REVSYS | revolution systems

Django Deployment Workshop

Jacob Kaplan-Moss
Frank Wiles

OSCON 2010

http://revsys.com/oscon2010

About us

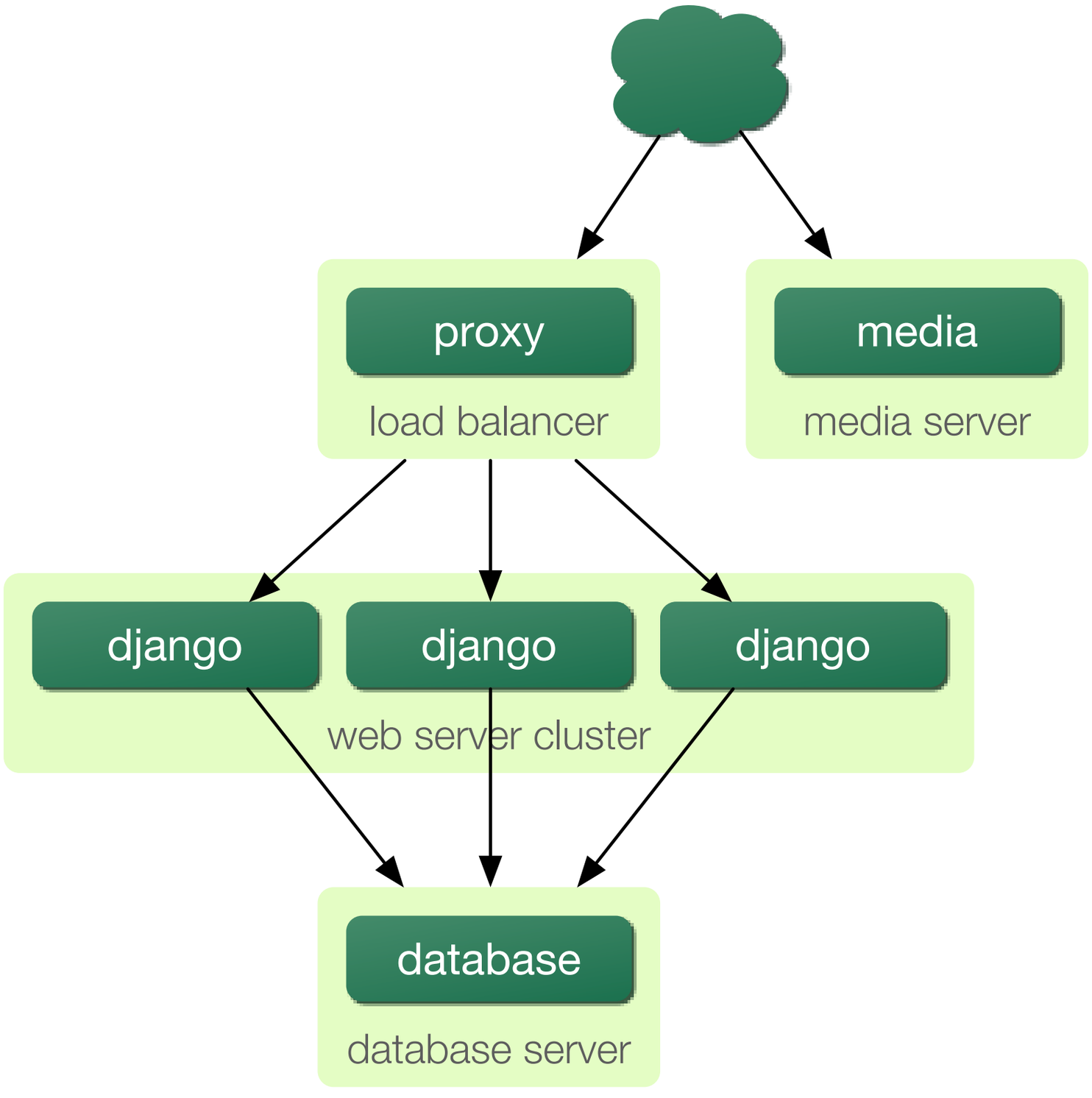
So you've written a Django site…

… now what?

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


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

Deployment steps

1. Bootstrap an app instance.
2. Configure the database.
3. Set up application servers.
4. Automate deployment.
5. Scale up to multiple web servers.
6. Start caching.
7. Tune, tune, tune…

Deployment steps

1. **Bootstrap an app instance.**
2. Configure the database.
3. Set up application servers.
4. Automate deployment.
5. Scale up to multiple web servers.
6. Start caching.
7. Tune, tune, tune…

Demo

Deployment steps

1. Bootstrap an app instance.
2. **Configure the database.**
3. Set up application servers.
4. Automate deployment.
5. Scale up to multiple web servers.
6. Start caching.
7. Tune, tune, tune…

Database servers

- SQLite
- MySQL
- PostgreSQL
- Oracle
- …

Why PostgreSQL?

Demo

Deployment steps

1. Bootstrap an app instance.
2. Configure the database.
3. **Set up application servers.**
4. Automate deployment.
5. Scale up to multiple web servers.
6. Start caching.
7. Tune, tune, tune…

App servers should…

- Have proven reliability.
- Be highly stable.
- Have predictable resource consumption.
- Speak WSGI.

Application servers

- Old school: Apache + mod_python
- New school: Apache + mod_wsgi
- Cutting edge: uWSGI, Gunicorn
- Avoid: FastCGI, SCGI, AJP, …

Recommendation:

Apache + mod_wsgi

Demo

Deployment steps

1. Bootstrap an app instance.
2. Configure the database.
3. Set up application servers.
4. **Automate deployment.**
5. Scale up to multiple web servers.
6. Start caching.
7. Tune, tune, tune…

Deployment should...

- Automatically manage dependencies.
- Be isolated.
- Be automated.
- 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		

Dependency 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 dependency; 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: chronically 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/

Automation

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: few high-level commands; not yet “1.0”.

Configuration management

- CFEngine, Puppet, Chef, …
- Will handle a *lot* more than code deployment!
- Almost a necessity past N = 20 or so.

Recommendations

- Pip, Virtualenv, and Fabric
- Buildout and Fabric.
- Buildout, Fabric, Puppet.
- Utility computing, Fabric, Chef.

Demo

Deployment steps

1. Bootstrap an app instance.
2. Configure the database.
3. Set up application servers.
4. Automate deployment.
5. **Scale up to multiple web servers.**
6. Start caching.
7. Tune, tune, tune…

Deployment steps

5. **Scale up to multiple web servers.**
   A. N > 1.
   B. Load balancers.
   C. Database connection middleware

Deployment steps

5. **Scale up to multiple web servers.**

A. N > 1.

B. Load balancers.

C. Database connection middleware

Why multiple servers?

- Eliminate resource contention.
- Easier to optimize for scarcity.
- 1 → 2 is much harder than 2 → N

“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?

Dealing with media

Demo

Deployment steps

5. **Scale up to multiple web servers.**

A. N > 1.

B. Load balancers.

C. Database connection middleware

Why load balancers?

Load balancer traits

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

Load balancers

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

Recommendation:

Nginx

Demo

Deployment steps

5. **Scale up to multiple web servers.**
   A. N > 1.
   B. Load balancers.
   C. Database connection middleware

Connection middleware?

```
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, …
- `DATABASE_HOST = '127.0.0.1'`

Connection middleware

- PostgreSQL: pgpool (I, II), pgbouncer.
- MySQL: MySQL Proxy.
- Database-agnostic: sqlrelay.
- Oracle: ?

pgpool vs. pgbouncer

Demo

Deployment steps

1. Bootstrap an app instance.
2. Configure the database.
3. Set up application servers.
4. Automate deployment.
5. Scale up to multiple web servers.
6. **Start caching.**
7. Tune, tune, tune…

“Performance is mostly an exercise in caching and cache invalidation.”

XXX or some such quote - three hard problems, maybe?

Caching layers

- Low-level (Python API)
- Template fragment caching
- Per-view cache
- Whole site cache
- Upstream caches
- CDNs

Cache backends

Production

- Use memcached.
- Really.

Other cache backends

- filesystem
- database
- local memory

Demo

Deployment steps

1. Bootstrap an app instance.
2. Configure the database.
3. Set up application servers.
4. Automate deployment.
5. Scale up to multiple web servers.
6. Start caching.
7. **Tune, tune, tune…**

What's next?

- Database redundancy / replication
- Monitoring (availability; capacity planning)
- Searching, queuing, and locking (oh my!)
- NoSQL.
- Scaling up and out.

Thank you!

Us: Jacob Kaplan-Moss

Frank Wiles

This talk: http://revsys.com/oscon2010

Web: http://revsys.com/

Twitter: @revsys