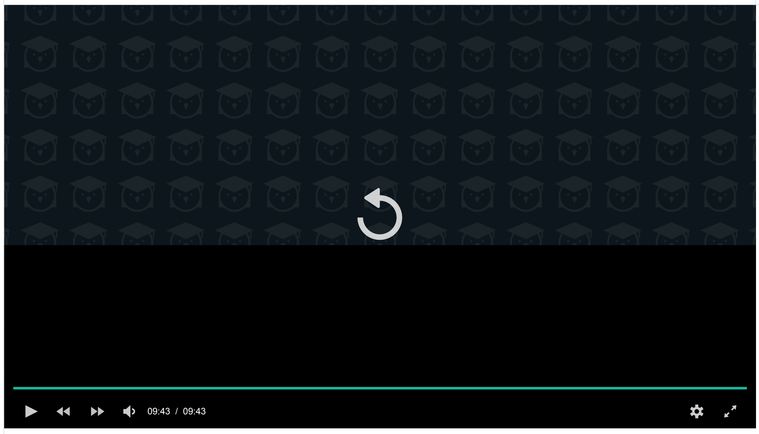
Lecture: Examining Load Balancing Methods



In this lesson, we look at the different ways we can configure NGINX to load balance traffic.

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

Documentation For This Video

- NGINX `http_upstream` module
- NGINX `upstream` directive
- NGINX `server` directive from the `upstream` module

The Default Load Balancing Strategy: Round-Robin

By default, the load balancer that we've already created for `photos.example.com` is routing traffic using a "round-robin" approach. This approach has an equal number of requests going to each server. If there are 4 requests made then, NGINX will cycle through the servers evenly until it runs out of requests, looking something like this:

```
Request 1 -> Server 1
Request 2 -> Server 2
Request 3 -> Server 3
Request 4 -> Server 1
...
```

The other available load balancing methods are:

- `hash` - Specifies how a key should be build based on the request, mapping all requests with the same key to the same server
- `ip_hash` - Specifies a client-server grouping based on the client IP address
- `least_conn` - Specifies that requests should be routed to the server with the least number of active connections. This approach also takes into account server weights

There is another load balancing method called `least_time` that routes requests to the server with the lowest average response time and least number of active connections, but this method is only available through NGINX Plus.

Setting a Different Load Balancing Method

Now that we know how each of the load balancing methods works, we should look at how to change them. To set the load balancing method for a specific `upstream` block we should use the corresponding directive before we specify any other directive. There are some potential ordering issues that you can run into with the `keepalive` directive, so it's best to set our load balancing method first. Here's what it would look like if we set our `photos.example.com` server group to use the `least_conn` method:

/etc/nginx/conf.d/photos.example.com.conf (partial)

```
upstream photos {
    least_conn;

    server 127.0.0.1:3000;
    server 127.0.0.1:3100;
    server 127.0.0.1:3101;

}

...
```

The only usage exception is if with the `hash` load balancing method is being used. For that method, we need to provide another parameter as the key. Here would be an example that routes to various servers based on the `$request_uri`:

/etc/nginx/conf.d/photos.example.com.conf (partial)

```
upstream photos {
    hash $request_uri;

    server 127.0.0.1:3000;
    server 127.0.0.1:3100;
    server 127.0.0.1:3101;

}

...
```

Weights & Passive Health Checking

Besides the load balancing method directives themselves, we can manipulate the load balancing that is done by adding more configuration to our upstream `server` directives.

Let's pretend that servers two and three are running on separate machines that we know to be underpowered. We're going to modify our configuration to achieve the following:

1. Prioritize sending traffic to server 1.
2. Temporarily remove servers 2 and 3 from the pool if they suffer 3 failed requests in 20 seconds. If this occurs, we'll have NGINX remove the server from the pool for 20 seconds.

We're using the round-robin approach again, and this is what our new configuration would look like:

/etc/nginx/conf.d/photos.example.com (partial)

```
upstream photos {
    server 127.0.0.1:3000 weight=2;
    server 127.0.0.1:3100 max_fails=3 fail_timeout=20s;
    server 127.0.0.1:3101 max_fails=3 fail_timeout=20s;
}
```

The `fail_timeout` option sets both the span of time to count failures and also the duration that a server with errors should be marked as "unavailable".

Looks like you haven't created any flash cards for this deck yet.

Create My First Flash Card

Popular Decks Popular Cards

Nginx Web Server Deep Dive

NGINX Web Server Deep Dive

NGINX Web Server Deep Dive