REVSYS | revolution systems

Django Deployment Workshop

Jacob Kaplan-Moss

http://bit.ly/ddw-slides
http://bit.ly/ddw-code

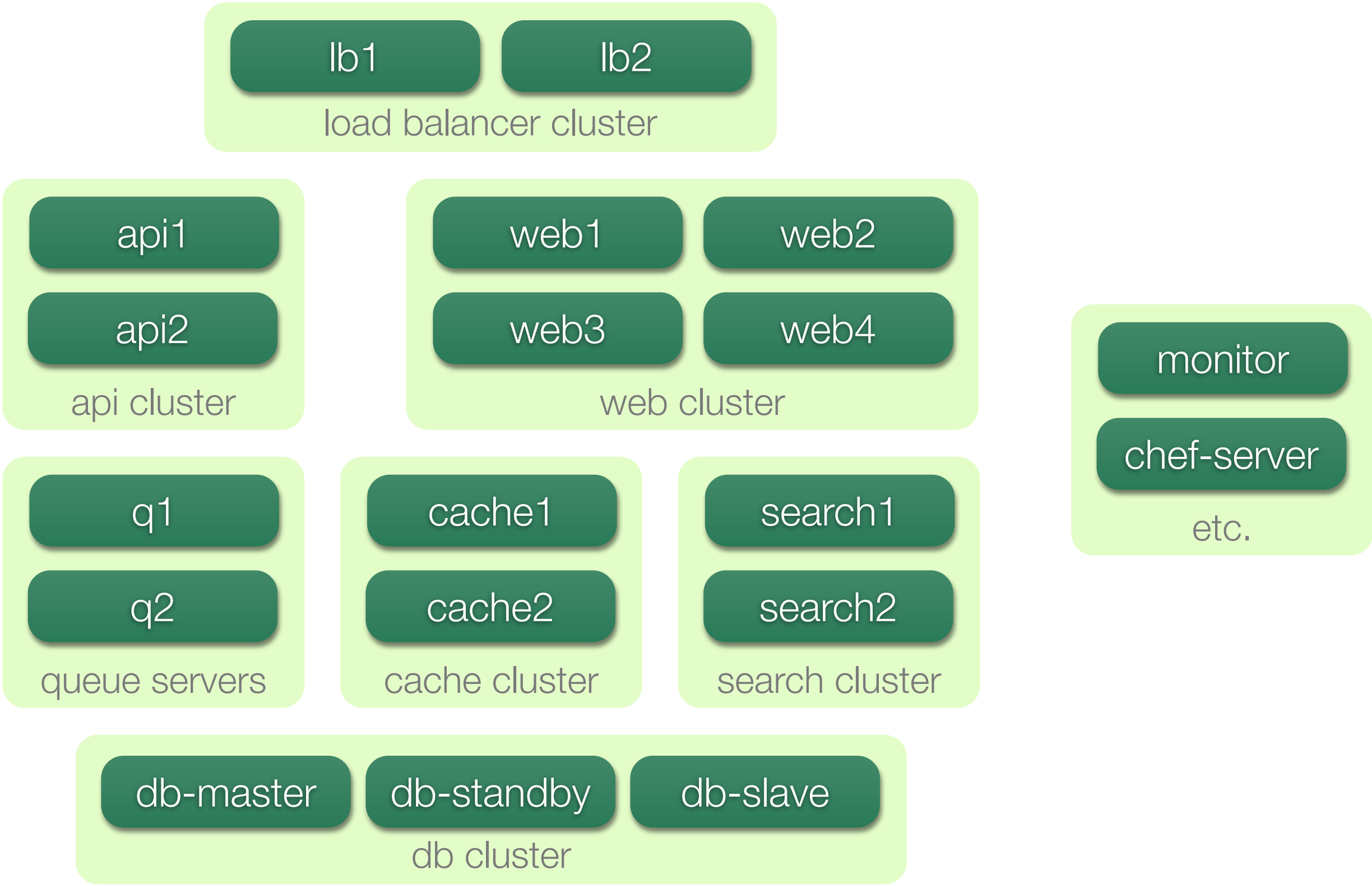
So you've written a Django site...

… now what?

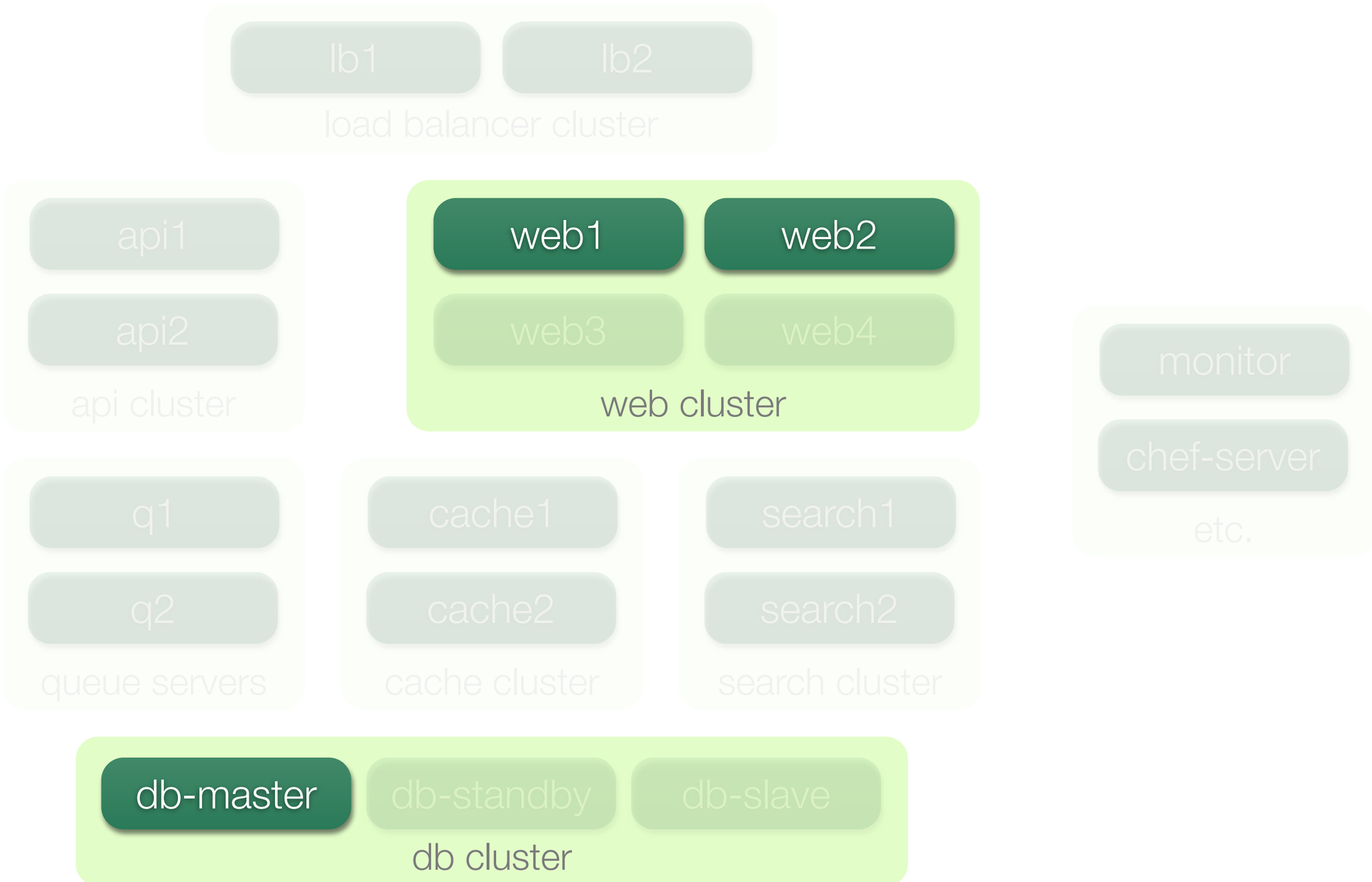
- API Metering
- Backups & Snapshots
- Counters
- Cloud/Cluster Management Tools
  - Instrumentation/Monitoring
  - Failover
  - Node addition/removal and hashing
  - Autoscaling for cloud resources
- CSRF/XSS Protection
- Data Retention/Archival
- Deployment Tools
  - Multiple Devs, Staging, Prod
  - 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

A real-world stack



What we're building



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. 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. Tune, tune, tune…

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/

Suggestions:

pip + virtualenv

Demo

Bootstrapping the application.

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. Tune, tune, tune…

Database servers

- SQLite
- MySQL
- PostgreSQL
- Oracle
- …

Why PostgreSQL?

Demo

Setting up PostgreSQL.

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. Tune, tune, tune…

App servers should…

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

Application servers

- Outdated: Apache + mod_python
- The new standard: Apache + mod_wsgi
- Cutting edge: uWSGI, Gunicorn, CherryPy…
- Avoid: FastCGI, SCGI, AJP.

Recommendations:

- **Most cases: Apache + mod_wsgi.**

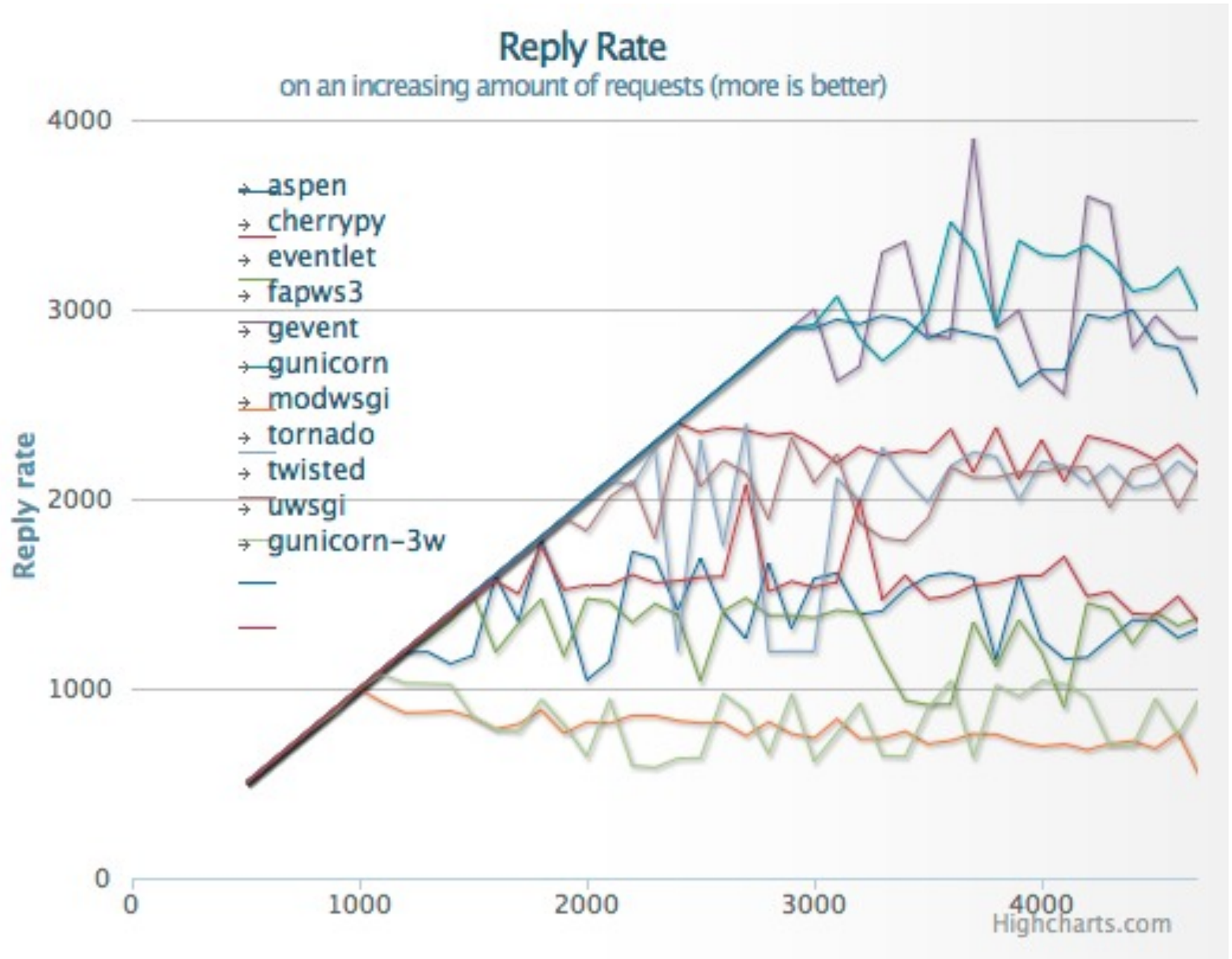
  “A rule of thumb that has worked well for me is that if I'm excited to play around with something, it probably doesn't belong in production.” — Maciej Ceglowski

  http://pinboard.in/blog/63/

- **Easier setup: Gunicorn.**

  But beware: scaling considerations are still an unknown.

Performance?



http://nichol.as/benchmark-of-python-web-servers

Server performance doesn't matter unless your app is this simple.

```
def application(environ, start_response):
    status = '200 OK'
    output = 'Pong!'
    response_headers = [('Content-type', 'text/plain'),
                        ('Content-Length', str(len(output)))]
    start_response(status, response_headers)
    return [output]
```

Demo

Playing with Apache, mod_wsgi, and Gunicorn

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. 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		
zc.buildout		

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”~~.

Demo

A first fabfile.

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. 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

```python
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

Rolling out a second web server.

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: ?

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. **Tune, tune, tune…**

What's next?

- Caching.
- Configuration management.
- Database redundancy / replication.
- Monitoring (availability; capacity planning).
- Searching, queuing.
- Scaling up and out.

Our toolkit:

- **Caching: memcached**
  http://memcached.org/, http://django.me/cache
- **Config. management: Chef, Puppet.**
  http://www.opscode.com/chef/, http://www.puppetlabs.com/
- **DB replication: PostgresSQL hot standby.**
  http://wiki.postgresql.org/wiki/Hot_Standby
- **Monitoring: Nagios, Munin.**
  http://www.nagios.org/, http://munin-monitoring.org/
- **Searching: Haystack, Solr.**
  http://haystacksearch.org/, http://lucene.apache.org/solr/
- **Queuing: Celery, Redis.**
  http://celeryproject.org/, http://redis.io/

Thank you!

Us: Jacob Kaplan-Moss

Frank Wiles

Web: http://revsys.com/

Twitter: @revsys

Bonus:

Configuration management

The basics

- Automate your entire infrastructure.
- No more `ssh server; vi /etc/hosts`!
- Smart updates, self-healing, etc.
- Imperative rather than declarative.
- Almost a necessity past N ≅ 20 or so, but highly recommended even for N = 1.

Options

- **CFEngine**

  The original, still somewhat popular, but fairly crusty.

- **Bcfg2**

  Python - yay! XML - boo!

- **Puppet**

  The first fairly popular modern tool. Probably the most popular among LAMPish stacks.

- **Chef**

  The new upstart, gaining lots of steam.

Our suggestion:

Chef or Puppet

Demo: Chef