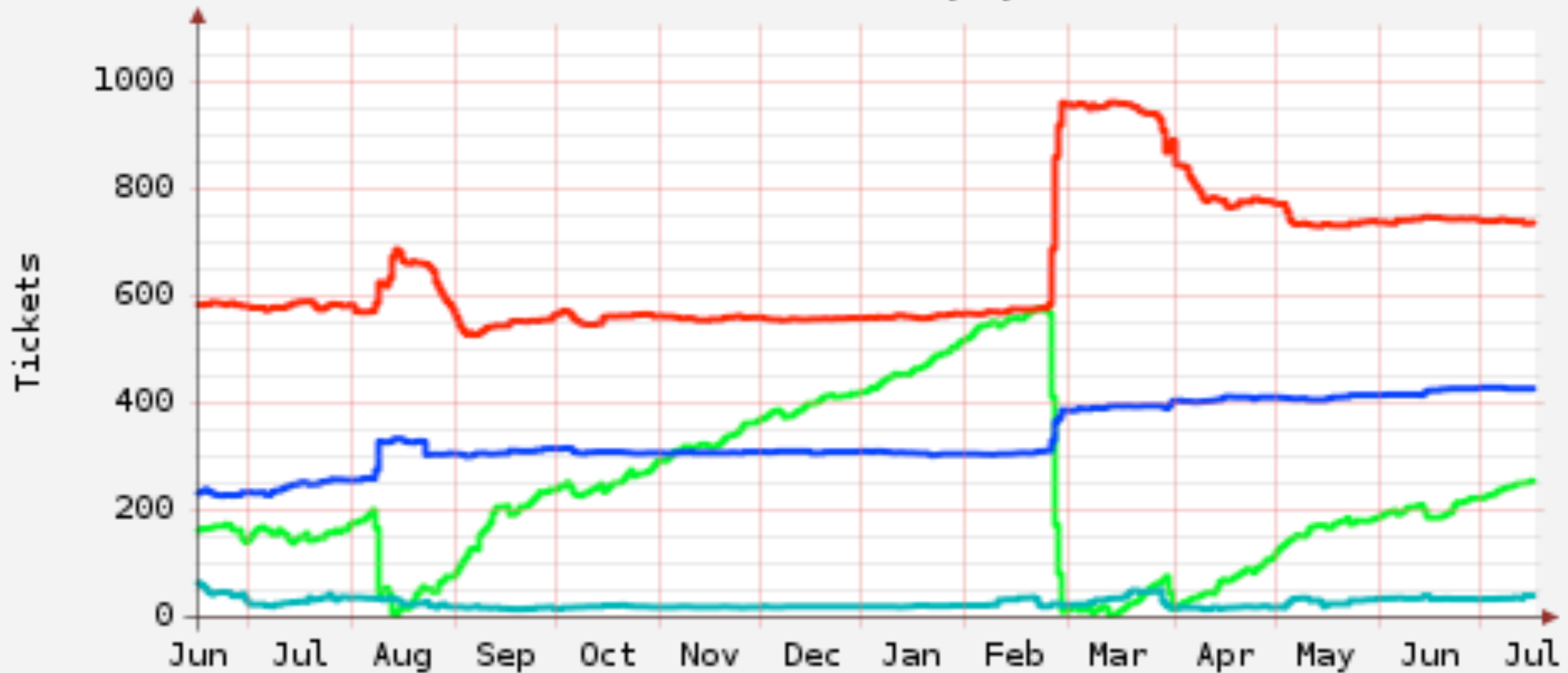
- RRDTool
- Munin
- Cacti
- Graphite


Load average - by week
load
6.0
5.0
4.0
3.0
2.0
1.0
0.0
18 19 20 21 22 23 24 25
load Cur: 1.63 Min: 0.39 Avg: 1.27 Max: 5.16
Last update: Wed Mar 25 12:20:08 2009
RRDTOOL / TOBI OETIKER


Perlbal requests in pool djangopool - by month
Requests
40
30
20
10
0
Week 09 Week 10 Week 11 Week 12
Requests per second Cur: 8.69 Min: 3.11 Avg: 7.81 Max: 36.74
Last update: Wed Mar 25 11:45:11 2009
RRDTOOL / TOBI OETIKER


eth0 traffic - by month
bits in (-) / out (+) per second
30 M
20 M
10 M
0
Week 09 Week 10 Week 11 Week 12
bps Cur (-/+) 214.66k/ 2.78M Min (-/+) 47.50k/511.11k Avg (-/+) 179.66k/ 2.00M Max (-/+) 850.94k/ 28.01M
Last update: Wed Mar 25 11:45:06 2009
RRDTOOL / TOBI OETIKER


CPU usage - by month
%
200
150
100
50
0
Week 09 Week 10 Week 11 Week 12
system Cur: 5.54 Min: 0.87 Avg: 3.64 Max: 13.48
user Cur: 71.50 Min: 13.52 Avg: 57.18 Max: 187.28
nice Cur: 0.64 Min: 0.09 Avg: 0.64 Max: 1.86
idle Cur: 118.89 Min: 3.00 Avg: 134.74 Max: 182.31
iowait Cur: 2.57 Min: 0.08 Avg: 3.08 Max: 59.88
irq Cur: 0.14 Min: 0.05 Avg: 0.14 Max: 2.37
softirq Cur: 0.72 Min: 0.25 Avg: 0.57 Max: 3.00
Last update: Wed Mar 25 11:40:06 2009
RRDTOOL / TOBI OETIKER



Trac tickets - by year
RRDTOOL / TOBI OETIKER
Tickets
1000
800
600
400
200
0
Jun
Jul
Aug
Sep
Oct
Nov
Dec
Jan
Feb
Mar
Apr
May
Jun
Jul
unreviewed Cur: 254.59 Min: 0.00 Avg: 230.63 Max: 581.00
design Cur: 427.00 Min: 226.00 Avg: 335.88 Max: 429.00
accepted Cur: 736.77 Min: 526.79 Avg: 653.95 Max: 969.00
ready Cur: 40.00 Min: 14.96 Avg: 26.38 Max: 82.00
Last update: Mon Jul 20 11:35:10 2009

Logging

- Record information about what's happening right now.
- Analyze historical data for trends.
- Provide postmortem information after failures.

Logging tools

- print
- Python’s logging module
- syslogd

Log analysis

- `grep | sort | uniq -c | sort -rn`
- Load log data into relational databases, then slice & dice.
- OLAP/OLTP engines.
- Splunk.
- Analog, AWStats, ...
- Google Analytics, Mint, ...

What to monitor?

- Everything possible.
- The answer to “should I monitor this?” is always “yes.”

Performance

And when you should care.

Ignore performance

Step 1: write your app.

Step 2: make it work.

Step 3: get it live.

Step 4: get some users.

…

Step 94,211: tune.

Ignore performance

- Code isn’t “fast” or “slow” until it’s deployed in production.
- That said, often bad code is obvious. So don’t write it.
- YAGNI doesn’t mean you get to be an idiot.

Low-hanging fruit

- Lots of DB queries.
- Rule of thumb: O(1) queries per view.
- Very complex queries.
- Read-heavy vs. write-heavy.

Anticipate bottlenecks

- It’s probably going to be your DB.
- If not, it’ll be I/O.

“It’s slow!”

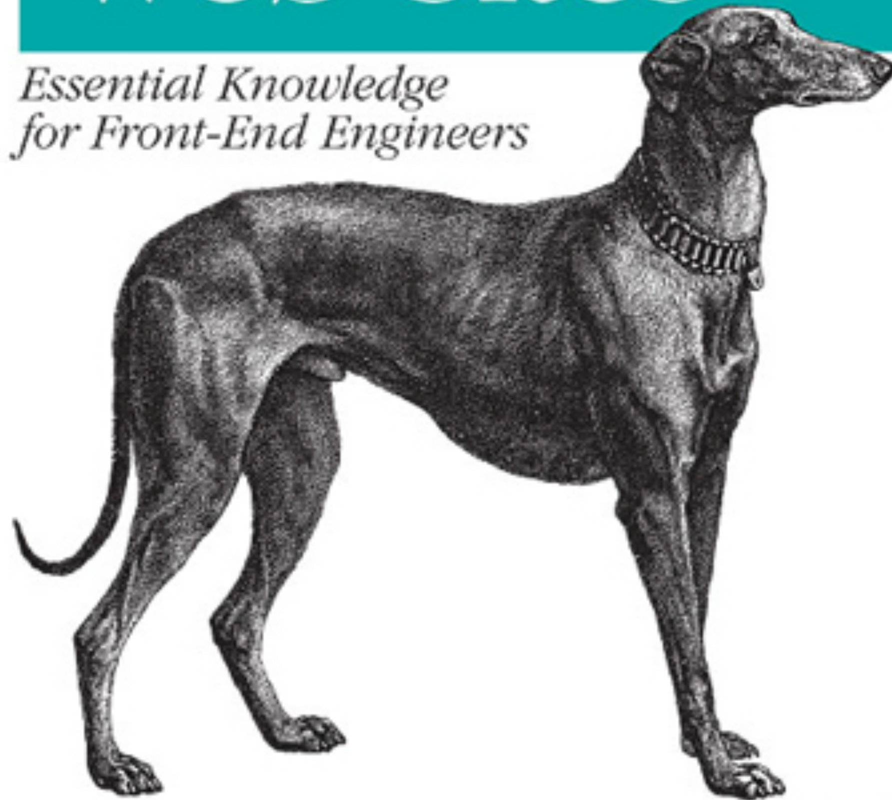
Define "slow"

- Benchmark in the browser.
- Compare to wget/curl.
- The results can be surprising.
- Often, "slow" is a matter of *perceived* performance.

14 Steps to Faster-Loading Web Sites

High Performance Web Sites

Essential Knowledge for Front-End Engineers



O'REILLY®

Steve Souders
Foreword by Nate Koechley

Essential Knowlege for Frontend Engineers

Even Faster Web Sites

O'REILLY®

Steve Souders

YSlow

http://developer.yahoo.com/yslow/

Server-side performance tuning

Tuning in a nutshell

- Cache.
- Cache some more.
- Improve your caching strategy.
- Add more cache layers.
- Then, *maybe,* tune your code.

Caching is magic

- Turns less hardware into more!
- Makes slow code fast!
- Lowers hardware budgets!
- Delays the need for new servers!
- Cures scurvy!

Caching is about trade-offs

Caching questions

- Cache for everybody? Only logged-in users? Only non-paying users?
- Long timeouts/stale data? Short timeouts/ worse performance?
- Invalidation: time-based? Data based? Both?
- Just cache everything? Or just some views? Or just the expensive parts?
- Django's cache layer? Proxy caches?

Common caching strategies

- Are most of your users anonymous? Use `CACHE_MIDDLEWARE_ANONYMOUS_ONLY`
- Are there just a couple of slow views? Use `@cache_page`.
- Need to cache everything? Use a site wide cache.
- Everything except a few views? Use `@never_cache`.

Site-wide caches

- Good: Django's cache middleware.
- Better: A proper upstream cache. (Squid, Varnish, …).

External caches

- Most work well with Django.
- Internally, Django just uses HTTP headers to control caching; those headers are exposed to external caches.
- Cached requests never even hit Django.

Conditional view processing

```
GET / HTTP/1.1
Host: www2.ljworld.com/

            HTTP/1.1 200 OK
            Server: Apache
            Expires: Wed, 17 Jun 2009 18:17:18 GMT
            ETag: "93431744c9097d4a3edd4580bf1204c4"
            …

GET / HTTP/1.1
Host: www2.ljworld.com/
If-None-Match: "93431744c9097d4a3edd4580bf1204c4"

            HTTP/1.1 304 NOT MODIFIED
            …

GET / HTTP/1.1
Host: www2.ljworld.com/
If-Modified-Since: Wed, 17 Jun 2009 18:00:00 GMT

            HTTP/1.1 304 NOT MODIFIED
            …
```

Etags

- Opaque identifiers for a resource.
- Cheaper to compute than the resource itself.
- Bad: “17”, “some title”, etc.
- Good: “93431744c9097d4a3edd4580bf1204c4”, “74c05a20-5b6f-11de-adc7-001b63944e73”, etc.

When caching fails…

“I think I need a bigger box.”

Where to spend money

- First, buy more RAM.
- Then throw money at your DB.
- Then buy more web servers.

No money?

Web server improvements

- Start with simple improvements: turn off Keep-Alive, tweak MaxConnections; etc.
- Use a better application server (mod_wsgi).
- Investigate light-weight web servers (nginx, lighttpd).

Database tuning

- Whole books can be — and many have been — written about DB tuning.
- MySQL: *High Performance MySQL* http://www.amazon.com/dp/0596101716/
- PostgreSQL: http://www.revsys.com/writings/postgresql-performance.html

Build a toolkit

- `profile`, `cProfile`
- `strace`, SystemTap, `dtrace`.
- Django debug toolbar
  http://bit.ly/django-debug-toolbar

More…

http://jacobian.org/r/django-cache
http://jacobian.org/r/django-conditional-views

Final thoughts

- Writing the code is the easy part.
- Making it work in the Real World is that part that'll make you lose sleep.
- Don't worry too much: performance problems are good problems to have.
- But worry a little bit: "an ounce of prevention is worth a pound of cure."

Fin.

Contact me: jacob@jacobian.org / @jacobian

Hire me: http://revsys.com/