nginx.com

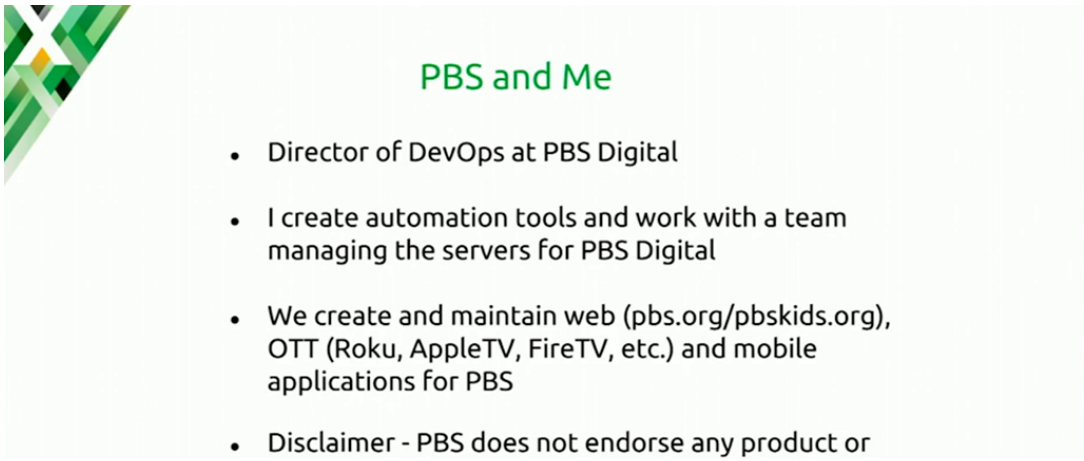
NGINX Proxy Cache Mitigates the Thundering Herd Problem

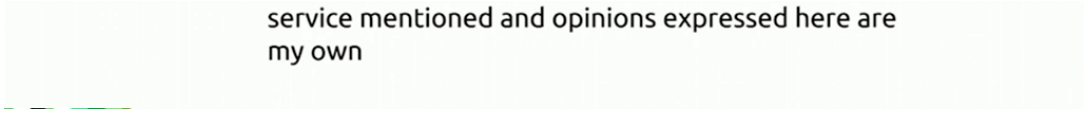
Mike Howsden of PBS DigitalDirector of DevOpsNovember 14, 2016

16-21 minutes

This post is adapted from a presentation delivered at nginx.conf 2016 by Mike Howsden of PBS Digital. You can view a recording of the presentation on YouTube.

Introduction



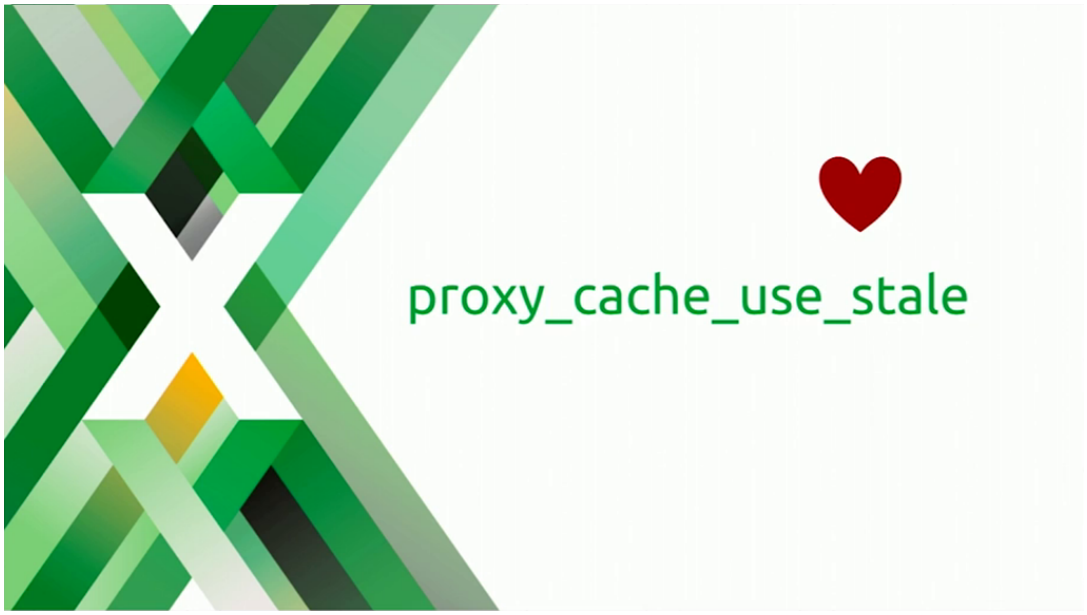


Mike Howsden: My name's Mike Howsden and I'm the Director of DevOps at PBS Digital.

I spend my days writing automation tools and working with a team that manages servers for PBS.
At PBS Digital, we create and maintain web OTT and mobile applications.

That being said, I have to note that PBS does not endorse any product or service mentioned, and the opinions expressed here are my own.

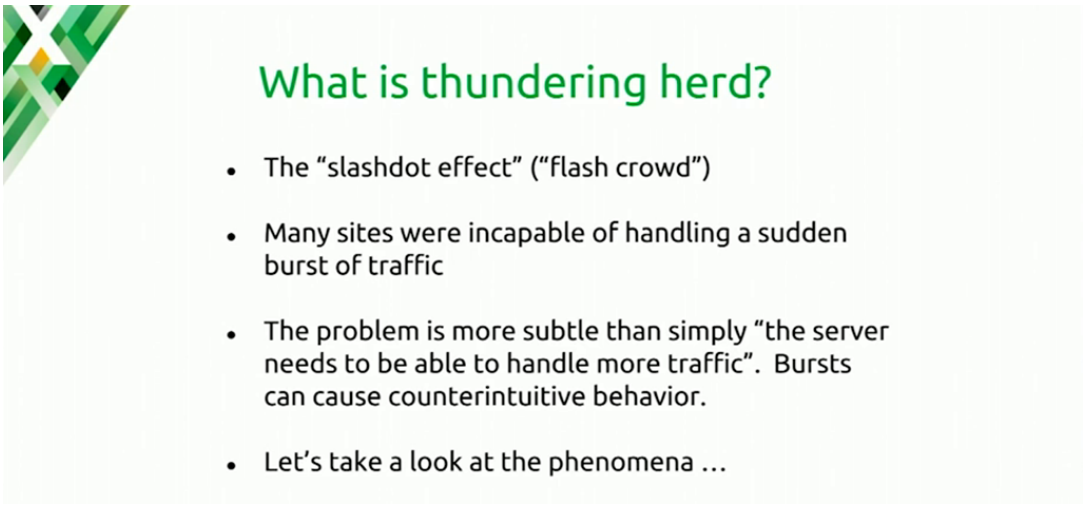
0:56 the `proxy_cache_use_stale` Directive



In any case, this isn't so much a story about PBS as it is the story of a configuration directive in NGINX – `proxy_cache_use_stale` – and the significance it can have when building websites that need to withstand high-traffic events.

I seriously love this NGINX directive. I've demoed the behavior of this directive to many co-workers over the years, and it's been the most useful single line of configuration that I've come across.

1:33 What Is Thundering Herd?



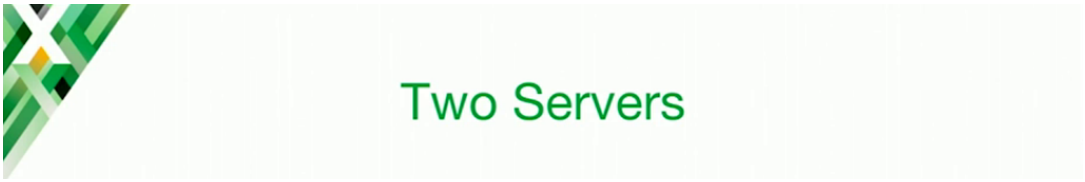
To explain the value of `proxy_cache_use_stale`, let me discuss a bit what I mean when I say “thundering herd”.

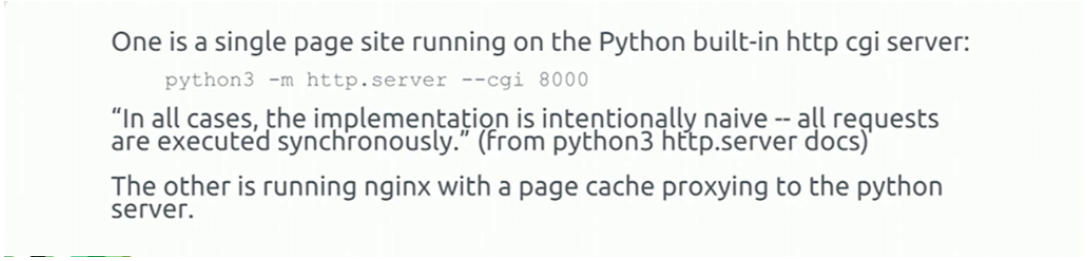
I’ve been creating websites for a couple of decades now. I don’t remember the first time I came across “thundering herd”, but I know that at the time, it was called the “slashdot effect”. Slashdot used to be an incredibly influential website, and many sites that became popular on Slashdot often were overwhelmed with huge amounts of traffic and became unusable.

Fast forward to five years ago and I’m building a very popular website for a media company where I need to be able to handle a significant amount of traffic regularly, with massive bursts on occasion. At the time, we were capable of running at scale, but those big bursts kept causing intermittent errors.

I had to take a closer look at the phenomenon. That inquiry led me to `proxy_cache_use_stale`, and I’m going to walk you through a simplified version of what I saw.

2:48 Two Servers





For the purposes of this presentation, we'll just use a single-page site running off the Python built-in CGI HTTP server, and another server running NGINX.

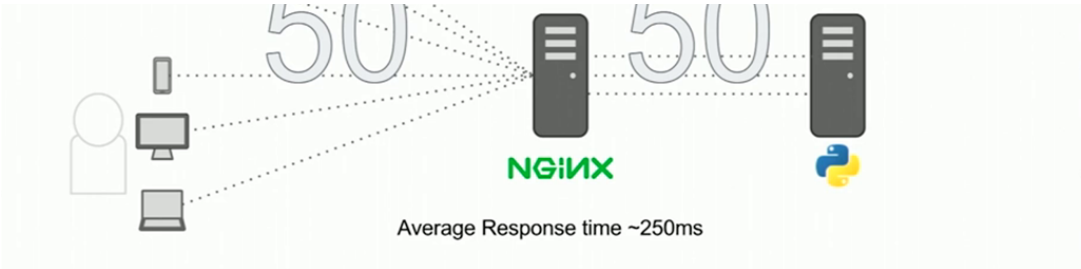
The Python built-in CGI server isn't very sophisticated at all, and it deliberately executes all requests synchronously. I chose it because it's less capable than almost anything you'd be working with in production.

3:22 The Python Code

The Python code I'll be running on the CGI server is very minimal. It just computes the current time and displays it.

Let's walk through a few different ways to configure this and see how well they work.

3:38 No Cache

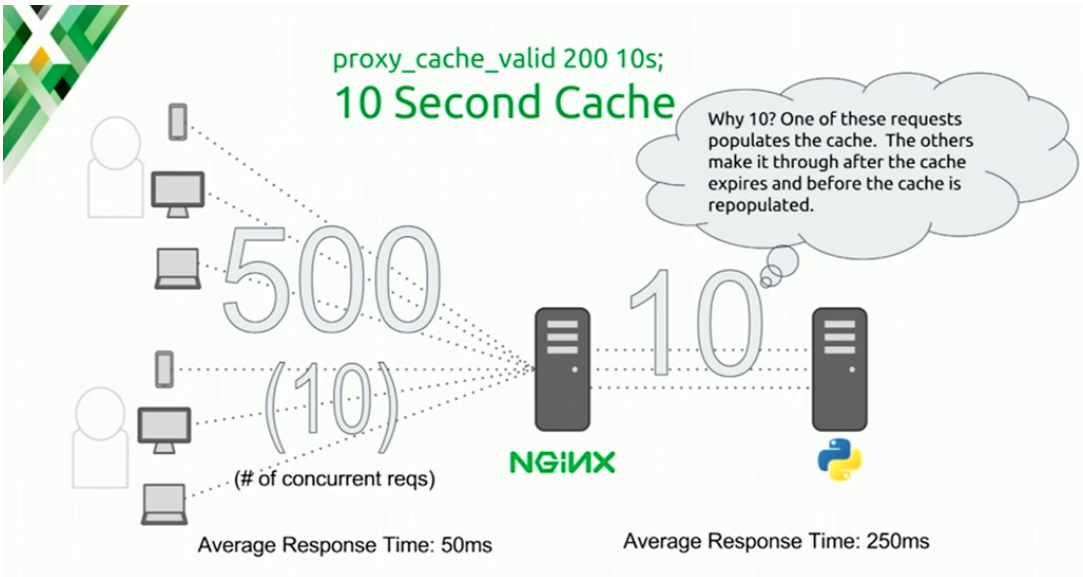


First, here's what it looks like with no caching at all.

Fifty requests come in from users, and fifty requests make it to the app server.

Let's say responses are coming in about every 215 milliseconds. This situation obviously doesn't scale. Python's built-in HTTP server is going to max out at around 80 requests per second.

4:05 10-Second Cache



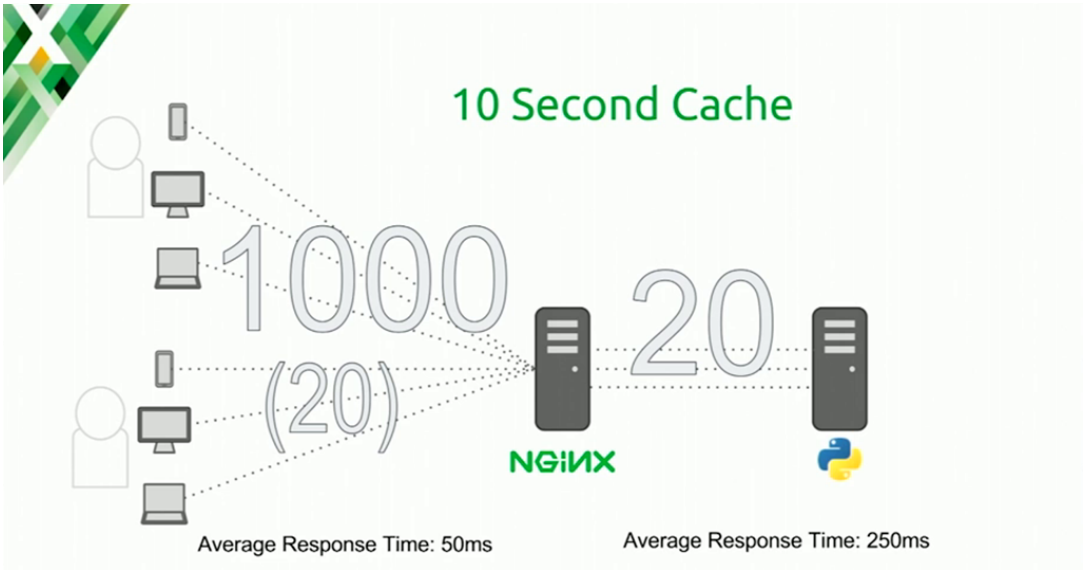
Now, let's add a 10-second cache. We do that in NGINX by defining a cache and using the `proxy_cache_valid` directive.

As you can see, the situation has drastically improved. We can now handle way more requests, and there are far fewer requests going to the application server.

The reason ten requests are making it to the application server is because those are coming in around the same time, and the cache hasn't had time to populate. Every 10 seconds, or however many

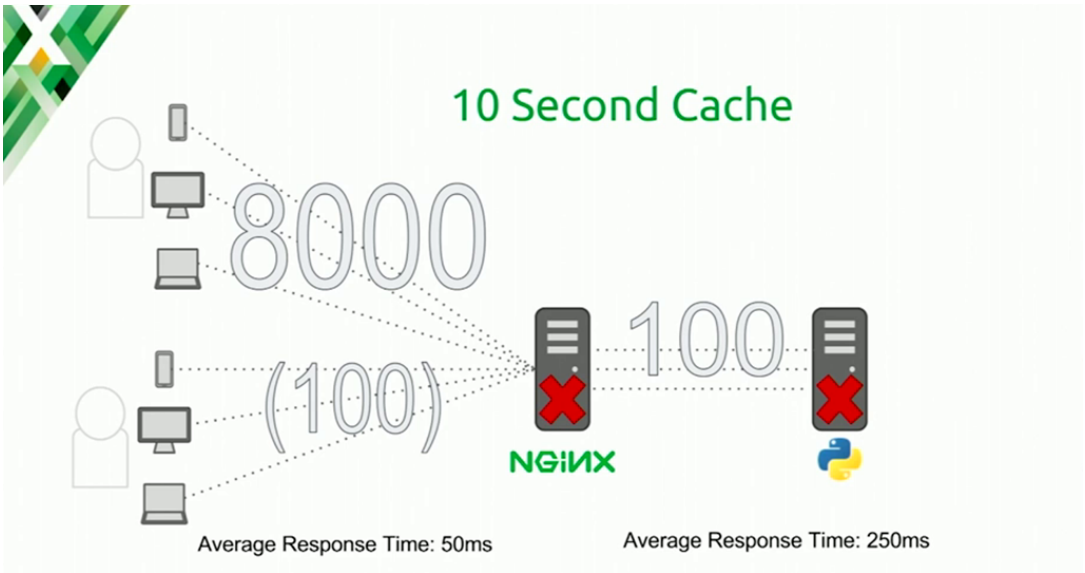
seconds you set the cache to, this little burst of traffic will come through. As the concurrent load increases, you'll have to watch out for this burst of traffic more and more.

4:55 More Load



Now, let's say the load had increased again and we have 1,000 requests per second so you get around 20 concurrent requests every 10 seconds. No big deal.

5:03 Even More Load



Then all of a sudden, some major event occurs and we have 8,000 requests per second and 100 are concurrent.

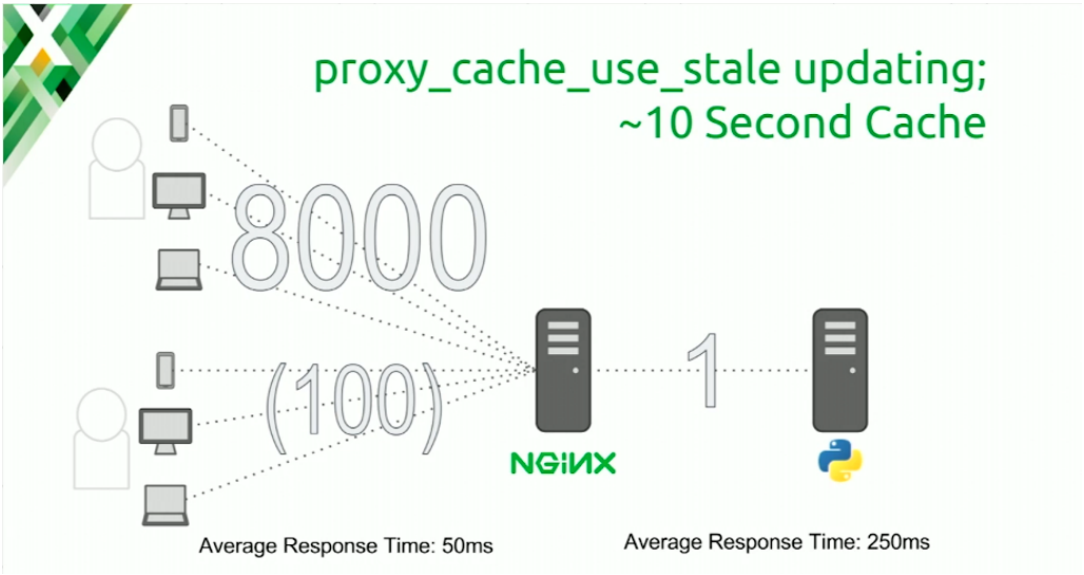
Uh-oh. Now we're exceeding the capacity of Python's built-in server. What happens?

Well, the backend server now can't handle requests and the page cache can't help much until that server comes back up.

We could solve the concurrent-request problem by adding more app servers, but that can get expensive, and may not be the best solution, so we want to try to solve it without adding more app servers.

5:49 Let's Try Using Stale Cache

Let's try using stale cache.

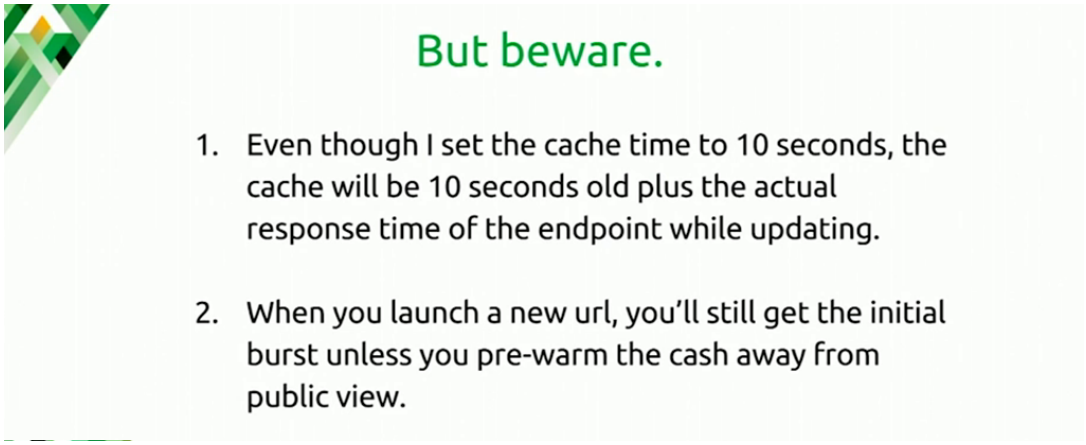


Here, we have the exact same situation as before, but now we've added the `proxy_cache_use_stale updating` directive.

Now, instead of those 100 concurrent requests making it to the application server, we have just 1 request going to the application server. Only 1 request goes through, because the others get stale responses while the cache is being populated.

Hopefully, now you're start to see the beauty of this little directive.

6:30 Stale Cache Caveats

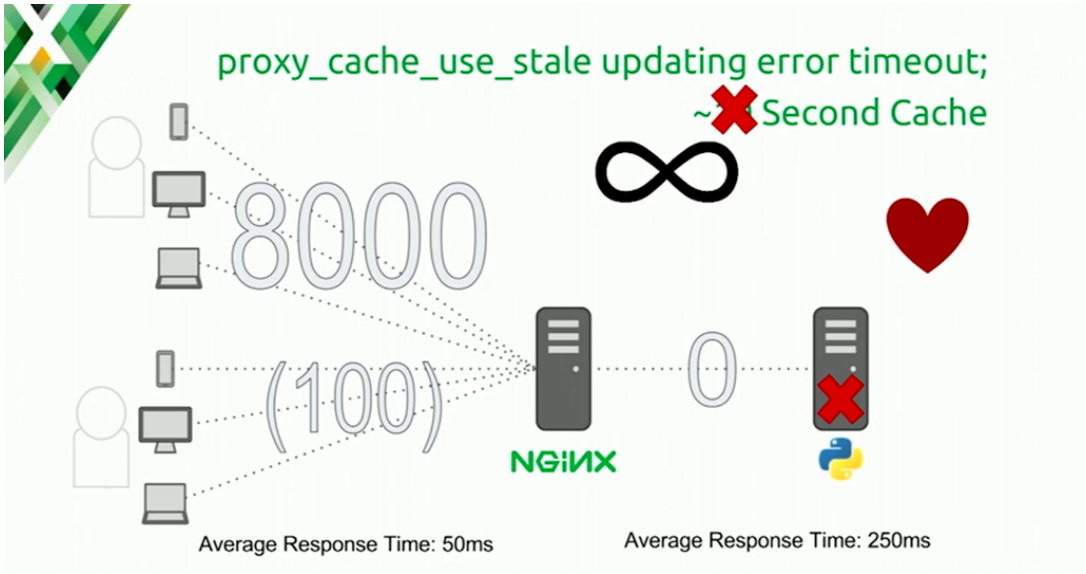


There are two caveats to be careful of.

First, even though I set the cache time to 10 seconds, the cache will be 10 seconds old plus the actual response time of the endpoint while updating.

Second, when you launch a new URL, you'll get the initial burst, unless you pre-warm the cache away from public view.

6:52 The `error` and `timeout` Parameters



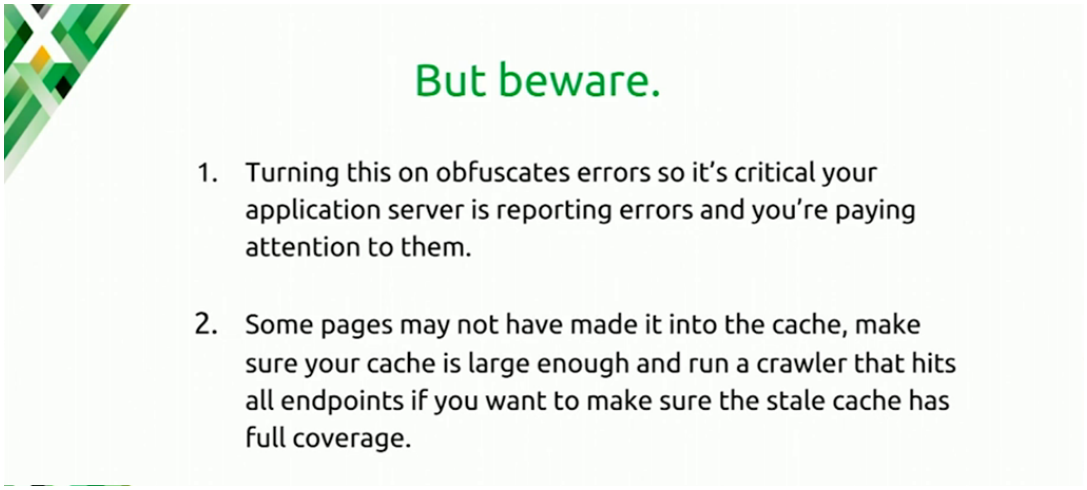
That was the `proxy_cache_use_stale updating` method. Now let's talk about the `error` and `timeout` [parameters].

So in this setup, what happens when the application server starts throwing errors or becomes unavailable from timeouts for whatever reason? Here's where we get some really cool effects.

The backend server went down and what does the user experience? Faster response times! As long as the URL has been visited frequently and is cached.

That means anybody visiting your most frequently used pages is having a good experience even in this non-ideal scenario. Users might be seeing errors, but only on less frequently used pages that either never made it into cache or are no longer in cache.

7:47 More Caveats

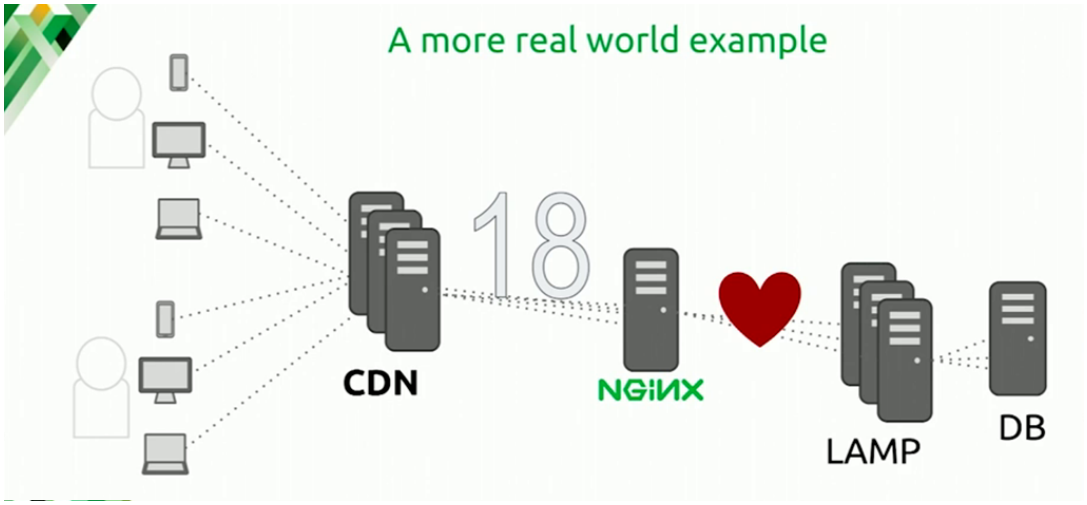


There's a couple things to be aware of here too.

One, turning this on obfuscates errors, so it's critical your application server is reporting errors and you're paying attention to them. Something like Sentry, New Relic, or similar is good there. You can live a long time off a broken app server and stale cache, and if you don't have proper monitoring in place, you may never know it.

Also, like I said, some pages may not have made it into the cache. Make sure your cache is large enough, and run a crawler that hits all endpoints if you want to ensure the stale cache has full coverage.

8:29 A More Real-World Example



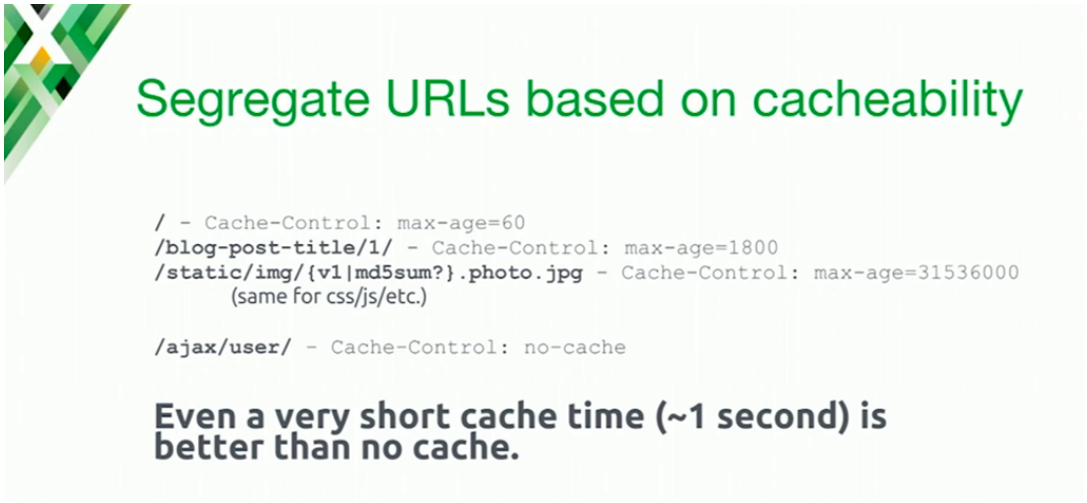
Now let's look at a more realistic architecture.

Here we have a CDN in front of NGINX, which is in front of a group of application servers running a LAMP stack. Probably still Python, maybe Django, hopefully not PHP.

The number of edge nodes is likely dependent on your CDN. In the case of CloudFront, it's at least 18 nodes.

A vanilla CDN is going to act as a page cache that does not use stale cache, so there's an advantage to having NGINX between the CDN and your application servers – it can take out those bursts. However, be careful, because depending on where and how you're setting cache control headers, you may be inadvertently doubling up on potential cache time.

9:21 Segregate URLS Based on Cacheability



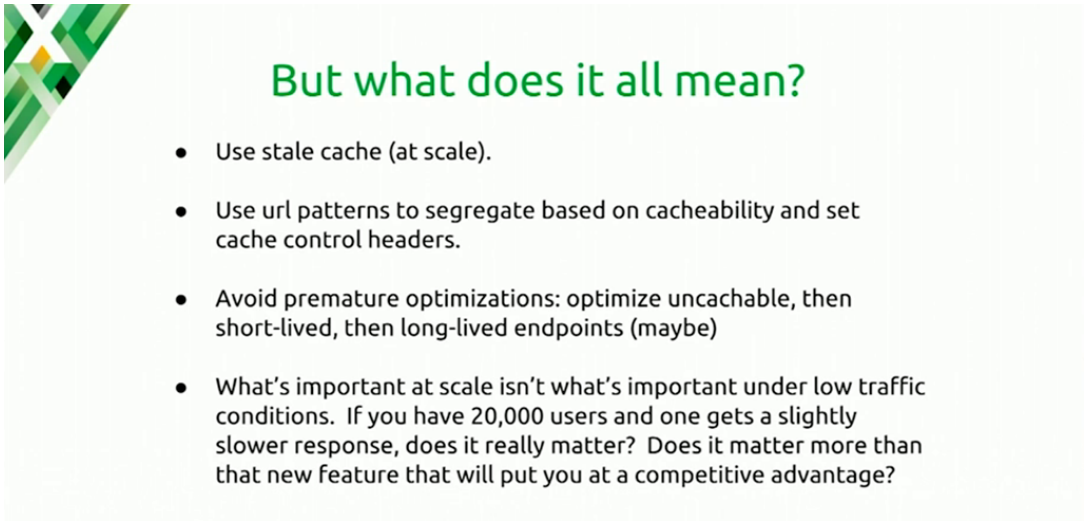
Now that we're talking about a real example, let's take a stab at segregating URL paths based on cacheability.
Here we have a blog site with a main home page set to a 1 minute cache. Each blog post is set to 30 minutes and large, static assets are set to 1 year.

Note that you should version everything with far future expiry via URL patterns so you don't have to purge the cache when you do a deploy. For some reason, lots of people still don't do that even though it's pretty well-documented at this point.

Finally, we have an AJAX endpoint that should not have a cache. It's best if you can make such endpoints optional, so that they can fail gracefully during high traffic events, especially if those features aren't usually what a user comes to your website for.

Remember that even a very short cache is significantly more desirable than an uncacheable endpoint; especially once you're using stale cache. Just having a 1 second cache can give you stale failover, whereas zero is going to give you nothing.

10:51 What Does It All Mean?



What does this all mean? It means you should use stale cache. It means you should segregate your URL paths based on cacheability. It means you have a new way to avoid premature

optimization.

What's important at scale isn't the same as what's important under low traffic conditions. If you have 20,000 users and one gets a slightly slower response, does it really matter? Does it matter more than the new feature that will put you at a competitive advantage?

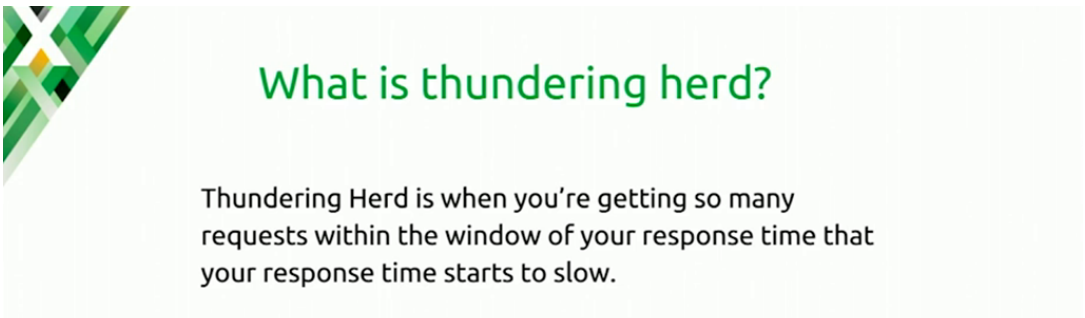
11:28 Demo

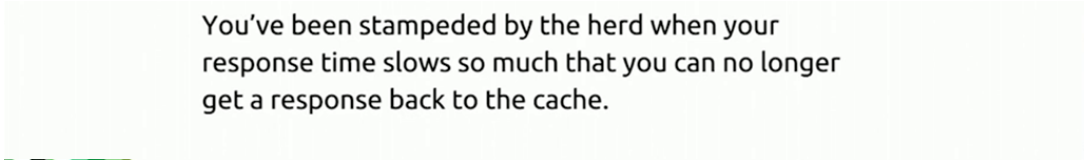
By now, hopefully you understand the value of this directive. But there's nothing quite like seeing it in action, so I'm going to demonstrate all this to you on my local machine.

Editor – The video below skips to the beginning of the demo at 12:07:

Ok, so that's a visual, live demonstration of all the concepts we've gone through. Obviously, for anything in production, you want to do better testing than what I'm doing here, but that was the basic concepts in action.

21:46 What Thundering Herd Really Is





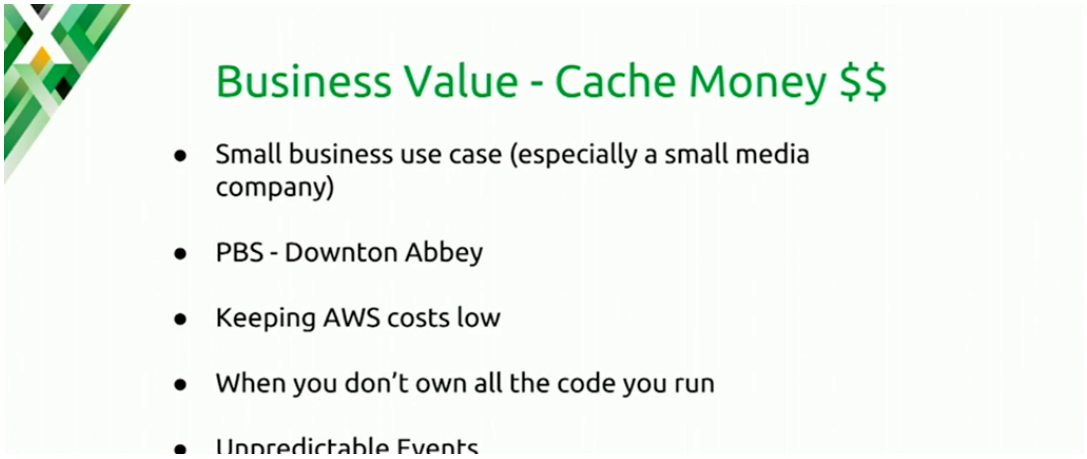
What thundering herd really is, from my point of view, is when you're getting so many requests within the window of your response time that your response time starts to slow.

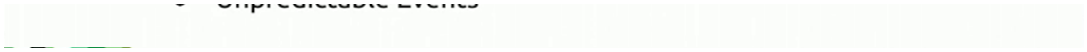
When I talk about concurrent requests, what I really mean is near-simultaneous requests that are coming in within your response window for that URL. So, if the average response time for your home page or whatever is 250 milliseconds, then the window for that URL is 250 milliseconds.

You've been stampeded by the herd when your response time slows so much that you can no longer get a response back to the cache. Say one request comes in, and while you're processing that request, a second request comes in. Now, that first request is getting a little bit slower to respond, right? If you get a big enough burst, you can get so many that that first response will never complete.

So the way to deal with this phenomenon, if it gets that big, is to use stale cache. With NGINX, I've been able to use the stale cache to deal with the thundering herd in a lot of high-traffic environments – major media websites.

23:54 Business Value





Let me just talk through a couple examples and use cases where I think this is really valuable.

Small Businesses

Suppose you're a small media company. You're creating marketing websites, you're creating WordPress websites, you're creating some kind of content-based website, and you're a small startup with two developers. With `proxy_use_cache_stale`, you can keep spending your time working on new features instead of optimizing queries, assuming that you're running at scale.

If you're not running at scale at all, then your users are going to see slow stuff all the time if you don't optimize every endpoint. But once you start running at scale, your slow endpoints are way, way less important – because you can just cache them. It becomes not so much a matter of what the response time is, as how stale that content can get.

If you design your site based on cacheability from the ground up, you can keep running with minimal resources so that your costs stay low. And if you're in the rare case where you're using your own hardware, stale cache also might be the only way to get higher capacity quickly.

Of course, this is going to be way more valuable for small media companies or sites that aren't primarily working on user-specific content that shouldn't be cached. But even in those other contexts, this can alleviate concerns about a marketing website, or anything else related, that might need to deal with a lot of traffic.

I recommend publishing your site to static pages. But since that pattern isn't as popular, and a lot of well-known CMSes and web frameworks still think it's a good thing to execute DB queries on every request, this method can protect you from that design pattern

with very little effort.

PBS

At PBS, during the Downton Abbey season, our traffic would increase about fivefold depending on the service. Because we usually use this feature, we had to scale up far less than we would otherwise, and we were able to retain traffic from spikes that we had lost during previous seasons before we used NGINX.

Incidentally, whenever I've seen people not capturing all the traffic they could, if you're looking at Google Analytics, you'll see a parabolic upside-down U shape, or you'll see a plateau, or an even curve like that. That means that you're not capturing all the traffic you could be. You're only always capturing all the traffic if you're seeing spikes and up-and-down zigzags. When you see that smooth plateau or parabola you know it's not real; you're losing something.

The `error` and `timeout` [parameters to] the `proxy_cache_use_stale` directive also came in handy on more than one occasion, and kept users from experiencing downtime.

We manage so many different sites and API services at PBS that reducing the amount of traffic to our application servers implies real cost savings. This would also be true for any kind of medium to large company. Once you have 15 different services, 5 APIs and 3 major websites, and a mobile platform – when you're delivering this kind of stuff, it implies big cost savings, because the number of backend servers needed is drastically increased.

You do have to design up front for this. But if you do that, since you have fewer application servers, you can save money across the board. And it's not really that difficult to do this.

Another thing is, at PBS, we don't own all the code we run. When someone hands us a site they built for a producer or a station, we

often have no control of the underlying software stack in use, or the quality of the code.

For my internal APIs, stuff that PBS is building in house, I can say “don’t give me any responses that are more than 200 milliseconds” [and] “Don’t build your service like that”. And if they build me some bad service, I can tell them to fix their code. But since we have a lot of this code that we didn’t build in house, I can’t put this requirement on it.

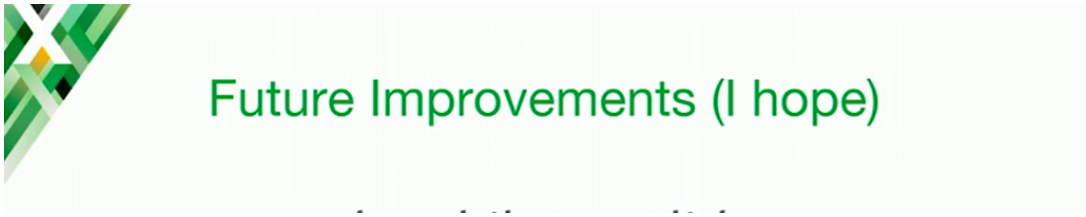
We often have no control of the underlying software stack in use, or the quality of the code. Without some of these tricks and minor modifications, those sites couldn’t scale.

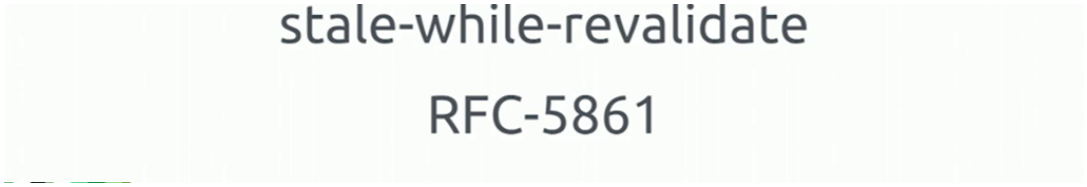
Unpredictable Events

We have a lot of PBS stations, and if some sort of unpredictable event occurs in the area, that station might get a ton of traffic. This happens a lot with shootings, because sometimes the only place that people are getting their news is a small PBS station. And that’s the only one that has the story about that incident that happened in that place. Then, all of a sudden, this tiny station that never gets any traffic just gets hammered and hammered. So, how do you deal with that?

Sometimes, I’ve been there to help to beforehand, but it’s not like every station’s development team has come to me and said, “Hey, help me figure this out” up front. But if they use one of our platforms, we’re often able to scale their site without modification, even though they’re seeing traffic way beyond their typical pattern.

30:57 Future Improvements





Okay, so I love using stale cache, but a feature improvement I'd like to see in NGINX is support for "stale-while-revalidate". And both Squid and Varnish have this option. I noticed that Fastly also has the option. Stale-while-revalidate is the same phenomenon, but instead of one slow request, zero slow requests. So, the server still has to give you back a good response that can go into the cache, but the user doesn't face that one slow response.

So then, you could have the worst possible application and just be like, "Whatever". And you're just running. So, you know, I like people not to have to waste effort on things if possible. And this gets people able to focus on the things that they really need to do.

So, I think that's all I had. I'm hoping that these guys get sstale-while-revalidate. And last year, I talked to [NGINX, Inc.] about it and it seemed like … they'd have to change something really significant to get this in NGINX.

I don't know what it was, like they have to rewrite some piece of code that was really essential. It might be difficult to rewrite. And that's why it hasn't shown up sooner as far as I know.

Editor – The stale-while-revalidate feature was implemented in NGINX Open Source 1.11.10 and NGINX Plus R12.

Recommended Reading

- Code and configuration examples from the presentation
- A Guide to Caching with NGINX and NGINX Plus
- NGINX Content Caching
- The Benefits of Microcaching with NGINX