How to Scale Django: Beyond the Basics

Authored by: ***ASPHostServer Administrator*** *[asphostserver@gmail.com]*
Saved From: *http://faq.asphosthelpdesk.com/article.php?id=254*

Let's dig into the guts of our application and server configuration a little. This article is written on the assumption that you're using Ubuntu 12.04, but the principles work with any version of Linux.

Cache Everything

The downside is that Varnish, unless you configure it carefully, will cache your entire site. If you have dynamic content, for example if your site is more of an app, Varnish can cause a lot of trouble. Imagine that user A visits the site and adds an item to their shopping cart. Then user B visits the site and sees the cached version of the shopping cart with user A's item in it. Not good.

Another downside is that Django, by default, tries to force upstream caches from caching the content, which causes a bit of a battle that ends in erratic behavior.

But I have great news: Django has it's own cache framework that will give you two things.

1. It will make your application more cache friendly so that it plays nice with Varnish (if you choose to use it)
- It will give you control over what parts of the site are cached

It is easy to turn on and avoids needing to run another server process. If you have enough RAM memcached provides an excellent cache backend. This article will demonstrate database based caching.

Peruse the documentation linked above for a lot of excellent detail, but here's what you need to do to get started.

Create a database table for the cache. In this case, our table is called "cache_table":

```
python manage.py createcachetable cache_table
```

Now edit your settings.py file and configure the cache by adding these lines:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.db.DatabaseCache',
        'LOCATION': 'cache_table',
    }
}
```

There are additional configuration options you can configure. Read about them in the documentation. I'd especially check out TIMEOUT and the CULL_FREQUENCY.

Now you need to decide what to cache. If you just want to make your site work more nicely with Varnish then

check out the per-site cache configuration. However that's not very exciting, so let's dig into per-view caching.

Let's say you have a shopping cart product page that rarely changes and isn't very dynamic. You have a view called`product_detail(request, product_id)`that would be perfect for caching.

You can use a simple decorator to enable caching for this:

```
from django.views.decorators.cache import cache_page
@cache_page(60 * 60)
def product_detail(request, product_id):
    ...
```

That will allow that view to be cached for 60 minutes (60 seconds * 60 minutes).

Again, I'd like to refer to the docs for some excellent tips. One particularly powerful capability is to vary based on headers. If you have a site that is translated into multiple languages, or you show different content based on browser version, this can be a great help.

What About Content that Changes Frequently?

If you have a semi-dynamic site, for example your application receives lots of comments or data is refreshed frequently, you may be concerned about using caching. This is a valid concern.

First identify all views that can be cached aggressively and then cache them.

Second, identify how long of a delay you could tolerate someone seeing stale content. For example, a site with active comments could probably tolerate a delay of 1 to 5 minutes. Cache pages as long as you can.

You'd be surprised -- even a cache timeout of 10 seconds can be very helpful. In many cases, a busy site that is suddenly overloaded is usually being hit by an onslaught of traffic to a particular page.

If 10,000 people visit a page on your site within a minute's time, it will melt your server when all of those queries to the database run at once. If you set a cache of 10 seconds, then that means the page will be regenerated only 6 times per minute, regardless of how many people request it. Any server can do that!

E-commerce sites and sites that use session data have to be very careful. Investigate the**vary**header options to see if caching can work in your situation. Don't be afraid to be creative!

Sites will sometimes serve all HTML pages as static content and load per-user session information using Ajax. In the case of an e-commerce site, users often won't notice if their shopping cart count takes 5 seconds to load as long as they can browse the rest of the page normally.

Serve Static Content

Django's built in development server happily serves static content. Often, I see Django sites configured for production use with the static content being served by Django.

The only time you should do this is if you need to control access to the content. But even then, consider other ways.

- **Dynamically generated CSS**- No, use a build tool like Grunt or Less to build your CSS before deployment. If it needs to be generated on the server periodically then use a background task.

- **Authenticated user access**- Yes, you probably need to use Django for this.

- **Everything else**- No, let's configure static content hosting with your webserver.

I had to read through the documentation for static-files a few times before I got it. I think you should read through it yourself, but in case you struggle with it too, here's a scenario you can use:

On your local development machine, in "settings.py" add a line like this:

```
STATIC_ROOT = os.path.join(BASE_DIR, "static")
```

Then run the manage.py command like this:

```
python manage.py collectstatic
```

You should now see in your project root a folder called "static" and in it you will see a lot of your static files, such as CSS and Javascript. Particularly if you have enabled the admin app.

To use these files you'll need to configure Django so that it knows where to look for them. In your settings.py folder you'll need a configuration option like this:

```
STATICFILES_DIRS = (
    os.path.join(BASE_DIR, "static"),
)
```

Now if you run the developer server and visit the admin area and view the source of any page you should see that the CSS files are being served from`/static/admin/css/`.

When you deploy your application, you can now point Apache or Nginx to serve files that begin with **/static/** by pointing them directly to the files.

If your app is deployed to **/srv/myapp/** and you therefore have a file called `/srv/myapp/static/admin/css/base.css`then you can configure your webserver like this:

Apache:

```
Alias /static/ "/srv/myapp/static/"
```

Nginx:

```
server {
    ...
    location /static/ {
        alias /srv/myapp/static/;
```

```
    }
}
```

In both cases you're just pointing the /static/ URL to the precise place on the hard drive where the files are found. Apache and Nginx can serve these much faster than Django can.

Use uWSGI

If I want to publish a site quickly, I use Apache's mod_wsgi. If I want the best performance I use uWSGI. In case you want to know how to pronounce that, it's "you whiskey".

uWSGI is a server process that runs your application outside of the webserver. For Apache you'll need to install these packages:

```
sudo apt-get install uwsgi uwsgi-plugin-python uwsgi-plugin-cgi
```

If you're using Apache as your webserver, also install:

```
sudo apt-get install libapache2-mod-uwsgi
```

Once you've done that, you'll notice that you now have two new folders for configuration: `/etc/uwsgi/apps-available`and`/etc/uwsgi/apps-enabled`, much like you'd see for your virtual host configuration with Apache or Nginx.

When you want to create a new application, you put a configuration file in`/etc/uwsgi/apps-available` and then create a symlink to it from apps-enabled.

Let's assume you've made a project called newproject and you have put it at`/srv/newproject`. As is typical for Django projects, there is a settings.py file at`/srv/newproject/newproject/settings.py`.

Create a file called `/etc/uwsgi/apps-available/newproject.ini and edit it to look like this:

```
[uwsgi]
touch-reload = /tmp/newproject
socket = 127.0.0.1:3031
workers = 2
chdir = /srv/newproject
env = DJANGO_SETTINGS_MODULE=newproject.settings
module = django.core.handlers.wsgi:WSGIHander()
```

Now create an empty file called`/tmp/newproject`:

```
touch /tmp/newproject
```

You have to activate the application by linking the file to apps-enabled:

```
sudo ln -s /etc/uwsgi/apps-available/newproject.ini
/etc/uwsgi/apps-enabled/newproject.ini
```

and then restart uWSGI:

```
sudo service uwsgi restart
```

If you check the output of "ps ax" you should see a few processes running related to your application.

Note that if you have more than one application running on the same server then you need to use a different port for each application's socket.

Also, if you ever update your application you should simply "touch" the file /tmp/newproject and uWSGI will reload itself.

At any time you can view`/var/log/uwsgi/app/newproject.log`to see information messages and errors.

Apache Configuration

If you're using mod_wsgi now is the moment of truth. You should disable mod_wsgi and enable mod_uwsgi. This may cause errors so do this on a test server until you find the right configuration.

```
sudo a2dismod wsgi
sudo a2enmod uwsgi
```

Now update the configuration for your app. In your apache config you may have a line like this:

```
WSGIScriptAlias / /srv/newproject/newproject.wsgi
```

And you will update to be a like this:

```
SetHandler uwsgi-handler
uWSGISocket 127.0.0.1:3031
```

Now reload your Apache configuration:

```
sudo service apache2 reload
```

Nginx Configuration

In your server configuration you will need to replace your existing wsgi directives with this:

```
uwsgi_pass      127.0.0.1:3031;
include         uwsgi_params;
uwsgi_param     UWSGI_SCHEME $scheme;
```

Pretty simple!

uWSGI Benefits

One of the beautiful aspects of using uWSGI is that you can limit the number of times your application is

running in memory. There will usually be one parent process and, if you used the config shared above, 2 workers. If you need more workers, you can simply edit the .ini file for the application and increase the number of "workers". Then reload uWSGI with "sudo service uwsgi reload" to update the change.

Before increasing the number of processes, consider how much server memory you're using. If you were running mod_wsgi before you should see a reduction in memory usage. Do watch the size of the uWSGI workers. They tend to increase in size as the application is used. Not because of memory leaks, but simply because the in-memory data set is small.

Maximizing Utilization

After your application has warmed up and been in use for a while, review your server's memory utilization.

Use the command`free -m`to see how much memory is being used by buffers/cache vs. applications. The key value is the the top number in the "free" column. Ideally a busy server will be using nearly all of the RAM.

If you find that you have RAM to spare and your site is slow then dig into what is being under-utilized.

Often times you'll see that processes are waiting for MySQL. It could be that you need to allocate more system memory to the database. This is extremely common because the default configuration for MySQL uses RAM very sparingly.

It is also possible that there is not enough RAM allocated to disk cache. Do you have unneeded processes taking up memory? By default, Linux will allocate as much RAM to disk cache as it can, grudgingly giving it up when applications need it.

In an ideal situation, which never occurs with memory constrained server environments, all of the read-only assets would be buffered in the cache. Therefore get rid of unneeded background processes.

Wrap Up

You've learned some techniques for caching, including caching the whole site with Varnish or controlling the cache settings on a per-view basis.

You've ensured static files are being served with the webserver, not Django.

You've learned how to use uWSGI to boost your Django app's performance by running it out of process from the webserver. (By the way, thanks to Ricardo Pascal for his excellent Apache/uWSGI documentation)

Finally, you've learned about allocating as much memory to your server applications as possible to maximize your utilization.