django

Django in the Real World

James Bennett • Jacob Kaplan-Moss

http://b-list.org/ http://jacobian.org/

PyCon 2009

http://jacobian.org/speaking/2009/real-world-django/

So you've written a web app…

Now what?

- API Metering
- Backups & Snapshots
- Counters
- Cloud/Cluster Management Tools
  - Instrumentation/Monitoring
  - Failover
  - Node addition/removal and hashing
  - Autoscaling 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

What’s on the plate

- ➜ Structuring for deployment
- ➜ Testing
- ➜ Production environments
- ➜ Deployment
- ➜ The “rest” of the web stack
- ➜ Monitoring
- ➜ Performance & tuning

Writing applications you can deploy, and deploy, and deploy…

The extended extended remix!

The fivefold path

- ➔ 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

Keep a tight 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...”
- ➔ The coding equivalent of a run-on sentence

Warning signs

- A lot of very good Django applications are very small: just a few files
- If your app is getting big enough to need lots of things split up into lots of modules, it may be time to step back and re-evaluate

Warning signs

- ➜ Even a lot of “simple” Django sites commonly have a dozen or more applications in `INSTALLED_APPS`
- ➜ If you’ve got a complex/feature-packed site and a short application list, it may be time to think hard about how tightly-focused those apps are

Approach features skeptically

Should I add this feature?

- ➔ What does the application do?
- ➔ Does this feature have anything to do with that?
- ➔ No? Guess I shouldn’t add it, then.

2

Don’t be afraid of multiple apps

The monolith mindset

- ➜ The “application” is the whole site
- ➜ Re-use is often an afterthought
- ➜ Tend to develop plugins that hook into the “main” application
- ➜ Or make heavy use of middleware-like concepts

The Django mindset

- Application == some bit of functionality
- Site == several applications
- Tend to spin off new applications liberally

Django encourages this

- ➜ Instead of one "application", a list: `INSTALLED_APPS`
- ➜ Applications live on the Python path, not inside any specific "apps" or "plugins" directory
- ➜ Abstractions like the `Site` model make you think about this as you develop

Should this be its own application?

- ➜ Is it completely unrelated to the app's focus?
- ➜ Is it orthogonal to whatever else I'm doing?
- ➜ Will I need similar functionality on other sites?
- ➜ Yes? Then I should break it out into a separate application.

Unrelated features

- ➜ Feature creep is tempting: “but wouldn’t it be cool if…”
- ➜ But it’s the road to Hell
- ➜ See also: Part 1 of this talk

I've learned this the hard way

djangosnippets.org

- ➔ One application
- ➔ Includes bookmarking features
- ➔ Includes tagging features
- ➔ Includes rating features

Should be about four applications

Orthogonality

- ➔ Means you can change one thing without affecting others
- ➔ Almost always indicates the need for a separate application
- ➔ Example: changing user profile workflow doesn’t affect user signup workflow. Make them two different applications.

Reuse

- ➜ Lots of cool features actually aren’t specific to one site
- ➜ See: bookmarking, tagging, rating...
- ➜ Why bring all this crap about code snippets along just to get the extra stuff?

Advantages

- ➜ Don’t keep rewriting features
- ➜ Drop things into other sites easily

Need a contact form?

```
urlpatterns += (‘’,
    (r’^contact/’, include(‘contact_form.urls’)),
)
```

And you’re done

But what about…

Site-specific needs

- 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

Form processing

- Supply a form class
- But let people specify their own if they want

Templates

- Specify a default template
- But let people specify their own if they want

Form processing

- You want to redirect after successful submission
- Supply a default URL
- But let people specify their own if they want

URL best practices

- Provide a URLConf in the application
- Use named URL patterns
- Use reverse lookups: `reverse()`, `permalink`, `{% url %}`

Working with models

- Whenever possible, avoid hard-coding a model class
- Use `get_model()` and take an app label/ model name string instead
- Don’t rely on `objects`; use the default manager

Working with models

- ➜ Don’t hard-code fields or table names; introspect the model to get those
- ➜ Accept lookup arguments you can pass straight through to the database API

Learn to love managers

- ➜ Managers are easy to reuse.
- ➜ Managers are easy to subclass and customize.
- ➜ Managers let you encapsulate patterns of behavior behind a nice API.

Advanced techniques

- ➜ Encourage subclassing and use of subclasses
- ➜ Provide a standard interface people can implement in place of your default implementation
- ➜ Use a registry (like the admin)

The API your application exposes is just as important as the design of the sites you'll use it in.

In fact, it's **more** important.

Good API design

- "Pass in a value for this argument to change the behavior"
- "Change the value of this setting"
- "Subclass this and override these methods to customize"
- "Implement something with this interface, and register it with the handler"

Bad API design

- “API? Let me see if we have one of those…” (AKA: “we don’t”)
- “It’s open source; fork it to do what you want” (AKA: “we hate you”)
- `def application(environ, start_response)` (AKA: “we have a web service”)

4

Build to distribute

So you did the tutorial

- ➔ `from mysite.polls.models import Poll`
- ➔ `mysite.polls.views.vote`
- ➔ `include(‘mysite.polls.urls’)`
- ➔ `mysite.mysite.bork.bork.bork`

Project coupling kills re-use

Why (some) projects suck

- ➔ You have to replicate that directory structure every time you re-use
- ➔ Or you have to do gymnastics with your Python path
- ➔ And you get back into the monolithic mindset

A good “project”

- A settings module
- A root URLConf module
- And that’s it.

Advantages

- ➜ No assumptions about where things live
- ➜ No tricky bits
- ➜ Reminds you that it's just another Python module

It doesn’t even have to be one module

ljworld.com

- ➔ `worldonline.settings.ljworld`
- ➔ `worldonline.urls.ljworld`
- ➔ And a whole bunch of reused apps in sensible locations

Configuration is contextual

What reusable apps look like

- ➜ Single module directly on Python path (`registration`, `tagging`, etc.)
- ➜ Related modules under a package (`ellington.events`, `ellington.podcasts`, etc.)
- ➜ No project cruft whatsoever

And now it’s easy

- ➔ You can build a package with `distutils` or `setuptools`
- ➔ Put it on the Cheese Shop
- ➔ People can download and install

Make it “packageable” even if it’s only for your use

General best practices

- ➜ Be up-front about dependencies
- ➜ Write for Python 2.3 when possible
- ➜ Pick a release or pick trunk, and document that
- ➜ But if you pick trunk, update frequently

I usually don’t do default templates

Be obsessive about documentation

- ➔ It’s Python: give stuff docstrings
- ➔ If you do, Django will generate documentation for you
- ➔ And users will love you forever

5

Embracing and extending

Don’t touch!

- Good applications are extensible without hacking them up.
- Take advantage of everything 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!

Use the Python (and the Django)

Want to extend a view?

- ➔ If possible, wrap the view with your own code.
- ➔ Doing this repetitively? Just write a decorator.

Want to extend a model?

- ➜ You can relate other models to it.
- ➜ You can write subclasses of it.
- ➜ You can create proxy subclasses (in Django 1.1)

Model inheritance is powerful.
With great power comes great responsibility.

Proxy models

- New in Django 1.1.
- Lets you add methods, managers, etc. (you're extending the Python side, not the DB side).
- Keeps **your** extensions in **your** code.
- Avoids many problems with normal inheritance.

Extending a form

- ➔ Just subclass it.
- ➔ No really, that’s all :)

Other tricks

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

But if you **must** make changes to someone else's code…

Keep changes to a minimum

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

Stay up-to-date

- ➔ You 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.

Make sure your VCS is up to the job of merging from upstream

Be a good citizen

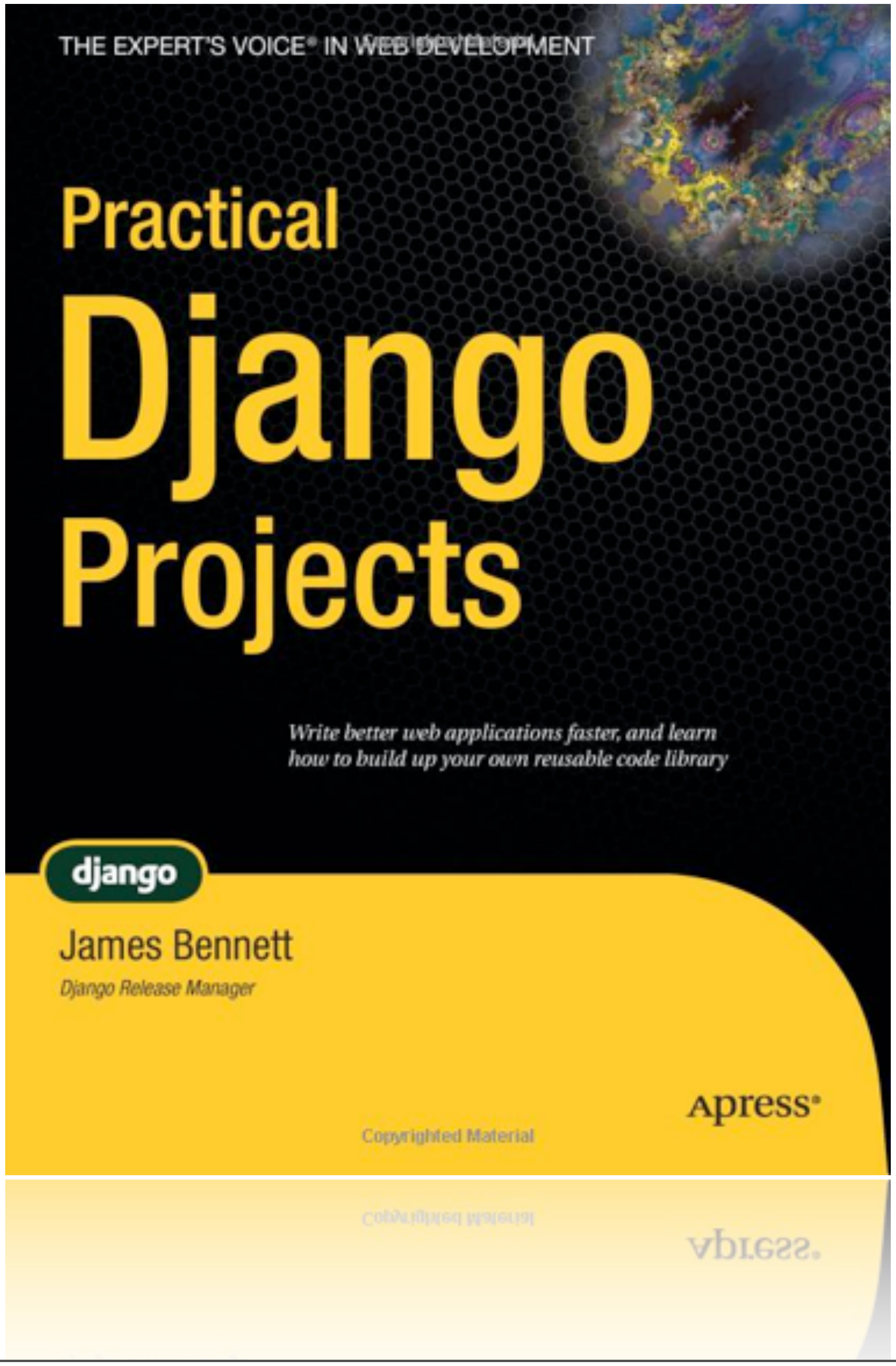
- ➔ If you change someone else’s code, let them know.
- ➔ Maybe they’ll merge your changes in and you won’t have to fork anymore.

What if it’s my own code?

Same principles apply

- ➜ Maybe the original code wasn’t sufficient. Or maybe you just need a new application.
- ➜ Be just as careful about making changes. If nothing else, this will highlight ways in which your code wasn’t extensible to begin with.

Further reading


THE EXPERT'S VOICE® IN WEB DEVELOPMENT

Practical Django Projects

Write better web applications faster, and learn how to build up your own reusable code library

django

James Bennett

Django Release Manager

Apress®

Copyrighted Material

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.

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

```
from django.test import TestCase
from django.http import HttpRequest
from django.middleware.common import CommonMiddleware
from django.conf import settings

class CommonMiddlewareTest(TestCase):
    def setUp(self):
        self.slash = settings.APPEND_SLASH; self.www = settings.PREPEND_WWW

    def tearDown(self):
        settings.APPEND_SLASH = self.slash; settings.PREPEND_WWW = self.www

    def _get_request(self, path):
        request = HttpRequest()
        request.META = {'SERVER_NAME':'testserver', 'SERVER_PORT':80}
        request.path = request.path_info = "/middleware/%s" % path
        return request

    def test_append_slash_redirect(self):
        settings.APPEND_SLASH = True
        request = self._get_request('slash')
        r = CommonMiddleware().process_request(request)
        self.assertEquals(r.status_code, 301)
        self.assertEquals(r['Location'], 'http://testserver/middleware/slash/')
```

django.test.TestCase

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

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 Template(object):
    """
    Deal with a URI template as a class::

        >>> t = Template("http://example.com/{p}?{-join|&|a,b,c}")
        >>> t.expand(p="foo", a="1")
        'http://example.com/foo?a=1'
        >>> t.expand(p="bar", b="2", c="3")
        'http://example.com/bar?c=3&b=2'
    """

def parse_expansion(expansion):
    """
    Parse an expansion -- the part inside {curlybraces} -- into its component
    parts. Returns a tuple of (operator, argument, variabledict)::

        >>> parse_expansion("-join|&|a,b,c=1")
        ('join', '&', {'a': None, 'c': '1', 'b': None})

        >>> parse_expansion("c=1")
        (None, None, {'c': '1'})
    """

def percent_encode(values):
    """
    Percent-encode a dictionary of values, handling nested lists correctly::

        >>> percent_encode({'company': 'AT&T'})
        {'company': 'AT%26T'}
        >>> percent_encode({'companies': ['Yahoo!', 'AT&T']})
        {'companies': ['Yahoo%21', 'AT%26T']}
    """
```

```
****************************************************
File "uri.py", line 150, in __main__.parse_expansion
Failed example:
    parse_expansion("c=1")
Expected:
    (None, None, {'c': '2'})
Got:
    (None, None, {'c': '1'})
****************************************************
```

Functional tests

- a.k.a "Behavior Driven Development."
- "Blackbox," holistic testing.
- All the hardcore TDD folks look down on functional tests.
- But it keeps 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.

```
def testBasicAddPost(self):
    """
    A smoke test to ensure POST on add_view works.
    """
    post_data = {
        "name": u"Another Section",
        # inline data
        "article_set-TOTAL_FORMS": u"3",
        "article_set-INITIAL_FORMS": u"0",
    }
    response = self.client.post('/admin/admin_views/section/add/', post_data)
    self.failUnlessEqual(response.status_code, 302)

def testCustomAdminSiteLoginTemplate(self):
    self.client.logout()
    request = self.client.get('/test_admin/admin2/')
    self.assertTemplateUsed(request, 'custom_admin/login.html')
    self.assertEquals(request.context['title'], 'Log in')
```

Web browser testing

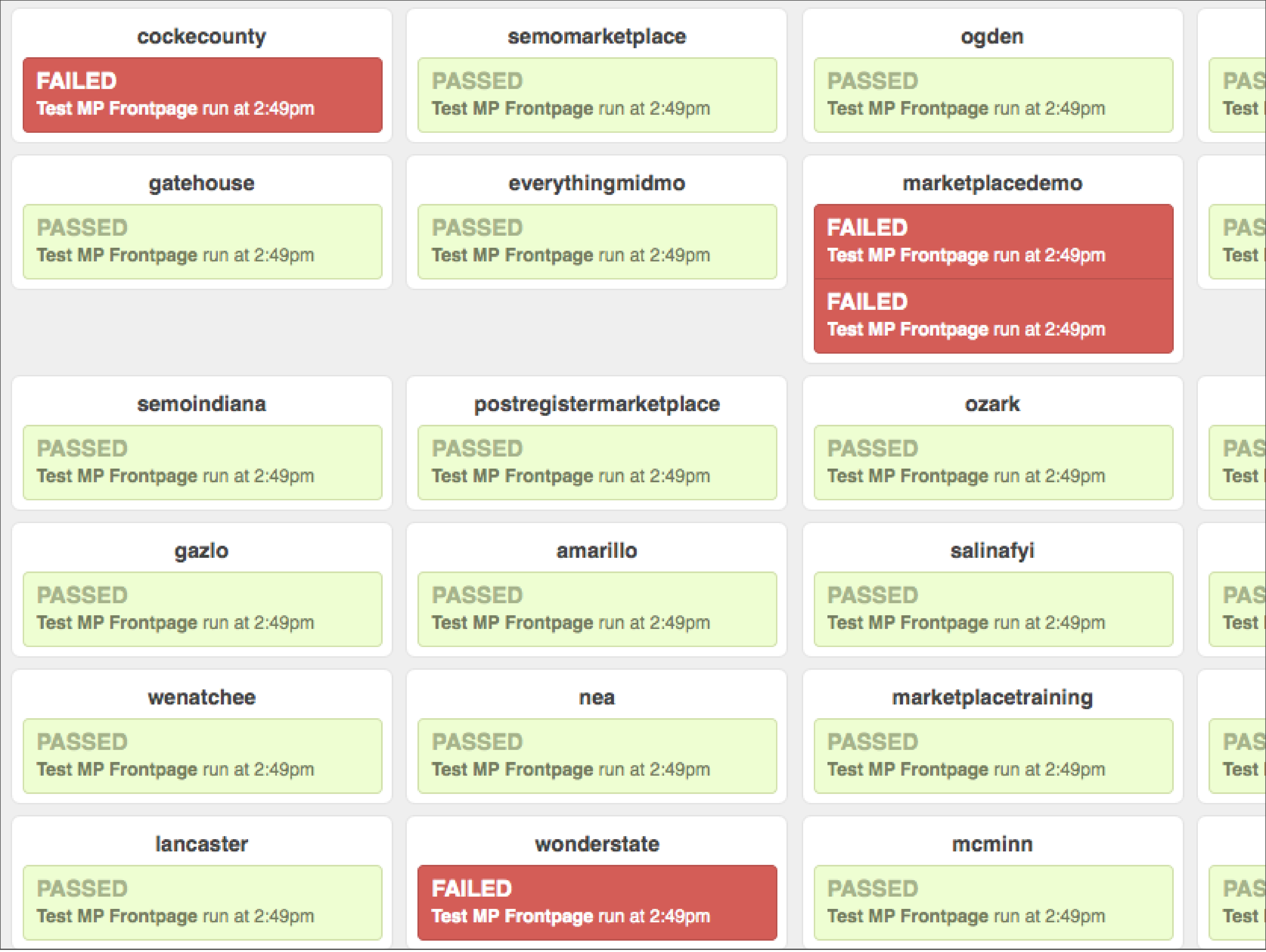
- The ultimate in functional testing for web applications.
- Run test in a web browser.
- Can verify JavaScript, AJAX; even design.
- Test your site across supported browsers.

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

- ➜ Talks here at PyCon!
  http://bit.ly/pycon2009-testing
- ➜ Don't miss the testing tools panel
  (Sunday, 10:30am)
- ➜ Django testing documentation
  http://bit.ly/django-testing
- ➜ Python Testing Tools Taxonomy
  http://bit.ly/py-testing-tools

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
zc.buildout		

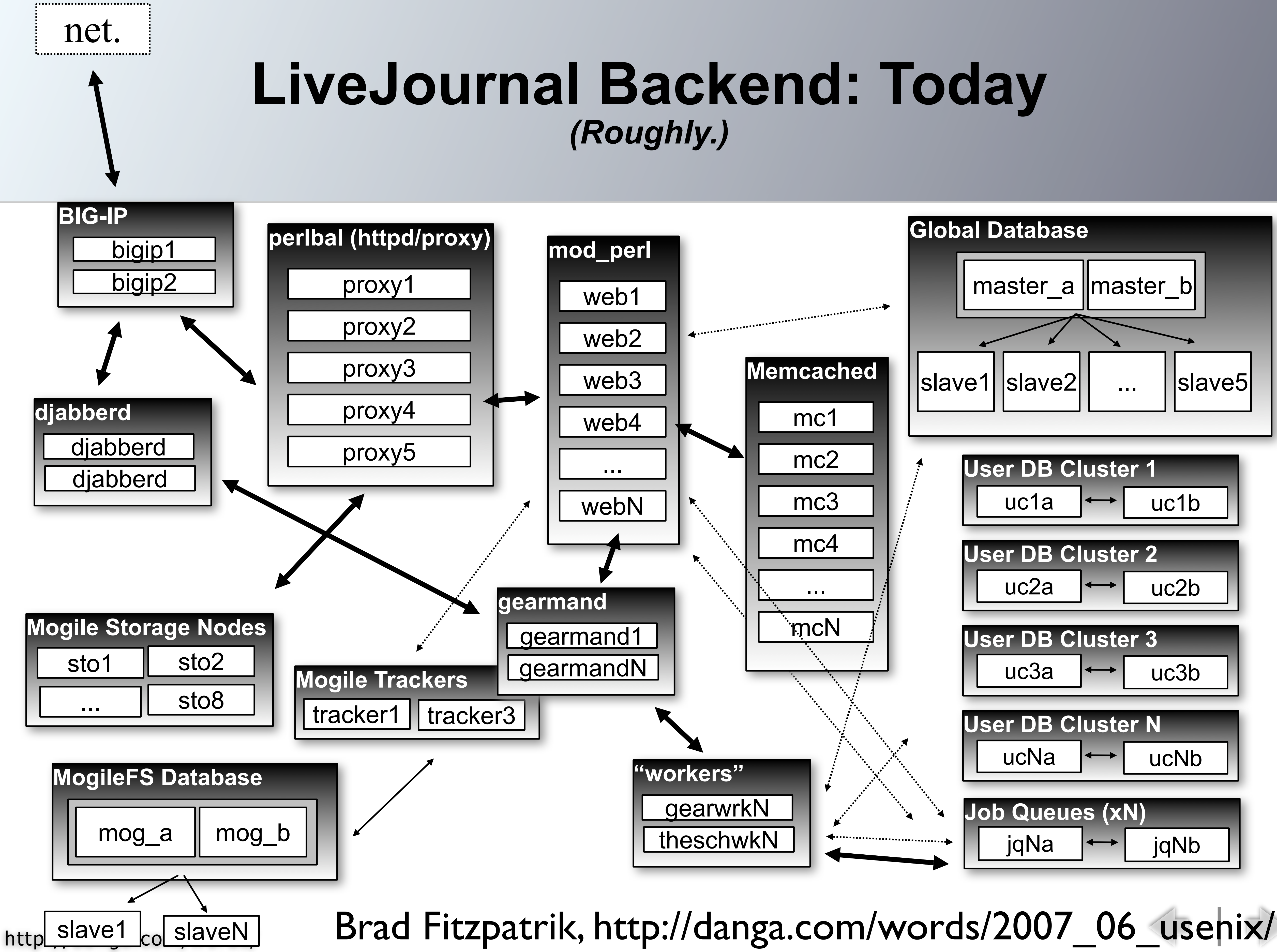
Let the live demo begin

(gulp)

Building your stack

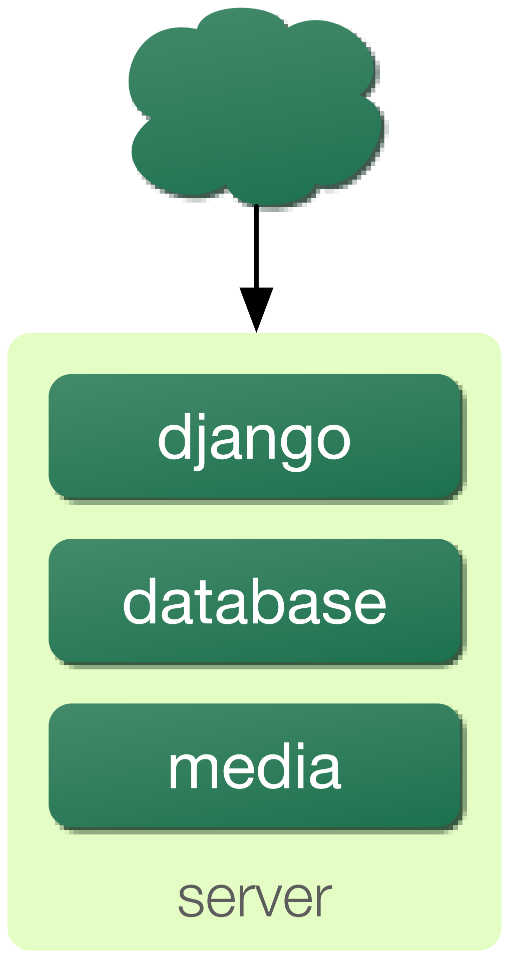
LiveJournal Backend: Today

(Roughly.)



http co

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


django
database
media
server

Application servers

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

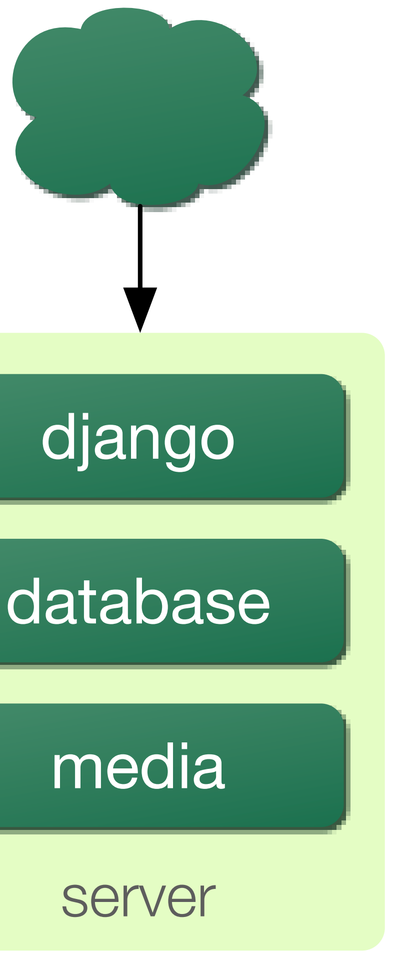
Use mod_wsgi

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

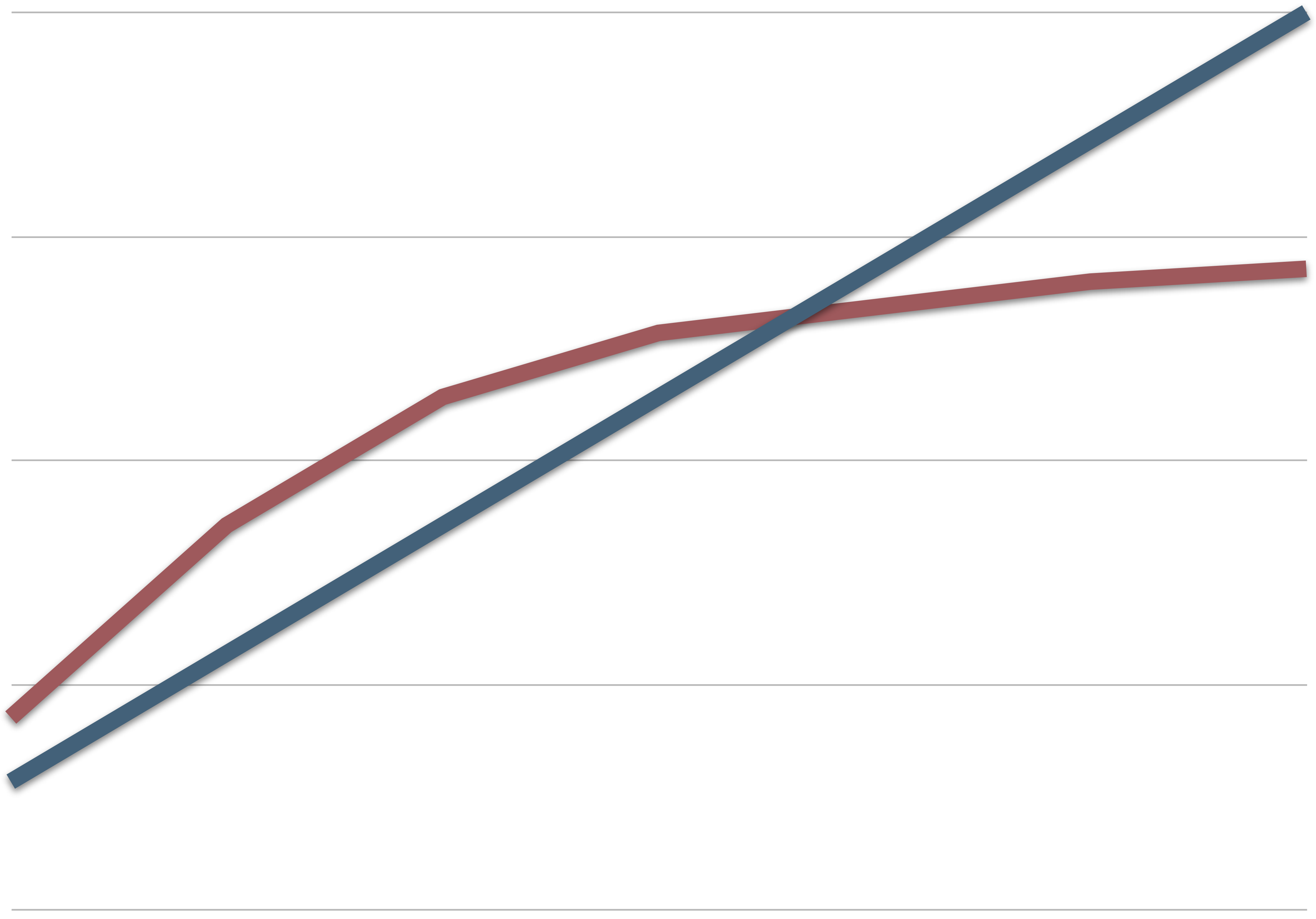
```
import os, sys

# 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()
```

A brief digression regarding the question of scale

Does this scale?



Maybe!



Real-world example

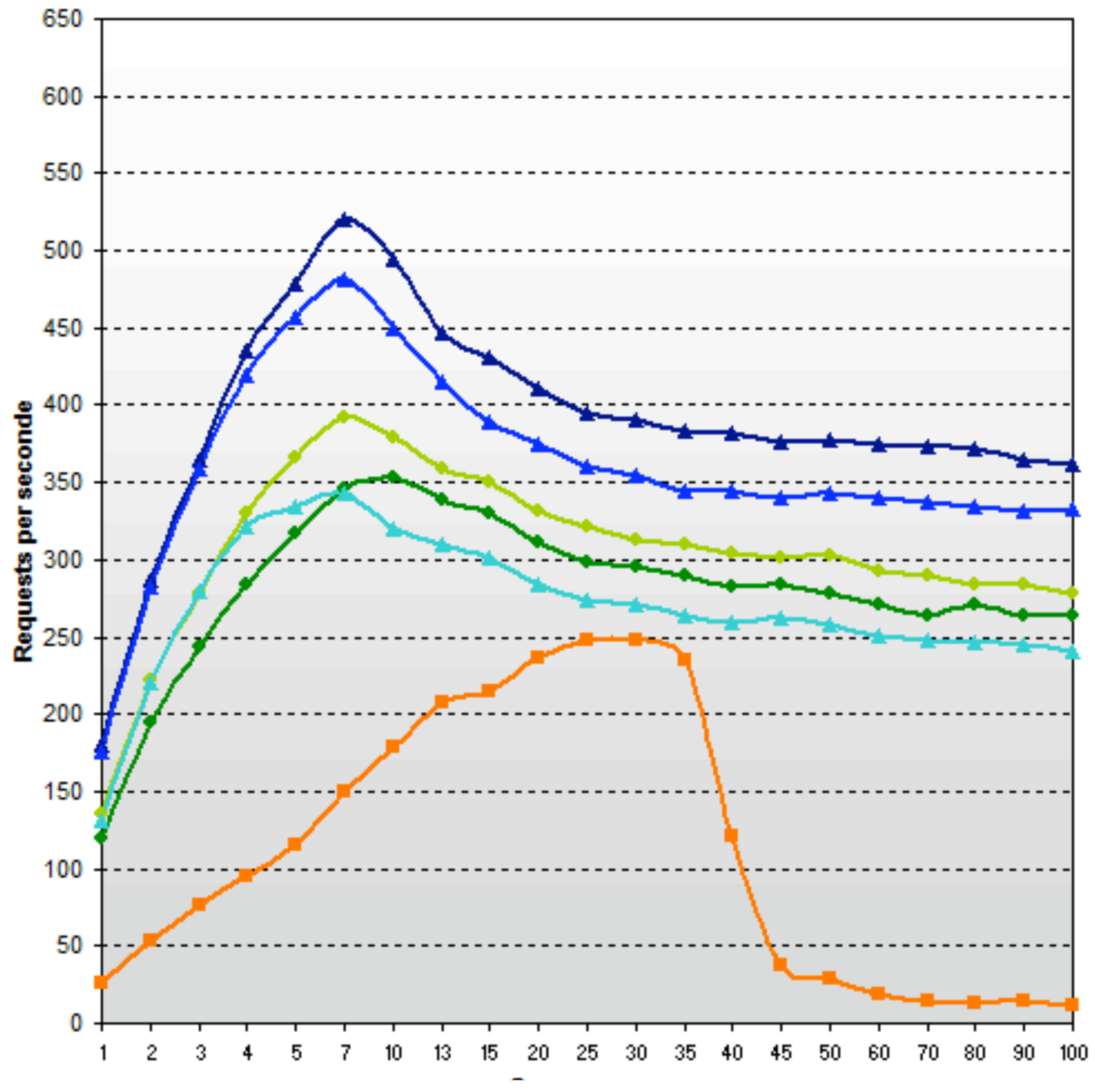
Database A

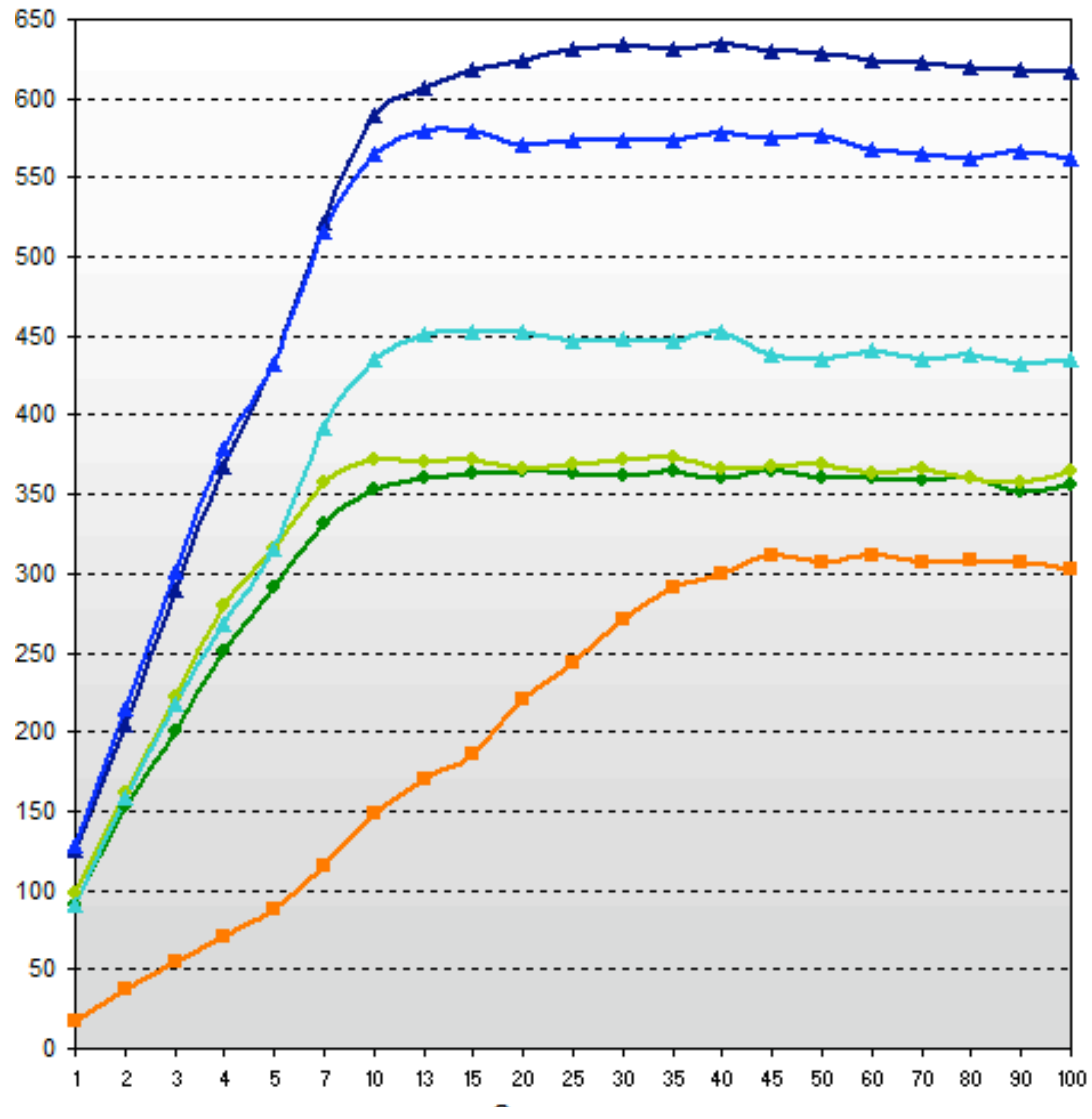
175 req/s

Database B

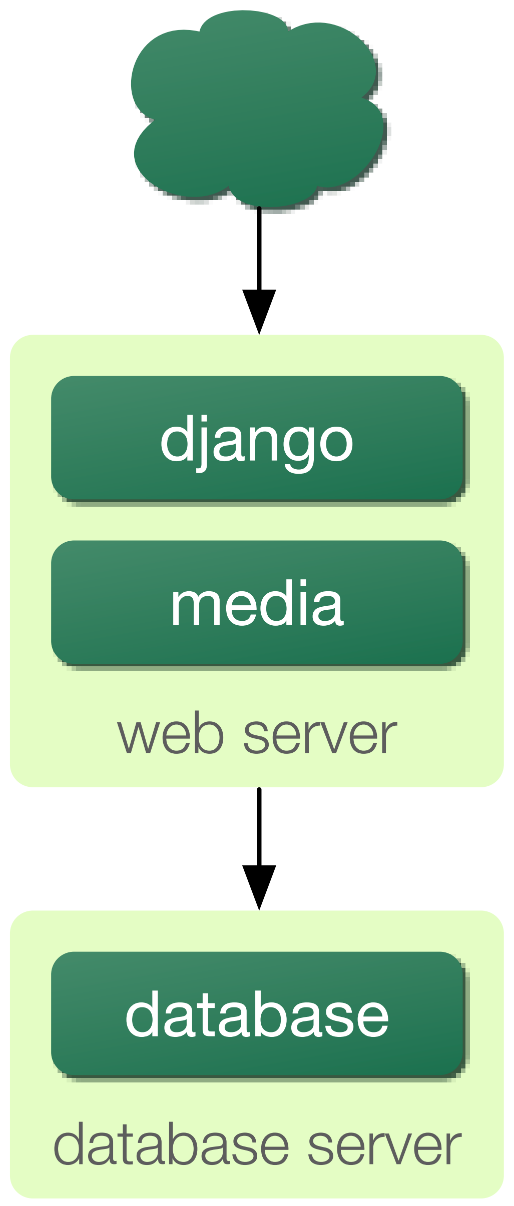
75 req/s

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'
```

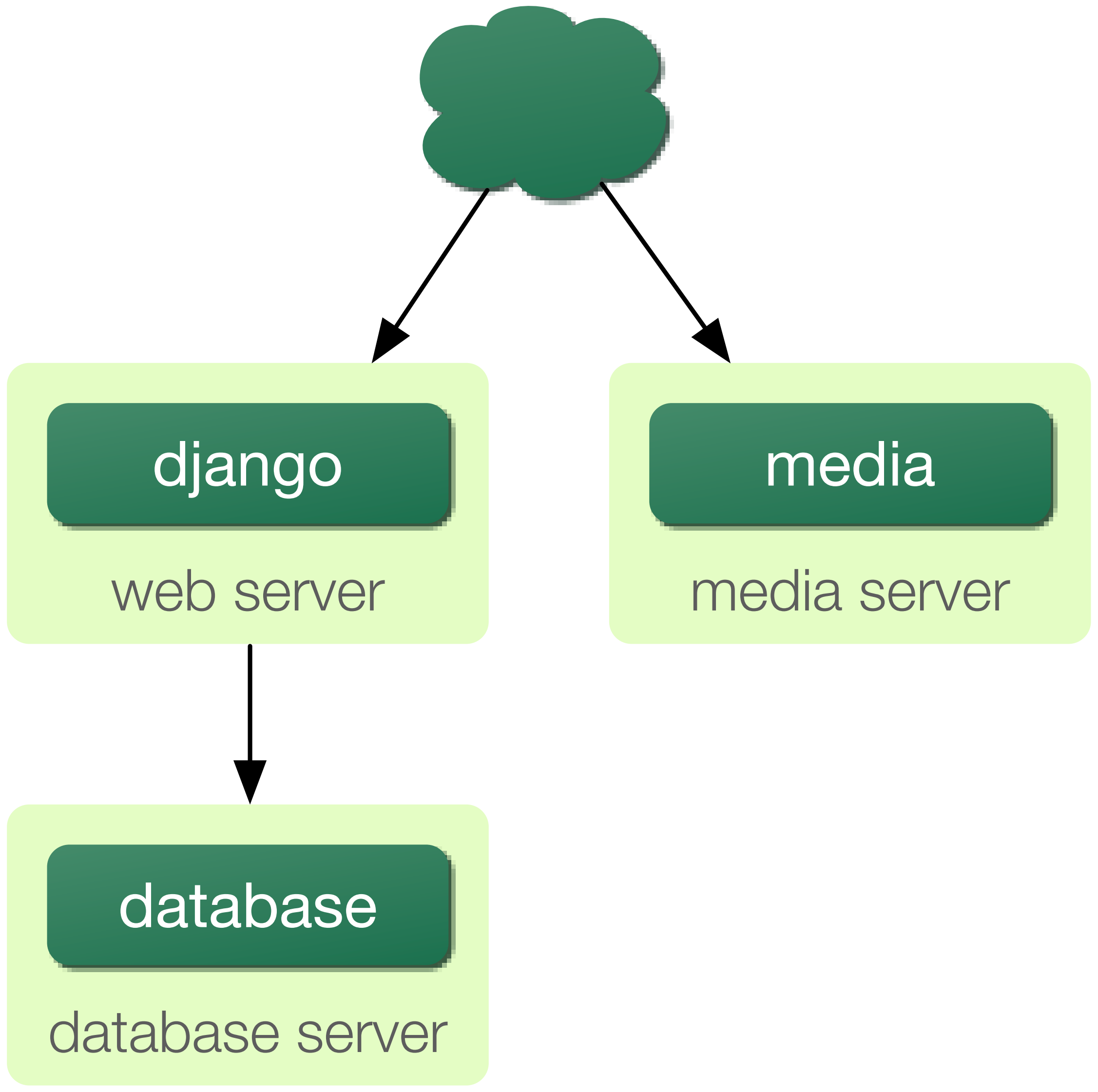
FAIL

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

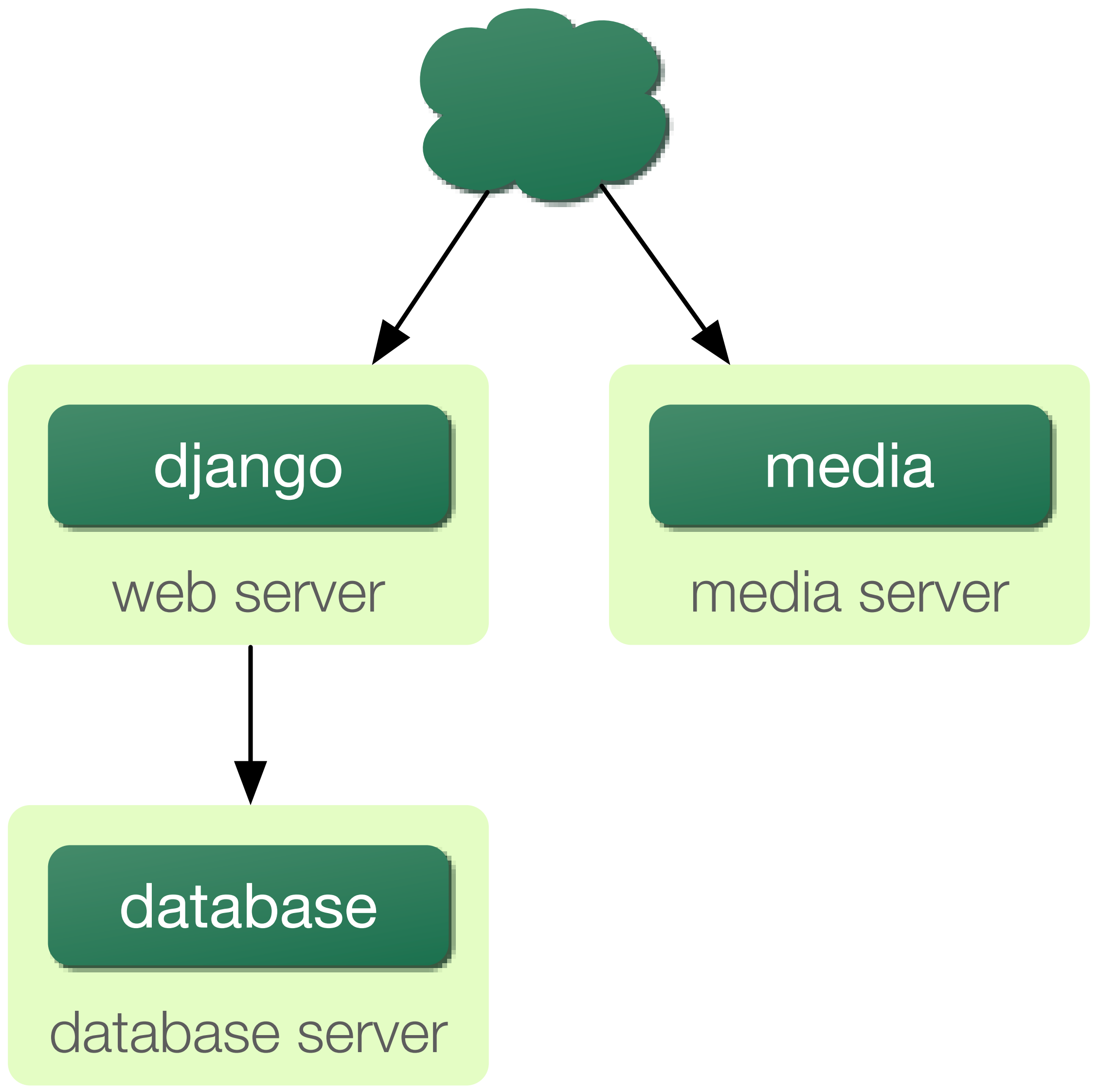
Media server traits

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

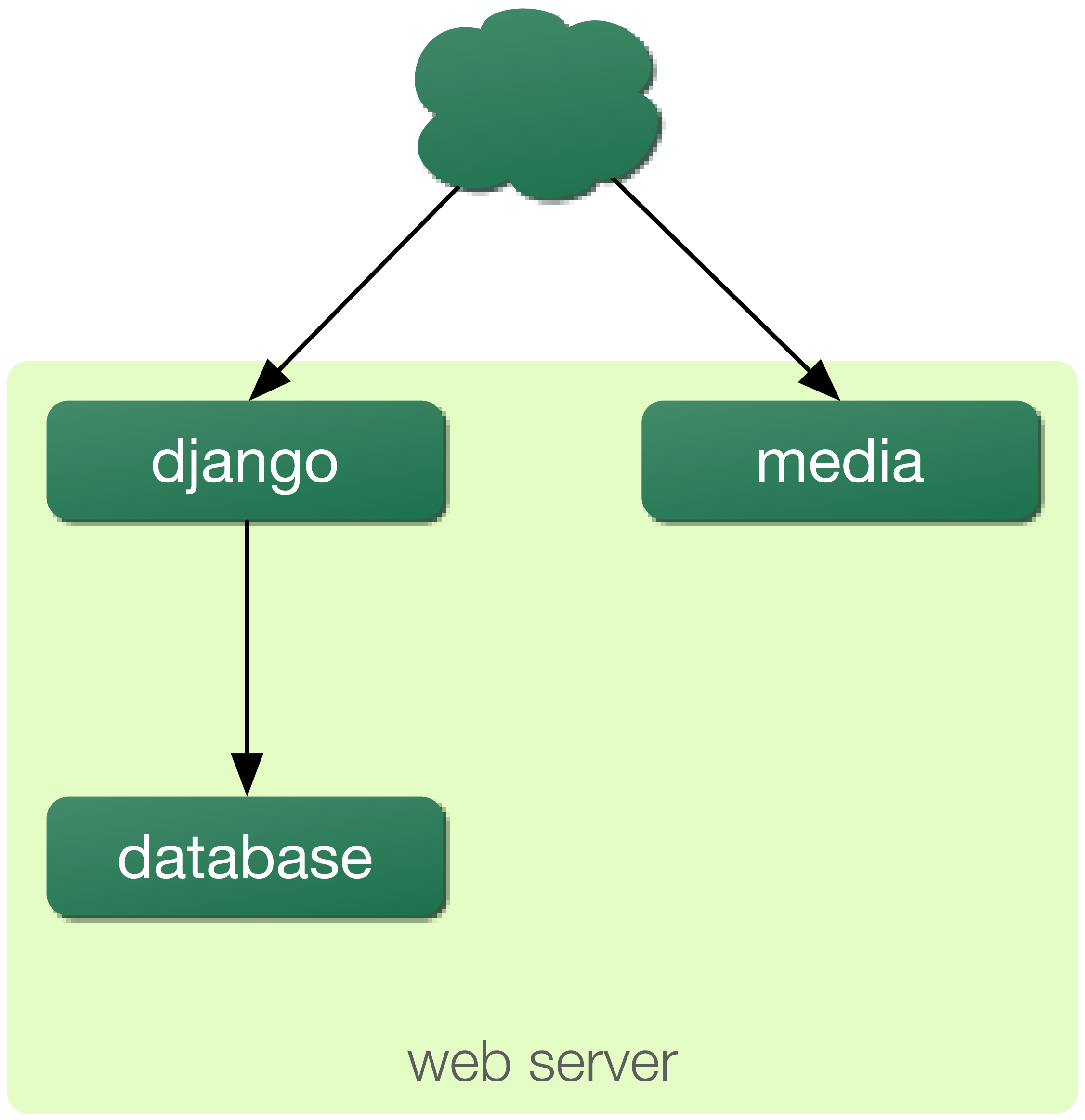
Media servers

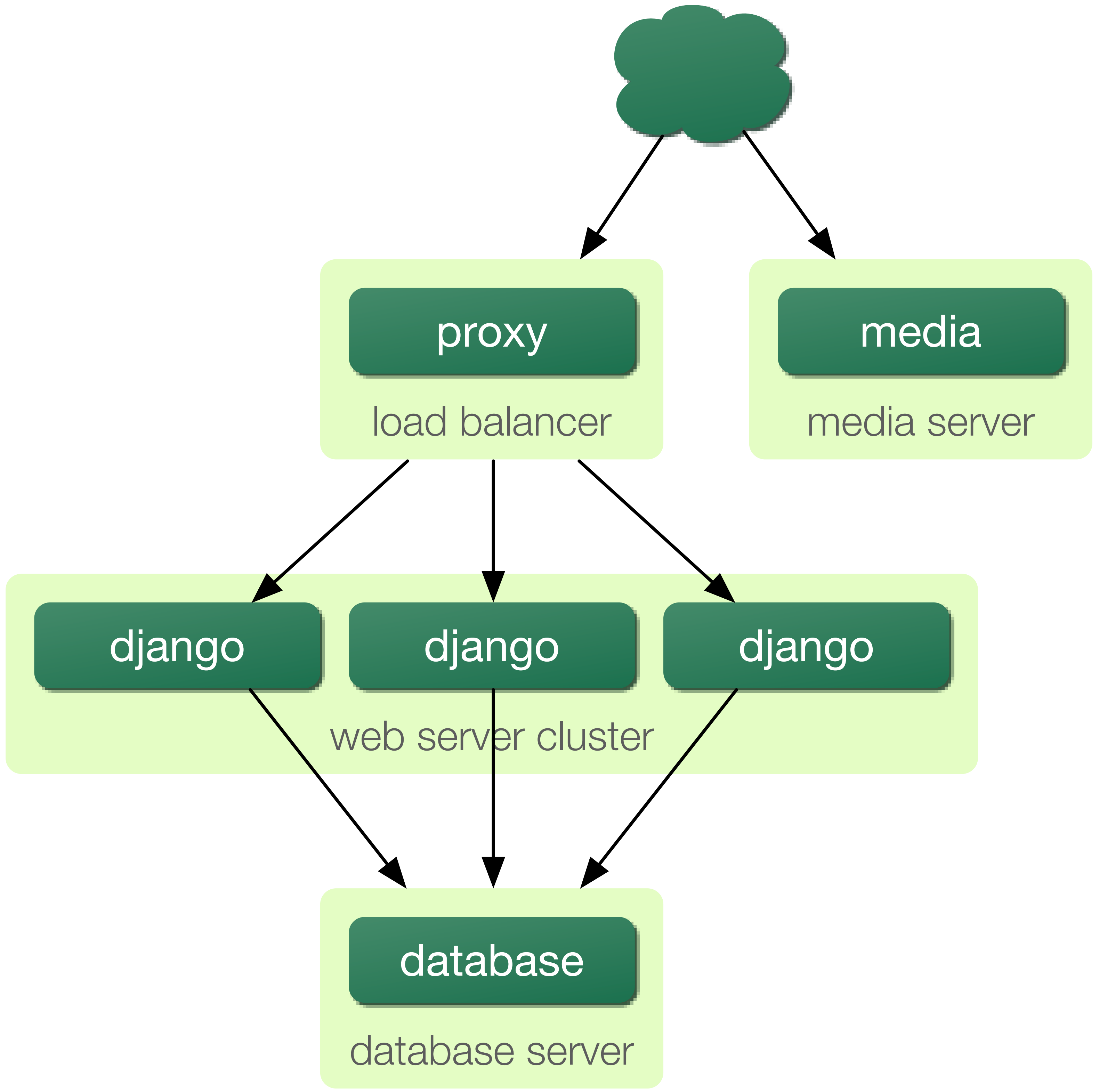
- Apache?
- lighttpd
- nginx

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

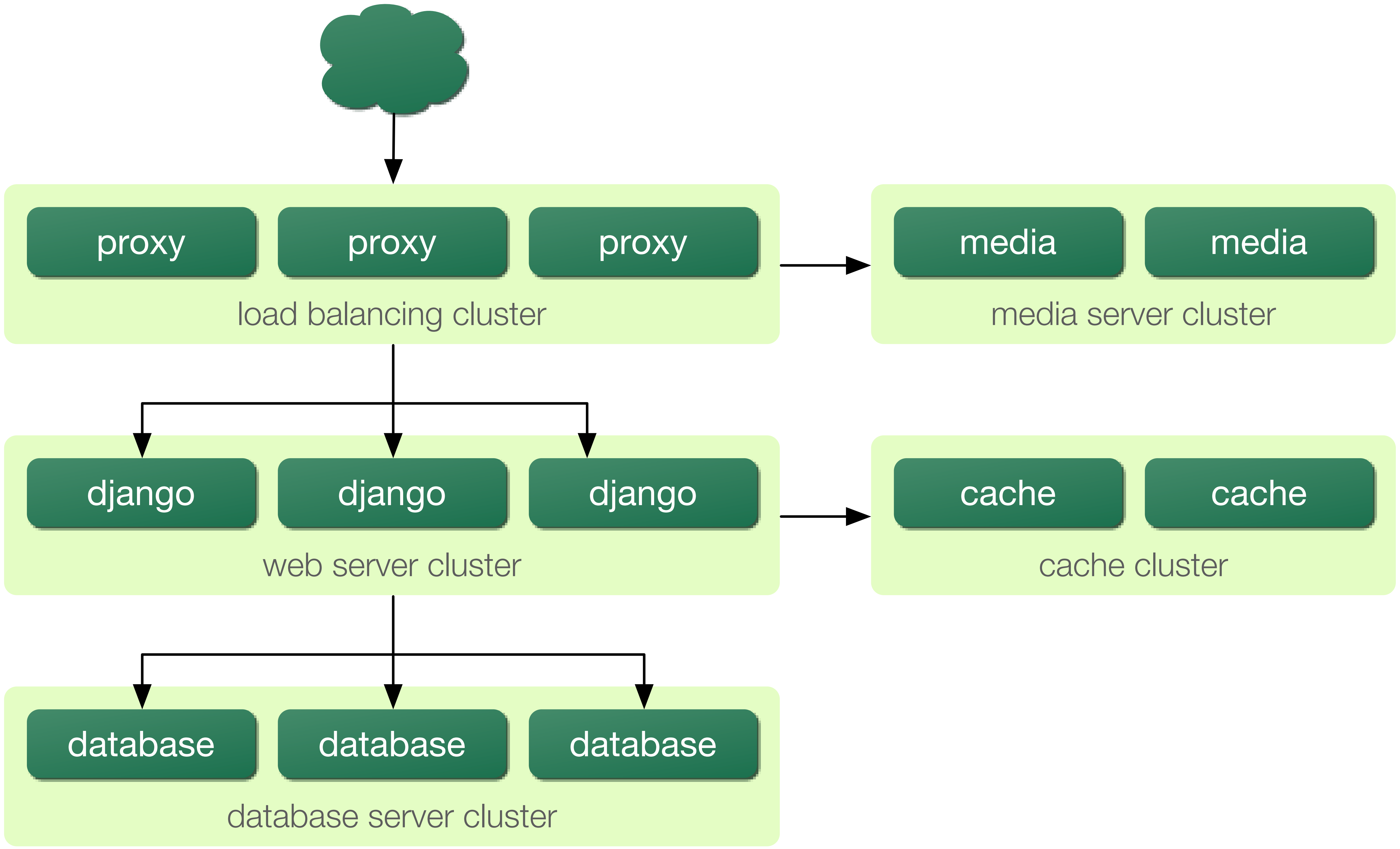
- ➜ Low memory overhead
- ➜ High concurrency
- ➜ Hot fallover
- ➜ Other nifty features...

Load balancers

- ➔ Apache + mod_proxy
- ➔ perlbal
- ➔ nginx

```
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, <u>Building Scalable Web Sites</u> (O'Reilly, 2006)
- ➜ John Allspaw, <u>The Art of Capacity Planning</u> (O'Reilly, 2008)
- ➜ 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 (if the db is down, I don’t need to also hear that the web servers are down.)
- ➔ 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

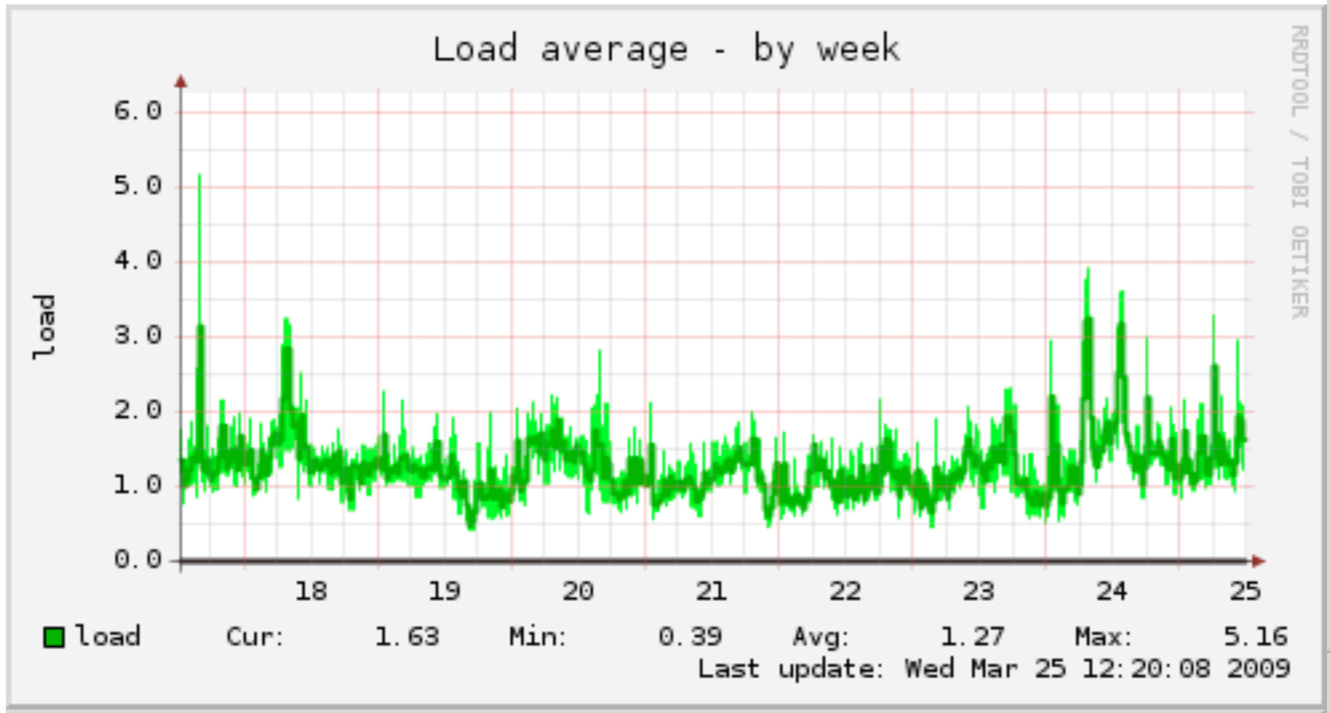
- Internal tools
  - Nagios
  - Monit
  - Zenoss
  - ...
- External monitoring tools

Usage monitoring

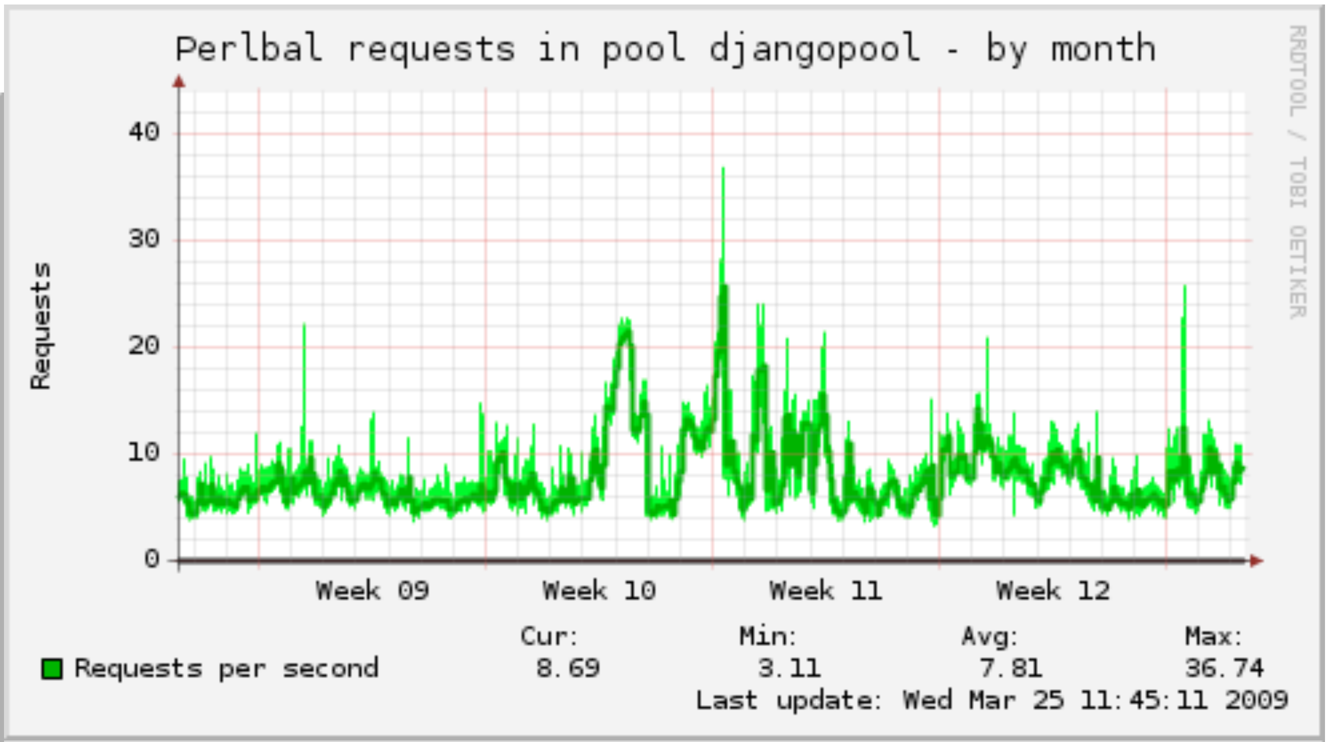
- ➔ Keep track of resource usage over time.
- ➔ Spot and identify trends.
- ➔ Aid in capacity planning and management.
- ➔ Look good in reports to your boss.

Usage monitoring tools

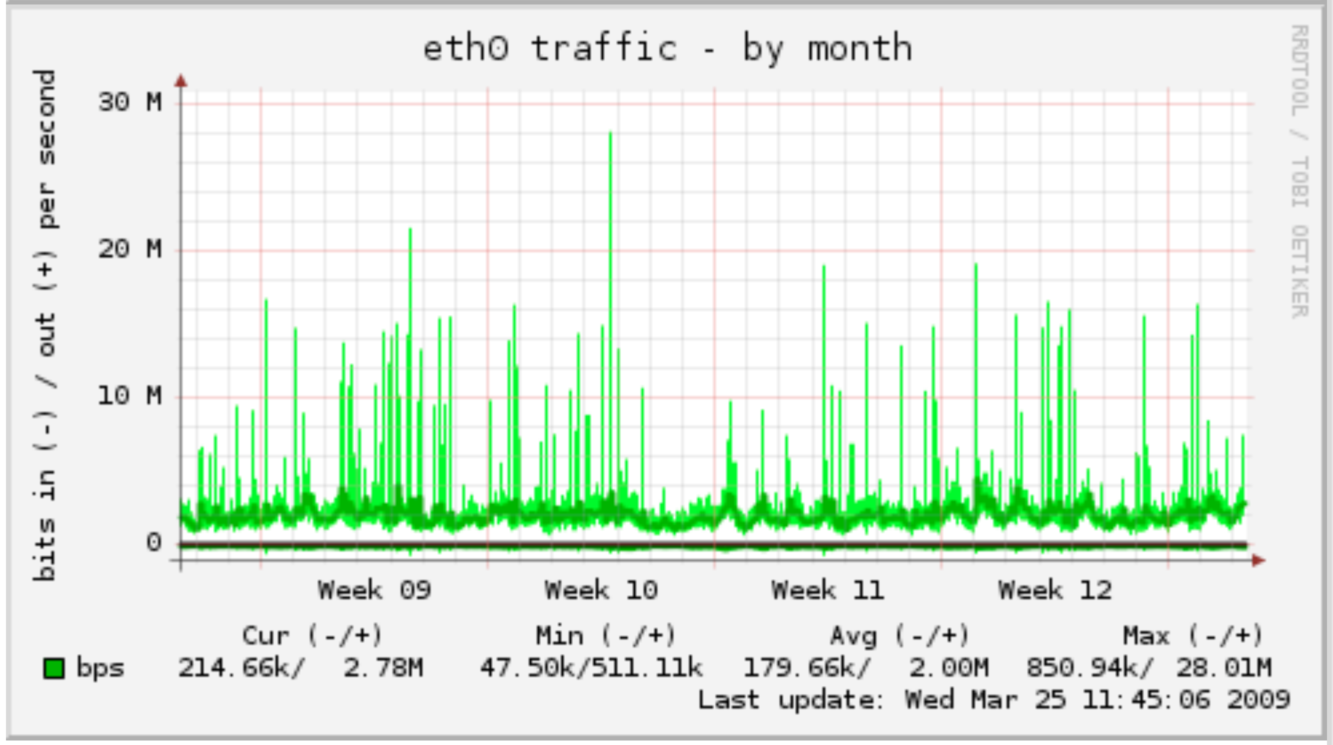
- ➔ RRDTool
- ➔ Munin
- ➔ Cacti
- ➔ Graphite



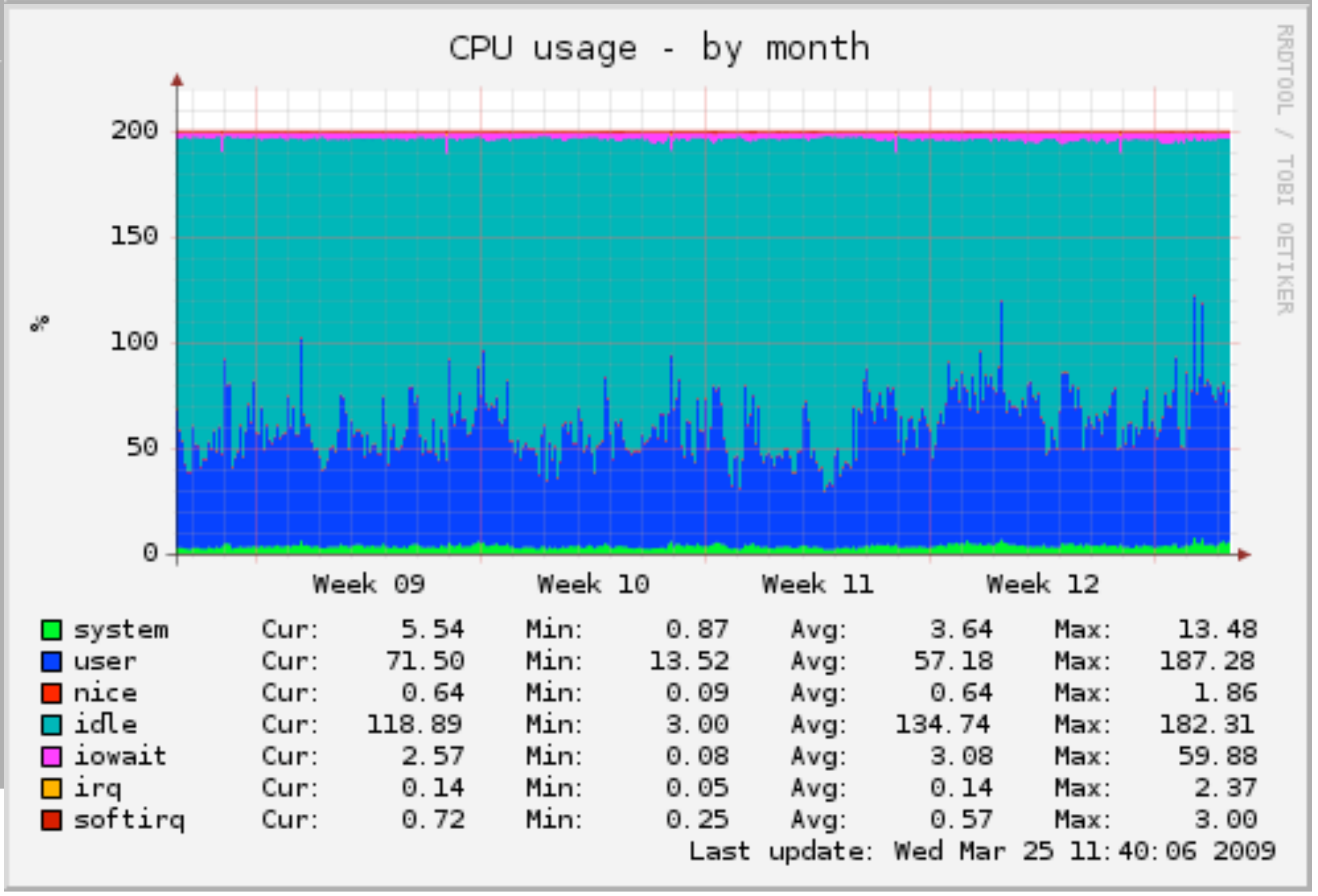
```
🟩 load      Cur:      1.63    Min:      0.39    Avg:      1.27    Max:      5.16
                                        Last update: Wed Mar 25 12:20:08 2009
```



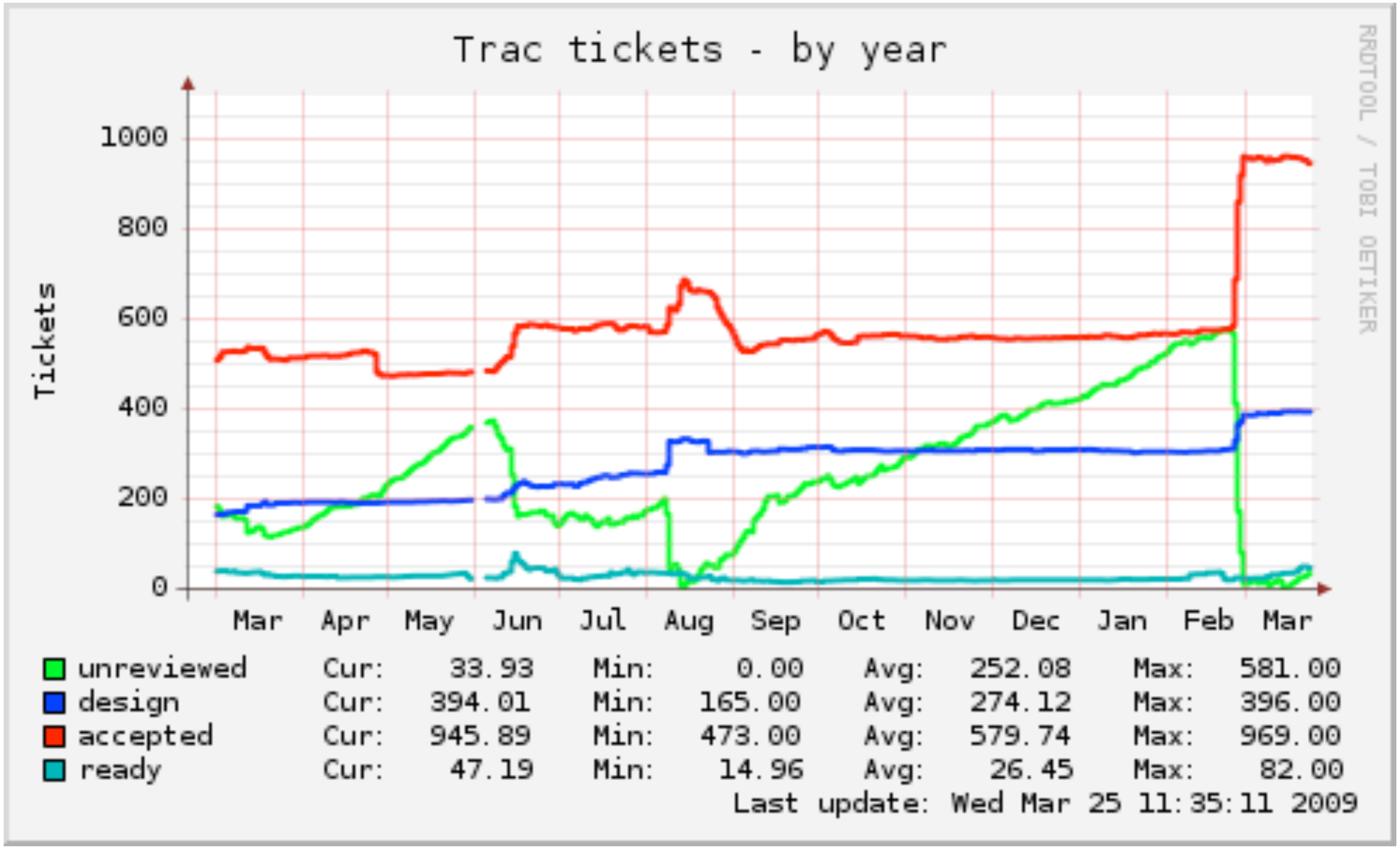
```
                              Cur:          Min:          Avg:          Max:
🟩 Requests per second        8.69          3.11          7.81          36.74
                                        Last update: Wed Mar 25 11:45:11 2009
```



```
          Cur (-/+)          Min (-/+)          Avg (-/+)          Max (-/+)
🟩 bps  214.66k/  2.78M   47.50k/511.11k   179.66k/  2.00M   850.94k/ 28.01M
                                        Last update: Wed Mar 25 11:45:06 2009
```



```
🟩 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
```



```
🟩 unreviewed    Cur:    33.93    Min:      0.00    Avg:   252.08    Max:   581.00
🟦 design        Cur:   394.01    Min:    165.00    Avg:   274.12    Max:   396.00
🟥 accepted      Cur:   945.89    Min:    473.00    Avg:   579.74    Max:   969.00
🟦 ready         Cur:    47.19    Min:     14.96    Avg:    26.45    Max:    82.00
                                     Last update: Wed Mar 25 11:35:11 2009
```

Logging and log analysis

- 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, ...

Performance (and when to care about it)

Ignore performance

- ➜ First, get the application written.
- ➜ Then, make it work.
- ➜ Then get it running on a server.
- ➜ Then, maybe, think about performance.

Code isn’t “fast” or “slow” until it’s been written.

Code isn't "fast" or "slow" until it works.

Code isn’t “fast” or “slow” until it’s actually running on a server.

Optimizing code

- ➜ Most of the time, “bad” code is obvious as soon as you write it. So don’t write it.

Low-hanging fruit

- Look for code doing lots of DB queries -- consider caching, or using select_related()
- Look for complex DB queries, and see if they can be simplified.

The DB is the bottleneck

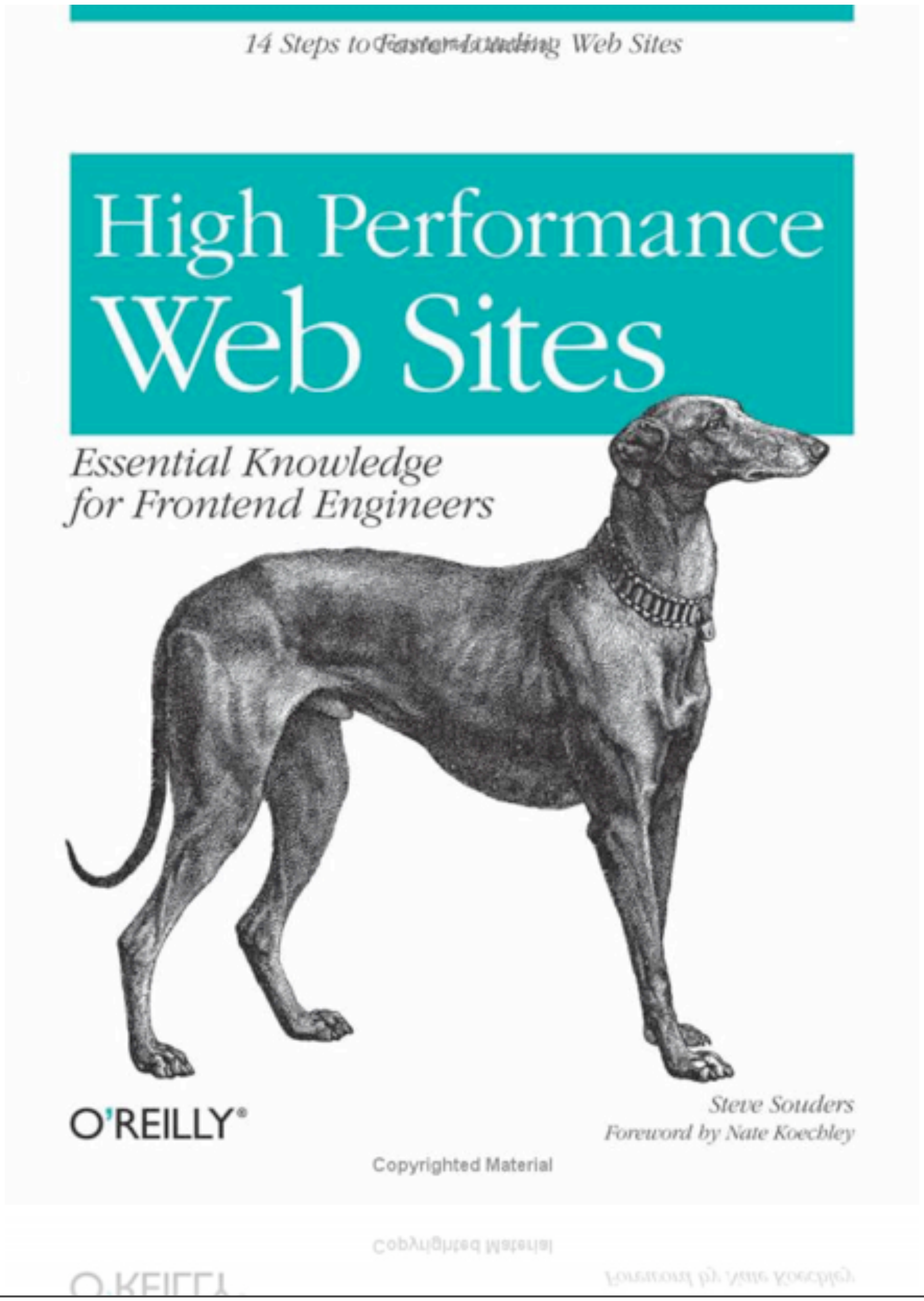
- ➔ And if it’s not the DB, it’s I/O.
- ➔ Everything else is typically negligible.

Find out what “slow” means

- Do testing in the browser.
- Do testing with command-line tools like wget.
- Compare the results, and you may be surprised.

Sometimes, perceived "slowness" is actually on the front end.

Read Steve Souders' book



14 Steps to Faster-Loading Web Sites

High Performance Web Sites

Essential Knowledge for Frontend Engineers

O'REILLY®

Steve Souders

Foreword by Nate Koechley

Copyrighted Material

YSlow

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

What to do on the server side

- ➔ First, try caching.
- ➔ Then try caching some more.

The secret weapon

- Caching turns less hardware into more.
- Caching puts off buying a new DB server.

But caching is a trade-off

Things to consider

- Cache for everybody? Or only for people who aren’t logged in?
- Cache everything? Or only a few complex views?
- Use Django’s cache layer? Or an external caching system?

Not all users are the same

- Most visitors to most sites aren't logged in.
- CACHE_MIDDLEWARE_ANONYMOUS_ONLY

Not all views are the same

- ➜ You probably already know where your nasty DB queries are.
- ➜ cache_page on those particular views.

Site-wide caches

- You can use Django's cache middleware to do this…
- Or you can use a proper caching proxy (e.g., Squid, Varnish).

External caches

- ➜ Work fine with Django, because Django just uses HTTP’s caching headers.
- ➜ Take the entire load off Django -- requests never even hit the application.

When caching doesn’t cut it

Throw money at your DB first

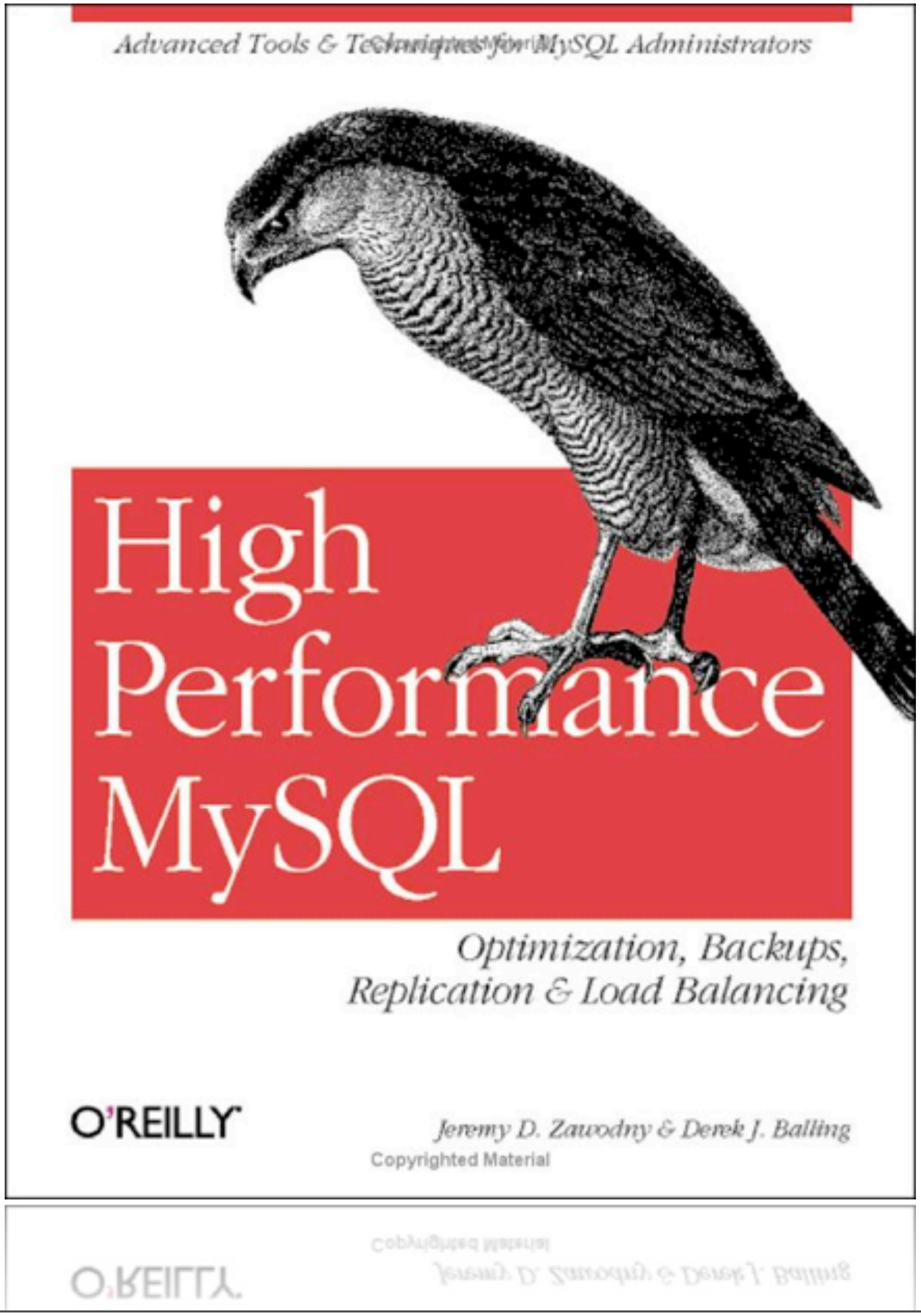
Web server improvements

- ➜ Simple steps first: turn off Keep-Alive, etc.
- ➜ Consider switching to a lighter-weight web server (e.g., nginx) or lighter-weight system (e.g., from mod_python to mod_wsgi).

Database tuning

- ➔ Whole books can be written on DB performance tuning

Using MySQL?


Advanced Tools & Techniques for MySQL Administrators
High Performance MySQL
Optimization, Backups, Replication & Load Balancing
O'REILLY
Jeremy D. Zawodny & Derek J. Balling
Copyrighted Material

Using PostgreSQL?

http://www.revsys.com/writings/postgresql-performance.html

Learn how to diagnose

- ➜ If things are slow, the cause may not be obvious.
- ➜ Even if you think it’s obvious, that may not be the cause.

Build a toolkit

- ➜ Python profilers: profile and cProfile
- ➜ Generic “spy on a process” tools: strace, SystemTap, and dtrace.
- ➜ Django debug toolbar
(http://bit.ly/django-debug-toolbar)

Shameless plug

REVSYS | revolution systems

http://revsys.com/

Fin.

Jacob Kaplan-Moss <jacob@jacobian.org>

James Bennett <james@b-list.org>