Chapter 21: Deploying Django

2

Throughout this book we've mentioned a number of goals that drive the development of Django. Ease of use, friendliness to new programmers, abstraction of repetitive tasks — these all drive Django's developers.

1

However, since its inception as an internal, closed-source project, there's always been another incredibly important goal: Django should be easy to deploy, and should make serving large amounts of traffic possible with limited resources.

1

The motivations for this goal are apparent when you look at Django's background: a small, family-owned newspaper in Kansas can hardly afford top-of-the-line server hardware, so Django's original developers were concerned with squeezing the best possible performance out of limited resources. Indeed, for years Django's developers acted as their own systems administrators — there simply wasn't enough hardware to *need* dedicated admins — even as their sites handled tens of millions of hits a day.

3

As Django became an open-source project, this focus on performance and ease of deployment became important for a different reason: hobbyist developers. Individuals who want to use Django are please to find out that they can host a small- to medium-traffic site for as little as $10 a month.

1

But being able to scale down is only half the battle. Django also needs to be capable of scaling *up* to meet the needs of large companies and corporations. Here, Django adopts a philosophy common among LAMP-like web stacks often called "shared nothing."

7

LAMP?

The acronym LAMP was originally coined to describe a populate set of open-source software used to drive many web sites:

- Linux (the operating system)
- Apache (web server)
- MySQL (database)
- PHP (programming language)

Over time, though, the acronym has come to refer more to the philosphy of these types of open-source software stacks than to any one particular stack. So while Django uses Python and is database-agnostic, the philosophies proven by the LAMP stack permeate Django's deployment mentality.

There have been a few (mostly humourous) attempts at coining a similar acronym to describe Django's technology stack; your authors are fond of **PAID** (PostgreSQL, Apache, Internet and Django) or **LAPD** (Linux, Apache, PostgreSQL, and Django).

"Shared nothing"

At its core, the philosophy of "shared nothing" is really just the application of loose coupling to the entire software stack. This architecture arose in direct response to what was at the time the prevailing architecture: a monolithic web application server that encapsulates the language, database, web server — even parts of the operating system — into a single process (e.g. Java).

5

When it comes time to scale, this can be a major problem; it's nearly impossible to split the work of a monolithic process across many different physical machines, so monolithic applications require enormously powerful servers. These servers, of course, cost tens or even hundreds of thousands of dollars, putting large-scale web sites out of the reach of cash-hungry individuals and small companies.

What the LAMP community noticed, however, was that if you broke each piece of the web stack up into individual components, you could easily start with an inexpensive server, and simply add more inexpensive servers as you grew. If your $3,000 database server couldn't handle the load, you'd simply buy a second (or third, or forth) until it could. If you needed more storage capacity, you'd add an NFS server.

For this to work, though, web applications had to stop assuming that the same server would handle each request — or even each part of a single request. In a large-scale LAMP (and Django) deployment, as many as half a dozen servers might be involved in handling a single page! The repercussions of this are numerous, but in essence they come down to these points:

- State cannot be saved locally. In other words, any data that must be available between multiple requests must be stored in some sort of persistent storage like the database or a centralized cache.
- Software cannot assume that resources are local. For example, the web platform cannot assume that the database runs on the same server; it must be capable of connecting to a remote database server.
- Each piece of the stack must be easily moved or replicated: if Apache for some reason doesn't work for a given deployment, you should be able to swap it out for another server with a minimum of fuss.

  Or, on a hardware level: if a web server fails, you should be able to replace it with another physical box with minimum downtime. Remember, this whole philosophy is based around deployment on cheap, commodity hardware; failure is to be expected.

As you've probably come to expect, Django handles this more or less transparently — no part of Django violates these principles — but knowing the philosophy helps when it comes time to scale up.

But does it work?

This philosophy might sound good on paper (or on your screen), but does it actually work?

Well, instead of answering directly, let's instead look at a by-no-means-complete list of a few companies who've based their business on this architecture. You might recognize some names:

- Amazon
- Blogger
- Craigslist
- Facebook
- Google
- LiveJournal
- Slashdot
- Wikipedia
- Yahoo
- YouTube

To paraphrase the famous scene from *When Harry Met Sally…*: "we'll have what they're having!"

A note on personal preferences

Before we get into the details, a quick aside.

Open source is famous for its so-called "religious wars": much (digital) ink has been spilled arguing about text editors (`emacs` vs. `vi`), operating systems (Linux vs. Windows vs. MacOS), database engines (MySQL vs. PostgreSQL), and — of course — programming languages.

We try to stay away from these battles; there just isn't enough time.

However, there are a number of choices when it comes to deploying Django, and we're constantly asked for our preferences. Since stating these preferences comes dangerously close to firing a salvo in one of these battles, we've mostly refrained. However, for the sake of completeness and full disclosure, we'll state them here. We prefer:

- Linux — Ubuntu specifically — as our operating system.
- Apache and mod_python for the web server.
- PostgreSQL as a database server.

1

Of course, we can point to many Django users who have made other choices and done perfectly well.

How to use Django with Apache and mod_python

Apache with mod_python currently is the most robust setup for using Django on a production server.

mod_python is an Apache plugin which embeds Python within Apache and loads Python code into memory when the server starts. Code stays in memory throughout the life of an Apache process, which leads to significant performance gains over other server arrangements.

2

Django requires Apache 2.x and mod_python 3.x, and you should use Apache's prefork MPM, as opposed to the worker MPM.

2

> **Note**
>
> Configuring Apache is *well* out of the scope of this book, so we'll simply mention details as needed. Luckily, there are a number of great resources available if you need to learn more about Apache. A few of them we like are:
>
> - The free online apache documentation.
> - Pro Apache by Peter Wainwrite (Apress).
> - Apache: The Definitive Guide by Ben Laurie and Peter Laurie (O'Reilly).

Basic configuration

1

To configure Django with mod_python, first make sure you have Apache installed with the mod_python module activated. This usually means having a `LoadModule` directive in your Apache conf; it'll usually look something like:

```
LoadModule python_module /usr/lib/apache2/modules/mod_python.so
```

Then edit your Apache conf and add the following:

```
<Location "/">
    SetHandler python-program
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
    PythonDebug On
</Location>
```

Make sure to replace `mysite.settings` with the appropriate `DJANGO_SETTINGS_MODULE` for your site.

This tells Apache: "Use mod_python for any URL at or under '/', using the Django mod_python handler." It passes the value of `DJANGO_SETTINGS_MODULE` so mod_python knows which settings to use.

Note that we're using the `<Location>` directive, not the `<Directory>` directive. The latter is used for pointing at places on your filesystem, whereas `<Location>` points at places in the URL structure of a Web site. `<Directory>` would be meaningless here.

4

Also, if you've manually altered your `PYTHONPATH` to put your Django project on it, you'll need to tell mod_python:

```
PythonPath "['/path/to/project'] + sys.path"
```

You can also add directives such as `PythonAutoReload Off` for performance. See the mod_python documentation for a full list of options.

Note that you should set `PythonDebug Off` on a production server. If you leave `PythonDebug On`, your users would see ugly (and revealing) Python tracebacks if something goes wrong within mod_python.

Restart Apache, and any request to your site (or virtual host if you've put this directive inside a `<VirtualHost>` block) will be served by Django.

> **Note**
>
> If you deploy Django at a subdirectory — that is, somewhere deeper that "/" — Django *won't* trim the URL prefix off of your URLpatterns. So, if your Apache config looks like:
>
> ```
> <Location "/mysite/">
>     SetHandler python-program
>     PythonHandler django.core.handlers.modpython
>     SetEnv DJANGO_SETTINGS_MODULE mysite.settings
>     PythonDebug On
> </Location>
> ```
>
> The *all* your URL patterns will need to start with `"/mysite/"`. For this reason we usually recommend deploying Django at the root of your domain or virtual host.

Multiple Django installations on the same Apache instance

It's entirely possible to run multiple Django installations on the same Apache instance. Just use `VirtualHost` for that, like so:

```
NameVirtualHost *

<VirtualHost *>
    ServerName www.example.com
    # ...
    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
</VirtualHost>

<VirtualHost *>
    ServerName www2.example.com
    # ...
    SetEnv DJANGO_SETTINGS_MODULE mysite.other_settings
</VirtualHost>
```

If you need to put two Django installations within the same `VirtualHost`, you'll need to take a special precaution to ensure mod_python's code cache doesn't mess things up. Use the `PythonInterpreter` directive to give different `<Location>` directives separate interpreters:

```
<VirtualHost *>
    ServerName www.example.com
    # ...
    <Location "/something">
        SetEnv DJANGO_SETTINGS_MODULE mysite.settings
        PythonInterpreter mysite
    </Location>

    <Location "/otherthing">
        SetEnv DJANGO_SETTINGS_MODULE mysite.other_settings
        PythonInterpreter mysite_other
    </Location>
</VirtualHost>
```

The values of `PythonInterpreter` don't really matter, as long as they're different between the two `Location` blocks.

Running a development server with mod_python

Because mod_python caches loaded Python code, when deploying Django sites on mod_python you'll need to restart Apache each time you make changes to your code. This can be a hassle, so here's a quick trick to avoid this:

1

Just add `MaxRequestsPerChild 1` to your config file to force Apache to reload everything for each request. But don't do that on a production server, or we'll revoke your Django privileges.

2

If you're the type of programmer who debugs using scattered `print` statements, note that `print` statements have no effect in mod_python; they don't appear in the Apache log, as one might expect. If you have the need to print debugging information in a mod_python setup, you'll probably want to use Python's standard logging package or add the debugging information to the template of your page.

Serving Django and media files from the same Apache instance

Django doesn't serve media files itself; it leaves that job to whichever Web server you choose. We recommend using a separate Web server — i.e., one that's not also running Django — for serving media; see the section on scaling, below.

If, however, you have no option but to serve media files on the same Apache `VirtualHost` as Django, here's how you can turn off mod_python for a particular part of the site:

```
<Location "/media/">
    SetHandler None
</Location>
```

Change `Location` to the root URL of your media files.

4

You can also use `<LocationMatch>` to match a regular expression. For example, this sets up Django at the site root but explicitly disables Django for the `media` subdirectory and any URL that ends with `.jpg`, `.gif` or `.png`:

```
<Location "/">
    SetHandler python-program
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
</Location>

<Location "media">
    SetHandler None
</Location>

<LocationMatch "\.(jpg|gif|png)$">
    SetHandler None
</LocationMatch>
```

1

Error handling

When you use Apache/mod_python, errors will be caught by Django — in other words, they won't propagate to the Apache level and won't appear in the Apache `error_log`.

The exception for this is if something is really messed up in your Django setup. In that case, you'll see an "Internal Server Error" page in your browser and the full Python traceback in your Apache `error_log` file. The `error_log` traceback is spread over multiple lines. (Yes, this is ugly and rather hard to read, but it's how mod_python does things.)

1

If you get a segmentation fault

Sometimes, Apache segfaults when you install Django. When this happens, it's almost *always* one of two causes mostly unrelated to Django itself:

- It may be that your Python code is importing the `pyexpat` module (used for XML parsing), which may conflict with the version embedded in Apache. For full information, see Expat Causing Apache Crash.
- It may be because you're running mod_python and mod_php in the same Apache instance, with MySQL as your database backend.

  In some cases, this causes a known mod_python issue due to version conflicts in PHP and the Python MySQL backend. There's full information in a mod_python FAQ entry.

If you continue to have problems setting up mod_python, a good thing to do is get a barebones mod_python site working, without the Django framework. This is an easy way to isolate mod_python-specific problems. Getting mod_python Working details this procedure.

The next step should be to edit your test code and add an import of any Django-specific code you're using — your views, your models, your URLconf, your RSS configuration, etc. Put these imports in your test handler function and access your test URL in a browser. If this causes a crash, you've confirmed it's the importing of Django code that causes the problem. Gradually reduce the set of imports until it stops crashing, so as to find the specific module that causes the problem. Drop down further into modules and look into their imports, as necessary.

How to use Django with FastCGI

2

Although Django under Apache and mod_python is the most robust depoyment setup, many people use shared hosting on which FastCGI is the only viable option.

1

Additionally, in some situations, FastCGI allows better security and possibly better performance than mod_python. For small sites, FastCGI can also be more light-weight than Apache.

What is FastCGI?

FastCGI is an efficient way of letting an external application serve pages to a Web server. The Web server

delegates the incoming Web requests (via a socket) to FastCGI, which executes the code and passes the response back to the Web server, which, in turn, passes it back to the client's Web browser.

Like mod_python, FastCGI allows code to stay in memory, allowing requests to be served with no startup time. Unlike mod_python, a FastCGI process doesn't run inside the Web server process, but in a separate, persistent process.

> **Why run code in a separate process?**
>
> The traditional `mod_*` arrangements in Apache embed various scripting languages (most notably PHP, Python and Perl) inside the process space of your Web server. Although this lowers startup time — because code doesn't have to be read off disk for every request — it comes at the cost of memory use.
>
> Each Apache process gets a copy of the Apache engine, complete with all the features of Apache that Django simply doesn't take advantage of. FastCGI processes, on the other hand, only have the memory overhead of Python and Django.
>
> Due to the nature of FastCGI, it's also possible to have processes that run under a different user account than the Web server process. That's a nice security benefit on shared systems, because it means you can secure your code from other users.

Prerequisite: `flup`

Before you can start using FastCGI with Django, you'll need to install flup, which is a Python library for dealing with FastCGI. Some users have reported stalled pages with older `flup` versions, so you may want to use the latest SVN version.

Running your FastCGI server

FastCGI operates on a client-server model, and in most cases you'll be starting the FastCGI server process on your own. Your Web server (be it Apache, lighttpd, or otherwise) only contacts your Django-FastCGI process when the server needs a dynamic page to be loaded. Because the daemon is already running with the code in memory, it's able to serve the response very quickly.

> **Note**
>
> If you're on a shared hosting system, you'll probably be forced to use Web server-managed FastCGI processes. See the section below on running Django with Web server-managed processes for more information.

A Web server can connect to a FastCGI server in one of two ways: It can use either a Unix domain socket (a "named pipe" on Win32 systems), or it can use a TCP socket. What you choose is a manner of preference; a TCP socket is usually easier due to permissions issues.

To start your server, first change into the directory of your project (wherever your `manage.py` is), and then run `manage.py` with the `runfcgi` option:

```
./manage.py runfcgi [options]
```

If you specify `help` as the only option after `runfcgi`, it'll display a list of all the available options.

You'll need to specify either a `socket` or both `host` and `port`. Then, when you set up your Web server, you'll just need to point it at the host/port or socket you specified when starting the FastCGI server.

A few examples should help explain this.

- Running a threaded server on a TCP port:

```
./manage.py runfcgi method=threaded host=127.0.0.1 port=3033
```

- Running a preforked server on a Unix domain socket:

```
./manage.py runfcgi method=prefork socket=/home/user/mysite.sock pidfile=django.pid
```

- Run without daemonizing (backgrounding) the process (good for debugging):

```
./manage.py runfcgi daemonize=false socket=/tmp/mysite.sock
```

Stopping the FastCGI daemon

If you have the process running in the foreground, it's easy enough to stop it: Simply hitting `Ctrl-C` will stop and quit the FastCGI server. However, when you're dealing with background processes, you'll need to resort to the Unix `kill` command.

If you specify the `pidfile` option to your `manage.py runfcgi`, you can kill the running FastCGI daemon like this:

```
kill `cat $PIDFILE`
```

…where `$PIDFILE` is the `pidfile` you specified.

To easily restart your FastCGI daemon on Unix, you could use this small shell script:

```
#!/bin/bash

# Replace these three settings.
PROJDIR="/home/user/myproject"
PIDFILE="$PROJDIR/mysite.pid"
SOCKET="$PROJDIR/mysite.sock"

cd $PROJDIR
if [ -f $PIDFILE ]; then
    kill `cat -- $PIDFILE`
    rm -f -- $PIDFILE
fi

exec /usr/bin/env - \
  PYTHONPATH="../python:.." \
  ./manage.py runfcgi socket=$SOCKET pidfile=$PIDFILE
```

Apache and FastCGI

To use Django with Apache and FastCGI, you'll need Apache installed and configured, with mod_fastcgi installed and enabled. Consult the Apache and mod_fastcgi documentation for instructions.

Once you've got that set up, point Apache at your Django FastCGI instance by editing the `httpd.conf` (Apache configuration) file. You'll need to do two things:

- Use the `FastCGIExternalServer` directive to specify the location of your FastCGI server.
- Use `mod_rewrite` to point URLs at FastCGI as appropriate.

Specifying the location of the FastCGI server

The `FastCGIExternalServer` directive tells Apache how to find your FastCGI server. As the FastCGIExternalServer docs explain, you can specify either a `socket` or a `host`. Here are examples of both:

```
# Connect to FastCGI via a socket / named pipe.
FastCGIExternalServer /home/user/public_html/mysite.fcgi -socket /home/user/mysite.sock

# Connect to FastCGI via a TCP host/port.
FastCGIExternalServer /home/user/public_html/mysite.fcgi -host 127.0.0.1:3033
```

In either case, the file `/home/user/public_html/mysite.fcgi` doesn't actually have to exist. It's just a URL used by the Web server internally — a hook for signifying which requests at a URL should be handled by FastCGI. (More on this in the next section.)

Using mod_rewrite to point URLs at FastCGI

The second step is telling Apache to use FastCGI for URLs that match a certain pattern. To do this, use the mod_rewrite module and rewrite URLs to `mysite.fcgi` (or whatever you specified in the `FastCGIExternalServer` directive, as explained in the previous section).

2

In this example, we tell Apache to use FastCGI to handle any request that doesn't represent a file on the filesystem and doesn't start with `/media/`. This is probably the most common case, if you're using Django's admin site:

```
<VirtualHost 12.34.56.78>
  ServerName example.com
  DocumentRoot /home/user/public_html
  Alias /media /home/user/python/django/contrib/admin/media
  RewriteEngine On
  RewriteRule ^/(media.*)$ /$1 [QSA,L]
  RewriteCond %{REQUEST_FILENAME} !-f
  RewriteRule ^/(.*)$ /mysite.fcgi/$1 [QSA,L]
</VirtualHost>
```

1

FastCGI and lighttpd

lighttpd is a lightweight Web server commonly used for serving static files. It supports FastCGI natively and, thus, is a good choice for serving both static and dynamic pages, if your site doesn't have any Apache-specific needs.

Make sure `mod_fastcgi` is in your modules list, somewhere after `mod_rewrite` and `mod_access`, but not after `mod_accesslog`. You'll probably want `mod_alias` as well, for serving admin media.

Add the following to your lighttpd config file:

```
server.document-root = "/home/user/public_html"
fastcgi.server = (
    "/mysite.fcgi" => (
        "main" => (
            # Use host / port instead of socket for TCP fastcgi
            # "host" => "127.0.0.1",
            # "port" => 3033,
            "socket" => "/home/user/mysite.sock",
            "check-local" => "disable",
        )
    ),
)
alias.url = (
    "/media/" => "/home/user/django/contrib/admin/media/",
)

url.rewrite-once = (
    "^(/media.*)$" => "$1",
    "^/favicon\.ico$" => "/media/favicon.ico",
    "^(/.*)$" => "/mysite.fcgi$1",
)
```

Running multiple Django sites on one lighttpd instance

lighttpd lets you use "conditional configuration" to allow configuration to be customized per host. To specify multiple FastCGI sites, just add a conditional block around your FastCGI config for each site:

```
# If the hostname is 'www.example1.com'...
$HTTP["host"] == "www.example1.com" {
    server.document-root = "/foo/site1"
    fastcgi.server = (
       ...
    )
    ...
}

# If the hostname is 'www.example2.com'...
$HTTP["host"] == "www.example2.com" {
    server.document-root = "/foo/site2"
    fastcgi.server = (
       ...
    )
    ...
}
```

You can also run multiple Django installations on the same site simply by specifying multiple entries in the `fastcgi.server` directive. Add one FastCGI host for each.

Running Django on a shared-hosting provider with Apache

Many shared-hosting providers don't allow you to run your own server daemons or edit the `httpd.conf` file. In these cases, it's still possible to run Django using Web server-spawned processes.

1

> **Note**
>
> If you're using Web server-spawned processes, as explained in this section, there's no need for you to start the FastCGI server on your own. Apache will spawn a number of processes, scaling as it

needs to.

In your Web root directory, add this to a file named `.htaccess`

```
AddHandler fastcgi-script .fcgi
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^(.*)$ mysite.fcgi/$1 [QSA,L]
```

3

Then, create a small script that tells Apache how to spawn your FastCGI program. Create a file `mysite.fcgi` and place it in your Web directory, and be sure to make it executable

```
#!/usr/bin/python
import sys, os

# Add a custom Python path.
sys.path.insert(0, "/home/user/python")

# Switch to the directory of your project. (Optional.)
# os.chdir("/home/user/myproject")

# Set the DJANGO_SETTINGS_MODULE environment variable.
os.environ['DJANGO_SETTINGS_MODULE'] = "myproject.settings"

from django.core.servers.fastcgi import runfastcgi
runfastcgi(method="threaded", daemonize="false")
```

Restarting the spawned server

If you change any Python code on your site, you'll need to tell FastCGI the code has changed. But there's no need to restart Apache in this case. Rather, just re-upload `mysite.fcgi` — or edit the file — so that the time stamp on the file will change. When Apache sees the file has been updated, it will restart your Django application for you.

If you have access to a command shell on a Unix system, you can accomplish this easily by using the `touch` command:

```
touch mysite.fcgi
```

1

Scaling

Now that you know how to get Django running on a single server, let's look at how you'd scale out a Django installation. This section will walk through how a site might scale from a single server to a large-scale cluster that could serve millions of hits an hour.

2

It's important to note, however, that nearly every large site is large in different ways, so scaling is anything but a one-size-fits-all operation. The following walkthrough should suffice to show the general principle, and we'll try to point out where different choices could me made whenever possible.

2

First off, we'll make a pretty big assumption and exclusively talk about scaling under Apache and mod_python. Though we know of a number of successful medium- to large-scale FastCGI deployments, we're simply much more familiar with Apache.

A single server

1

Most sites start out running on a single server, with an architecture that looks something like this:

1

This works just fine for small-to-medium sites, and is extremely cheap — you can put together a single-server designed for Django for under $3,000.

1

However, as traffic increases you'll quickly run into **resource contention** between the different pieces of software. Database servers and web servers *love* to have the entire server to themselves, so when run on the same server they often end up "fighting" over the same resources (RAM, CPU) which they'd prefer to monopolize.

This is solved easily by moving the database server to a second machine.

Separating out the database server

1

As far as Django is concerned, this is extremely easy: you'll simply need to change the `DATABASE_HOST` setting to the IP or DNS name of your database server. It's probably a good idea to use the IP if at all possible; relying on DNS for the connection between your web and database server could be a big problem.

With a separate database server, our architecture now looks like:

Here we're starting to move into what's usually called **N-tier** architecture. Don't be scared by the buzzword: it just refers to the fact that different "tiers" of the web stack get separated out onto different physical machines.

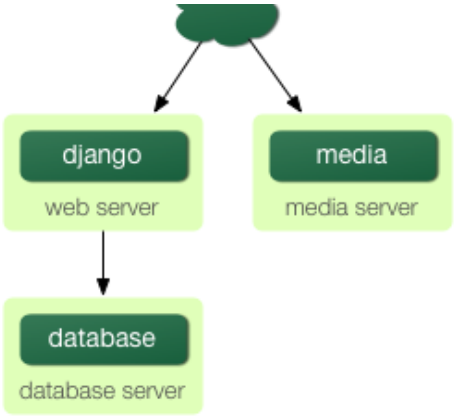
At this point if you anticipate ever needing to grow beyond a single database server, it's probably a good idea to start thinking now about connection pooling and/or database replication. There's not nearly enough space to do those topics justice in this book — unfortunately — so you'll need to consult your database's documentation and/or community for more information.

A separate media server

We've still got a big problem left over from the single server setup: the serving of media from the same box that handles dynamic content.

2

Those two activities perform best under different circumstances, and by smashing them together on the same box you end up with neither performing particularly well. So the next step is to separate out the media — that is, anything *not* generated by a Django view — onto a dedicated server:



3

Ideally, this media server should run a stripped-down web server optimized for static media deliver. lighttpd and tux are both excellent choices here, but a heavily stripped down Apache could work, too.

For sites heavy in static content — photos, videos, etc. — moving to a separate media server is doubly important, and should likely be the *first* step in scaling up.

This step can be slightly tricky, however. Django's admin needs to be able to write uploaded media to the media server (the `MEDIA_ROOT` setting controls where this media is written). If media lives on another server, however, you'll need to arrange a way for that write to happen across the network.

The easiest way to do this is to simply use NFS to mount the media server's media directories onto the web server(s). If you mount them in the same location pointed to by `MEDIA_ROOT`, media uploading will Just Work™.

Load balancing and redundancy

At this point, we've now broken things down as much as possible. This three-server setup should handle a very large amount of traffic — we served around ten million hits a day from an architecture of this sort — so if you grow further, you'll need to start adding redundancy.

This is a good thing, actually: one glance at the above diagram shows you that if even a single one of your three servers fails, you'll bring down your entire site. So as you add redundant servers, you not only increase capacity, you also increase reliability.

1

For the sake of this example, let's assume that the web server hits capacity first. It's relatively easy to get multiple copies of a Django site running on different hardware — just copy all the code onto multiple machines, and start Apache on both of them.

1

However, you'll need another piece of software to distribute traffic over your multiple servers: a **load balancer**. You can buy expensive and proprietary hardware load balancers, but there are a few incredibly high-quality open-source software load balancers out there.

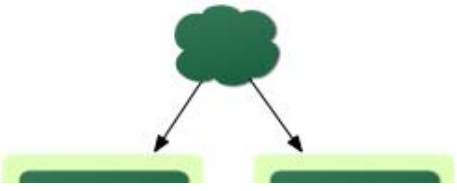
2

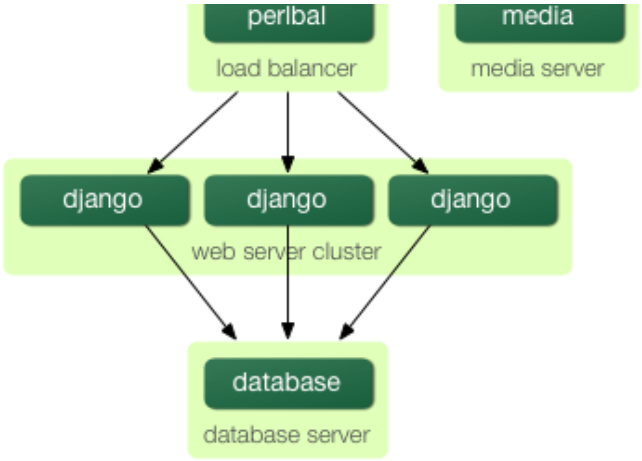
Apache's `mod_proxy` is one option, but we've found Perlbal to be simply fantastic. It's a load balancer and reverse proxy written by the same folks that wrote `memcached` (see Chapter 14).

> **Note**
>
> If you're using FastCGI, you can accomplish this same distribution/load balancing step by separating your front-end web servers and back-end FastCGI processes onto different machines. The front-end server essentially becomes the load balancer and the back-end FastCGI processes replace the Apache/mod_python/Django servers.

3

With the web servers now clustered, our evolving architecture starts to look more complex:





Notice that in the diagram the web servers are referred to as a "cluster" to indicate that the number of servers is basically variable; once you have a load balancer out front, you can easily add and remove backend web servers without a bit of downtime.

Going big

At this point, the next few steps are pretty much derivatives of the last one:

1

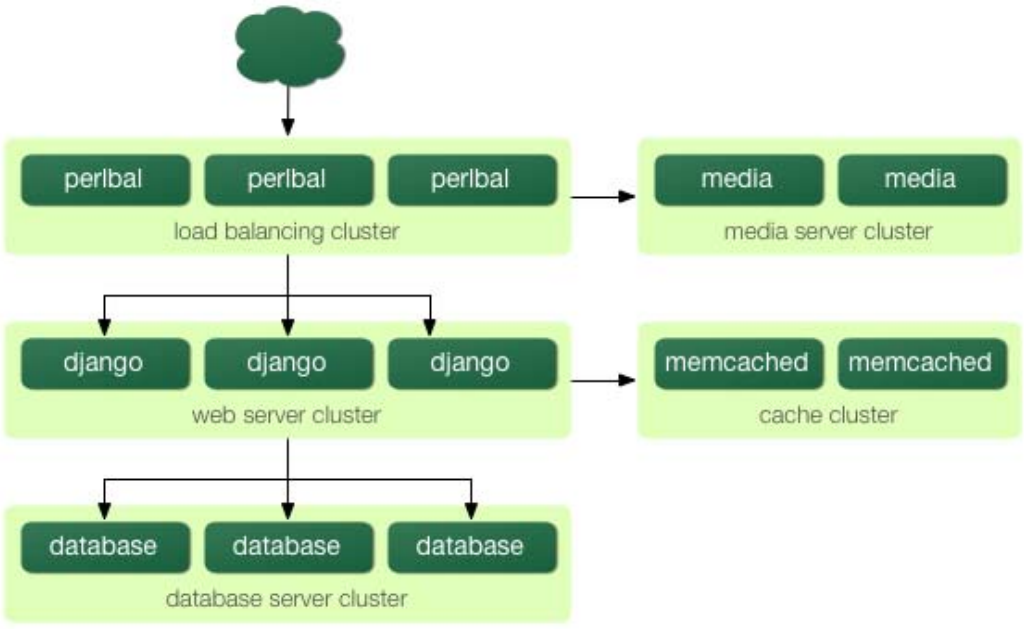
- As you need more database performance, you'll need to add replicated database servers. MySQL includes built-in replication; PostgreSQL users should look into Slony and pgpool for replication and connection pooling, respectively.

2

- If the single load balancer isn't enough, you can add more load balancer machines out front and distribute amount them using round-robin DNS.
- If a single media server doesn't suffice, you can add more media servers and distribute the load with your load balancing cluster.
- If you need more cache storage, you can add dedicated cache servers.
- At any stage, if a cluster isn't performing well you can add more servers to the cluster.

1

After a few of these iterations, a large-scale architecture might looks like:



2

Though we've only shown two or three servers at each level, there's fundamentally no limit to how many you can add.

Performance tuning

If you've got a huge amount money, you can just keep throwing hardware at scaling problems. For the rest of us, though, performance tuning is a must.

> **Note**
>
> Incidentally, if anyone with monstrous gobs of cash is actually reading this book, please consider a substantial donation to the Django project. We accept uncut diamonds and gold ingots, too.

Unfortunately, performance tuning is much more of an art than a science, and is even more difficult to write about than scaling. If you're serious about deploying a large-scale Django application, you should spend a large amount of time learning how to tune each piece of your stack.

Here, though, are a few Django-specific tuning tips we've discovered over the years:

There's no such thing as too much RAM.

Even really expensive RAM costs only about $200 per gigabyte — pennies compared to the time spent tuning elsewhere. Buy as much RAM as you can possibly afford, and then buy a little bit more.

Faster processors really won't improve performance all that much; most web servers spend up to 90% of their time waiting on disk IO. As soon as you start swapping, performance will just die. Faster disks might help slightly, but they're much more expensive than RAM that it doesn't really matter.

If you've got multiple servers, the first place to put your RAM is in the database server. If you can afford it, get enough RAM to get fit your entire database into memory. This shouldn't be too hard; LJWorld.com's database — including over half a million articles dating back to 1989 — is under 2 GB.

Next max out the RAM on your web server. The ideal situation is one where neither swaps — ever. If you get to that point you should be able to withstand most normal traffic.

Turn off `Keep-Alive`

Keep-alive is a feature of HTTP that allows multiple HTTP requests to be served over a single TCP connection, avoiding the TCP setup/teardown overhead.

This sounds good at first glance, but can actually kill performance of a Django site. If you're properly serving media from a separate server, each user browsing your site will actually only a page from your Django server every 10 seconds at best. This leaves HTTP servers waiting around for the next keep-alive request, and a idle HTTP server just consumes RAM that an active one should be using.

Use memcached

Although Django supports a number of different cache backends, none of them even come *close* to being as fast as memcached. If you've got a high traffic site, don't even both with the other backends; go straight to memcached.

Use memcached often

Of course, selecting memcached does you no good if you don't actually use it. Chapter 14 is your best friend here: learn how to use Django's cache framework, and use it everywhere possible. Aggressive, preemptive caching is usually the only think that will keep a site up under major traffic.

Join the conversation

Each piece of the Django stack — from Linux to Apache to PostgreSQL or MySQL — has an awesome community behind it. If you really want to get that last 1% out of your servers, join the open-source communities behind your software and ask for help. Most free software community members will be thrilled to help.

And also be sure to join the Django community. Your humble authors are only two members of an incredible active, growing group of Django developers; our community has a huge amount of collective experience to offer.

Good luck!

5

We wish you the best of luck in running your Django site, whether it's a little toy for you and a few friends, or the next Google.

« previous ◇ table of contents