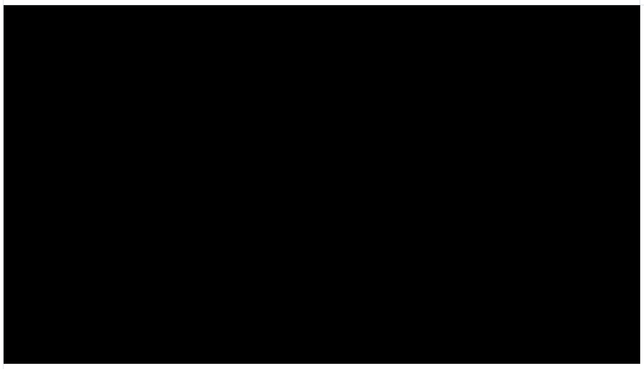
Lecture: Microcaching for Non-Personalized Dynamic Content



In this lesson, we'll look at an alternative caching strategy that is useful if you have an application with dynamic content that is receiving a lot of traffic.

Note: the commands in this video are run as the `root` user.

Documentation For This Video

- NGINX `http_uwsgi` module
- NGINX `uwsgi_cache_valid` directive
- NGINX `uwsgi_cache_key` directive
- NGINX `uwsgi_cache` directive

What is "Microcaching"?

If an application or website is updated very frequently, then caching becomes pretty difficult because you don't know when to remove the cache. If that same application receives a lot of traffic then routing all of that traffic to the application server can overwhelm the server quite quickly. NGINX is good at responding to a lot of traffic, and if the content of the website doesn't always need to be updated in real-time, then microcaching might be worth using. This is where "microcaching" comes into play. With microcaching, we're going to cache the contents of requests for just a few seconds, and for the same content, we'll only send one request every 10 seconds or so to the application server.

Implementing Microcaching

To demonstrate microcaching, we're going to add it to our UWSGI application and then send a lot of traffic to it. Before we add the caching, let's install [`boom`] and see how our application handles a large spike in traffic.

```
[root] $ pip install boom
```

With `boom` installed, we'll send traffic to `https://notes.example.com`:

```
[root] $ boom http://localhost/ --header Host:notes.example.com -c 30 -n 3000
Server Software: nginx/1.12.2
Running GET http://127.0.0.1:80/
       Host: notes.example.com
Running 3000 queries - concurrency 30
[=================================================================>.] 99% Done

-------- Results ---------
Successful calls                3000
Total time                      15.1118 s
Average                         0.1503 s
Fastest                         0.0444 s
Slowest                         0.1600 s
Amplitude                       0.1157 s
Standard deviation              0.006783
RPS                             198
BSI                             Pretty good

-------- Status codes ---------
Code 200                        3000 times.

-------- Legend ---------
RPS: Request Per Second
BSI: Boom Speed Index
```

Sending 3000 total requests using 30 concurrent users 15 seconds. That performance is bad really, but let's modify the configuration for `notes.example.com` to see if NGINX can make a difference for us without aggressively caching content like we did for the blog:

/etc/nginx/conf.d/notes.example.com.conf

```
uwsgi_cache_path /var/cache/nginx/micro levels=1:2
                 keys_zone=micro:10m max_size=1g;

server {
    listen 80;
    server_name notes.example.com;

    uwsgi_cache_key $scheme$request_method$host$request_uri;

    location /static {
        root /var/www/notes.example.com;
    }

    location / {
        add_header X-Cache-Status $upstream_cache_status;
        include uwsgi_params;
        uwsgi_pass unix:/var/run/uwsgi/notes.sock;
        uwsgi_cache micro;
        uwsgi_cache_valid 10s;
    }

}
```

The main difference between this configuration and the cache we set for the blog is with this line:

```
uwsgi_cache_valid 10s;
```

This means we're only caching content for 10 seconds. Since our original result took more than 10 seconds, we should be able to see a noticeable difference running the same load test. Let's see what kind of difference the microcaching gives us:

```
boom http://localhost/ --header Host:notes.example.com -c 30 -n 3000
Server Software: nginx/1.12.2
Running GET http://127.0.0.1:80/
       Host: notes.example.com
Running 3000 queries - concurrency 30
[=================================================================>.] 99% Done

-------- Results ---------
Successful calls                3000
Total time                      6.4929 s
Average                         0.0595 s
Fastest                         0.0200 s
Slowest                         0.0808 s
Amplitude                       0.0608 s
Standard deviation              0.004789
RPS                             462
BSI                             Pretty good

-------- Status codes ---------
Code 200                        3000 times.

-------- Legend ---------
RPS: Request Per Second
BSI: Boom Speed Index
```

Our Requests Per Second (RPS) went from 198 to 462! That's a huge increase in performance without needing to worry about returning stale content for a long period of time like we potentially could with more aggressive caching.

Knowing how long to cache information is a pretty hard problem to solve. If we have an application that doesn't deploy very often or that doesn't display that much dynamic data, then we can be aggressive with caching setting a longer `cache_valid` time. Microcaching is a good alternative to get some more performance out of your application servers when you do have dynamic content that is allowed to be stale for a short period of time.