Intelligent Caching

Leveraging Cache to Scale at the Frontend

Tom Barker
Learn from experts.
Find the answers you need.

Sign up for a 10-day free trial to get unlimited access to all of the content on Safari, including Learning Paths, interactive tutorials, and curated playlists that draw from thousands of ebooks and training videos on a wide range of topics, including data, design, DevOps, management, business—and much more.

Start your free trial at:
oreilly.com/safari

(No credit card required.)
Short. Smart. Seriously useful.

Free ebooks and reports from O’Reilly at oreil.ly/ops-perf

HTTP/2
A New Excerpt from High Performance Browser Networking

Ilya Grigorik

Get even more insights from industry experts and stay current with the latest developments in web operations, DevOps, and web performance with free ebooks and reports from O’Reilly.
Intelligent Caching
Leveraging Cache to Scale at the Frontend

Tom Barker
Preface. ................................................................. v

1. Utilizing Cache to Offload Scale to the Frontend. ................. 1
   What Is Cache? 1
   Setting Cache 2
   Summary 10

2. Leveraging a CDN. ......................................................... 11
   Edge Caching 11
   Quantifying the Theory 14
   CDN Offload 15
   Summary 15

3. Intentional Cache Rules. .................................................... 17
   Hot, Warm, and Cold Cache 17
   Static Content 18
   Personalized Content 19
   Summary 21

4. Common Problems. .......................................................... 23
   Problem: Bad Response Cached 23
   Problem: Storing Private Content in Shared Cache 25
   Problem: GTM Is Ping-Ponging Between Data Centers 26
   Solution: Invalidating Your Cache 28
   Summary 29
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5. Getting Started</td>
<td>31</td>
</tr>
<tr>
<td>Evaluate Your Architecture</td>
<td>31</td>
</tr>
<tr>
<td>Cache Your Static Content</td>
<td>31</td>
</tr>
<tr>
<td>Evaluate a CDN</td>
<td>32</td>
</tr>
<tr>
<td>Summary</td>
<td>33</td>
</tr>
</tbody>
</table>
The idea for this book started when I came to understand how hard it is to hire engineers and technical leaders to work at scale. By scale I mean having tens of millions of users and hundreds of millions of requests hitting your site. Before I started working on properties on the national stage, these would have been DDOS numbers. At these numbers, HTTP requests start stacking up, and users start getting turned away. At these numbers, objects start to accumulate, and the heap starts to run out of memory in minutes. At these numbers, even just logging can cause your machines to run out of file handles.

Unless you are working or have worked at this scale, you haven’t run into the issues and scenarios that come up when running a web application nationally or globally. To compound the issue, no one was talking about these specific issues; or if they were, they were focusing on different aspects of the problem. Things like scaling at the backend, resiliency, and virtual machine (VM) tuning are all important topics and get the lion’s share of the coverage. Very few people are actually talking about utilizing cache tiers to scale at the frontend. It was just a learned skill for those of us that had been living and breathing it, which meant it was hard to find that skill in the general population.

So I set about writing a book that I wish I had when I started working on my projects. As such the goal of this book is not to be inclusive of all facets of the industry, web development, the HTTP specification, or CDN capabilities. It is to simply to share my own learnings and experience on this subject, maybe writing to prepare a future teammate.
What this book is:

- A discussion about the principals of scaling on the frontend
- An introduction to high-level concepts around cache and utilizing cache to add a buffer to protect your infrastructure from enormous scale
- A primer on benefits of adding a CDN to your frontend scaling strategy
- A reflection of my own experiences, both the benefits that I’ve seen, and issues that I have run into and how I dealt with them

What this book is not:

- An exhaustive look at all caching strategies
- An in-depth review of CDN capabilities
- A representation of every viewpoint in the field

I hope that my experiences are useful and that you are able to learn something and maybe even bring new strategies to your day-to-day problems.
Since 2008 I have run, among other things, a site that handles around 500 million page views per month, hundreds of transactions per second, and is on the Alexa Top 50 Sites for the US. I’ve learned how to scale for that level of traffic without incurring a huge infrastructure and operating cost while still maintaining world-class availability. I do this with a small staff that handles new features, in addition to a handful of virtual machines.

When we talk about scalability, we are often talking about capacity planning and being able to handle serving requests to an increasing amount of traffic. We look at things like CPU cycles, thread counts, and HTTP requests. And those are all very important data points to measure and monitor and plan around, and there are plenty of books and articles that talk about that. But just as often there is an aspect of scalability that is not talked about at all, that is offloading your scaling to the frontend. In this chapter we look at what cache is, how to set cache, and the different types of cache.

What Is Cache?

Cache is a mechanism to store data as responses to future requests to prevent the need to look up and retrieve that data again. When talking about web cache, it is literally the body of a given HTTP response that is indexed and retrieved using a cache key, which is the HTTP method and URI of the request.
Moving your scaling to the frontend allows you to serve content faster, incur far fewer origin hits (thus needing less backend infrastructure to maintain), and even have a higher level of availability. The most important concept involved in scaling at the frontend is intentional and intelligent use of cache.

**Setting Cache**

Leveraging cache is as easy as specifying the appropriate headers in the HTTP response. Let’s take a look at what that means.

When you open up your browser and type in the address of a website the browser makes an HTTP request for the resource to the remote host. This request looks something like this:

```plaintext
GET /assets/app.js HTTP/1.1
Host: []
Connection: keep-alive
Cache-Control: max-age=0
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_4)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/51.0.2704.84
Safari/537.36
Accept: */*
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
If-Modified-Since: Thu, 09 Jun 2016 02:49:35 GMT
```

The first line of the request specifies the HTTP method (in this case **GET**), the URI of the requested resource, and the protocol. The remainder of the lines specify the HTTP request headers that outline all kinds of useful information about the client making the request, like what the browser/OS combination is, what the language preference is, etc.

The web server in turn will issue an HTTP response, and in this scenario, that is what is really interesting to us. The HTTP response will look something like this:

```plaintext
HTTP/1.1 200 OK
Date: Sat, 11 Jun 2016 02:08:40 GMT
Server: Apache
Cache-Control: max-age=10800, public, must-revalidate
Connection: Keep-Alive
Keep-Alive: timeout=15, max=98
ETag: "c7c-2268d-534c78e98dc0"
```

The first line of the HTTP response specifies the protocol and the status code. Generally you will see either a **200 OK** for cache misses,
a 304 Not Modified with an empty body for cache hits, or a 200 (from cache) for content served from browser cache.

The remainder of the lines are the HTTP response headers that detail specific data for that response.

**Cache-Control**

The most important header for caching is the Cache-Control header. It accepts a comma-delimited string that outlines the specific rules, called directives, for caching a particular piece of content that must be honored by all caching layers in the transaction. The following are some of the supported cache response directives that are outlined in the HTTP 1.1 specification:

*public*

This indicates that the response is safe to cache, by any cache, and is shareable between requests. I would set most shared CSS, JavaScript libraries, or images to public.

*private*

This indicates that the response is only safe to cache at the client, and not at a proxy, and should not be part of a shared cache. I would set personalized content to private, like an API call that returns a user’s shopping cart.

*no-cache*

This says that the response should not be cached by any cache layer.

*no-store*

This is for content that legally cannot be stored on any other machine, like a DRM license or a user’s personal or financial information.

*no-transform*

Some CDNs have features that will transform images at the edge for performance gains, but setting the no-transform directive will tell the cache layer to not alter or transform the response in any way.

*must-revalidate*

This informs the cache layer that it must revalidate the content after it has reached its expiration date.
**proxy-revalidate**
This directive is the same as must-revalidate, except it applies only to proxy caches; browser cache can ignore this.

**max-age**
This specifies the maximum age of the response in seconds.

**s-maxage**
This directive is for shared caches and will override the max-age directive.

**ETag**
The ETag header, short for *entity tag*, is a unique identifier that the server assigns to a resource. It is an *opaque identifier*, meaning that it is designed to leak no information about what it represents.

When the server responds with an ETag, that ETag is saved by the client and used for conditional GET requests using the If-None-Match HTTP request header. If the ETag matches, then the server responds with a 304 Not Modified status code instead of a 200 OK to let the client know that the cached version of the resource is OK to use.

See the waterfall chart in **Figure 1-1** and note the Status column. This shows the HTTP response status code.
Vary

The Vary header tells the server what additional request headers to take into consideration when constructing the response. This is useful when specifying cache rules for content that might have the same URI but differs based on user agent or accept-language.

Legacy response headers

The Pragma and Expires headers are two that were part of the HTTP 1.0 standard. But Pragma has since been replaced in HTTP 1.1 by Cache-Control. Even still, conventional wisdom says that it’s important to continue to include them for backward compatibility with HTTP 1.0 caches. What I have found is that applications built when HTTP 1.0 was the standard—legacy middleware tiers, APIs, and even proxies—look for these headers and if they are not present do not know how to handle caching.
I personally ran into this with one of my own middleware tiers that I had inherited at some point in the past. We were building new components and found during load testing that nothing in the new section we were making was being cached. It took us a while to realize that the internal logic of the code was looking for the `Expires` header.

Pragma was designed to allow cache directives, much like Cache-Control now does, but has since been deprecated to mainly only specify no-cache.

`Expires` specifies a date/time value that indicates the freshness lifetime of a resource. After that date the resource is considered stale. In HTTP 1.1 the `max-age` and `s-maxage` directives replaced the `Expires` header. See Figure 1-2 to compare a cache miss versus a cache hit.
Figure 1-2. Sequence diagram showing the inherent efficiencies of a cached response versus a cache miss.
Tiers of Cache

As web developers, we can leverage three main types of cache defined by where along the flow the cache is set:

- Browser cache
- Proxy cache
- Application cache

See Figure 1-3 for a diagram of each kind of cache, including a cache miss.

![Diagram of different tiers of cache](image)

**Figure 1-3. A request traversing different tiers of cache**

**Browser cache**

*Browser cache* is the fastest cache to retrieve and easiest cache to use. But it is also the one that we have the least amount of control over. Specifically we can’t invalidate browser cache on demand; our users have to clear their own cache. Also certain browsers may choose to ignore rules that specify not to cache content, in favor of their own strategies for offline browsing.
With browser cache the web browser takes the response from the web server, reads the cache control rules, and stores the response on the user’s computer. Then for subsequent requests the browser does not need to go to the web server, it simply pulls the content from the local copy.

As an end user, you can see your browser’s cache and cache settings by typing `about:cache` in the location bar. Note this works for most browsers that are not Internet Explorer.

To leverage browser cache, all we need to do is properly set our cache control rules for the content that we want cached.

See Figure 1-4 for how Firefox shows its browser cache stored on disk in its `about:cache` screen.

![Figure 1-4. Disk cache in Firefox’s about:cache screen](image)

**Proxy cache**

Proxy cache is leveraging an intermediate tier to serve as a cache layer. Requests for content will hit this cache layer and be served cached content rather than ever getting to your origin servers.

In Chapter 2 we will discuss combining this concept with a CDN partner to serve edge cache.

**Application cache**

Application cache is where you implement a cache layer, like memcached, in your application or available to your application that
allows you to store API or database calls so that the data from those calls is available without having to make the same calls over and over again. This is generally implemented at the server side and will make your web server respond to requests faster because it doesn’t have to wait for upstream to respond with data.

See Figure 1-5 for a screenshot of https://memcached.org.

![Figure 1-5. Homepage for memcached.org](https://memcached.org)

**Summary**

Scaling at the backend involves allocating enough physical machines, virtual machines, or just resources to handle large amounts of traffic. This generally means that you have a large infrastructure to monitor and maintain. A node is down, gets introduced to the load balancer, and is seen by the end user as an intermittent error, impacting your site-availability metrics.

But when you leverage scale at the frontend, you need a drastically smaller infrastructure because far fewer hits are making it to your origin.
Browser cache is a great tool, and I would say table stakes for starting to create a frontend, scalable site. But when your traffic and performance goals demand more, it is usually time to step up to partnering with a content delivery network (CDN). This chapter we look at leveraging a CDN to both improve your performance and offload the number of requests via proxy caching at the edge, called edge caching.

The purpose of a CDN is to provide availability and performance for content served over the Internet. There are several ways that this is accomplished, from providing Global Traffic Management (GTM) services to route content to the closest or fastest data center, to providing edge serving.

**Edge Caching**

Edge serving is where a CDN will provide a network of geographically distributed servers that in theory will reduce time to load by moving the serving of the content closer to the end user. This is called edge serving, because the serving of the content has been pushed to the edge of the networks, and the servers that serve the content are sometimes called edge nodes.

To visualize the benefits of edge computing, picture a user who lives in Texas trying to access your content. Now you don’t yet use a CDN, and all of your content is hosted in your data center in Nevada. In order for your content to reach your user, it must travel
across numerous hops, with each hop adding tens or even hundreds of milliseconds of latency.

See Figure 2-1 for a request made in Texas traversing a series of hops back and forth to a data center in Nevada.

**Figure 2-1. Content hosted in a data center in Nevada served to an end user in Texas traversing seven theoretical hops**

Now say you are leveraging a CDN that has edge nodes across the country. Your origin servers are still your data center, but mirrors of your content are stored on your CDN partner’s hundreds or thousands of edge nodes, and there is an edge node in the same area as your end user. You have eliminated all of the hops and any latency that they would bring. Even if you think of the packets as light traveling down fiber optics, if light has less distance to travel, it will reach the end user faster.

See Figure 2-2 for this same request served from the edge.
Now that your content is served at the edge, make sure your cache rules for your content are set correctly, using the Cache-Control and ETag headers that we discussed in Chapter 1. Suddenly you have edge caching. Note that in addition to honoring your origin cache settings, your CDN may apply default cache settings for you. When you combine the benefits of both GTM and edge caching, you drastically increase your potential uptime.

**Last Known Good**

Picture the following scenario: you have two or more data centers that host the content that your CDN propagated to its edge nodes. You experience a catastrophic failure at one of the data centers. Your CDN notices that your origin is not responding, so it shifts all incoming traffic to your good data center. If your last data center then goes down, the CDN also caches the last successful response (sometimes referred to as last known good) for each resource at the edge, so your end users never experience an outage as long as your resource's cache lives.
Quantifying the Theory

I leverage edge caching for most of the sites that I run, so I thought it would be fun to quantify this theory with real, live content. To do this, I opened my command line and ran ping against a subset of content on each of my tiers: my data center origins, my origins utilizing just GTM, and finally my content served from the CDN edge cache.

What I found is that over the course of many tests, serving content via a CDN was 45% faster compared to serving it from the data center directly. Even better, there was a 67% performance improvement when serving content from our CDN’s edge cache versus serving content directly from our data center origins. See the data from my experiments in Figure 2-3.

Figure 2-3. Bar chart comparing response times for the same piece of content served from a data center origin server, from behind a CDN Global Traffic Manager, and from a proxy cache at the edge
**CDN Offload**

Besides speeding up delivery of the content to the end user, another big benefit of using a CDN’s edge cache is offloading traffic. Sometimes called *CDN Cache Hit Ratio*, this is the amount of traffic, both total bandwidth and sheer number of transactions, that can be handled by the cache nodes versus the number that gets passed back to your origin servers.

The ratio is calculated by dividing the total number of requests that are reported by the CDN by the total number of offloaded or cached responses over a time period, so:

$$ CDN \text{ offload} = \frac{\text{offloaded responses}}{\text{total requests}} $$

Think about it this way: say you get in 25,920,000 requests per day, and your CDN offload is 95%. This means that the CDN would absorb 23,328,000 requests, and your origins would only need to handle 2,592,000 requests. In other words, you’re going from 300 requests per second down to only 30 requests per second, thereby drastically reducing the amount of infrastructure you would need at your origin.

**Summary**

Content delivery networks are great tools and an integral part of your client-side scaling strategy. By leveraging proxy caching on an edge network, we can serve content faster to our end users and have a more efficient infrastructure because the CDN is absorbing a percentage of the incoming requests.
In this chapter we will look at different types of content we can cache, but first let's start by quickly covering some concepts that apply to what we will be talking about:

- Hot, warm, and cold cache
- Cache freshness

**Hot, Warm, and Cold Cache**

There are several states cache can be in:

*Cold cache*

A cold cache is empty and results in mostly cache misses.

*Warm cache*

The cache has started receiving requests and has begun retrieving objects and filling itself up.

*Hot cache*

All cacheable objects are retrieved, stored, and up to date.

Cache starts cold, either with no objects stored or with any that are stored stale. Then over time, and as requests start coming in, the server retrieves the objects and fills the cache. A hot cache will start to cool and turn cold as time goes by and the content starts to expire.
Cache Freshness

A cold cache can also be the result of expired content, based on the \texttt{max-age} directive specified. A cache is considered fresh if it’s creation date is within the \texttt{max-age}. That window is its \textit{time to live} (sometimes called TTL). See Figure 3-1 for a pattern that describes the warming and cooling of cache.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{patternสื่อ{Cache Warming and Cooling}}
\caption{Cold cache warming and cooling over time and requests}
\end{figure}

Now that we understand what the concept of cache is, as well as the types of caches and the benefits that we can get from using them, let’s talk about the types of content we may have and how we can think about applying cache rules to them.

Static Content

The most obvious thing to cache is \textit{static content} because static content is shared across users and doesn’t change often. Things like fonts, images, and CSS and JavaScript files that are shared and are not going to be updated frequently are your low-hanging fruit. Go through and adjust their cache control rules for an immediate and noticeable bang for your buck.
Shared static content is especially effective because the first person to access that content warms the cache, and every subsequent hit after that, for every user, is going to be a cache hit.

The most important thing to think of when caching your content is how frequently that content is updated. For example, one of sites that I’ve been overseeing for a long while now is a web portal where editorial staff, via a CMS, update content various times throughout the day. The content is stored as flat files that get fingerprinted and stored on a content server.

The content flat files are cached heavily, from 30 days up to a year. This is because every time a new file is produced, it has a unique URL so even if the editors needed to make a change to an existing story, they would output a new file with its own cache rules. The base page that reads in the flat files, on the other hand, is only cached for five minutes (see Figure 3-2).

![Image](image.png)

*Figure 3-2. An editor creates content that has a different TTL than the page that reads it in, allowing for new content to be created and loaded on to the page quickly if need be, or be long lived if there are no updates.*

**Personalized Content**

The more challenging scenario is how to apply the concepts of client-side scaling to sites that are primarily made up of personalized content. By definition personalized content is created for the specific user that is logged in. Think of Amazon.com’s Your Orders page that shows all of your recent orders, and their real-time delivery status (see Figure 3-3).
In the past this has been accomplished by gathering all of the personalized content on the server side, and then once everything is assembled, only then returning and presenting that page to the end user. There may be reasons for this approach, but the end result is that this ends up with a worse perception of speed for the end user.

When my team and I have been presented with this challenge, our solution was instead to keep with our philosophy of scaling at the frontend. To do this we have actually created a design pattern where we split the frontend from the middleware, each being their own applications with their own build and deploy processes, and their own node clusters.

In the frontend application, we store all of the mostly unchanging shared static content—think JavaScript libraries, images, and CSS files—and we set very long TTLs for this. The frontend then calls the middleware via XHR requests and populates the current page and proactively retrieves the data for the subsequent pages.

The middleware, on the other hand, required us to be very careful about the cache rules. This is where users would take action on their account and expect to see the results immediately.
In my example we had the luxury of having a previous version of the site live for years prior, so we could look at data of past usage and see things like how long were sessions, how frequently did users return to a page, and how long did they stay on pages.

If you don't have this luxury, you can conduct a user study to see how your test group uses your application. If you aren't familiar with user studies, Travis Lowdermilk has written extensively about them in his book *User-Centered Design* (O'Reilly). If all else fails, just take educated guesses. Either way you will be analyzing your usage data once you are live and can adjust and course correct as necessary.

So based on our established usage data, some API calls we cached for one minute, some for five minutes, and some even longer.

**Summary**

Caching is not just for static content that rarely ever gets updated. All sorts of content can benefit from caching, even if the TTL is small. It is worth performing an analysis of frequency of usage and updates and setting cache rules based on that, so that even personalized data can get the performance boost of caching.

But no matter a piece of content's TTL, its cache will still likely experience a sine wave of warming and cooling based on usage and how long the content will be fresh. On creation, the content cache starts out cold, warms up as users begin requesting it, then cools down as the requests slow down and time passes.
While caching is a great approach to addressing scale and performance, it comes with its own unique set of issues that could arise. In this chapter we look at some of the most prevalent issues, talk about the root causes, both technical and social, and finally present solutions to these problems.

These are issues that I have personally run into and seen over and over again with different teams and products.

**Problem: Bad Response Cached**

Picture this: your site is up, the cache is hot, performance is great, and traffic is soaring. You open your browser, and the site comes up, because of course you’ve made it your homepage, and you immediately notice that instead of content, your site is serving up an error message.

No alarms have gone off, your availability report says everything is great, your application performance monitoring (APM) software is reporting no errors from your stack (only a small blip from one of your API dependencies a half hour ago).

So what in the world is going on? What went wrong, why is it showing no evidence, and how do you recover from this?
**Cause**

The most likely scenario is this: your site called an API or your own middleware tier to load content. This content is usually long lived, so your TTL is set for seven days.

The only problem is, that small blip a half an hour ago was the dependency throwing an error. It wasn't an HTTP 500 error, but instead an upstream dependency to your API that threw a 500. Your API decided to return an HTTP 200 response but surface the error from its upstream dependency in the response body.

The issue is that HTTP error response codes are not cacheable to protect just this scenario. By wrapping the error response in an HTTP 200, the API has made the error cacheable.

What this means is that your cache layer stored this bad response from your API, and your application has loaded that in and is storing it in cache for the next seven days. All because the HTTP status codes were not being honored in the dependency chain of your application (see Figure 4-1).

![Figure 4-1. A sequence diagram that outlines how a bad response can be cached if the API returning the response does not also surface the error in the HTTP status code that it returns](image-url)
So the clock is ticking, users are being presented with this error, and your site is essentially down. The calls are starting to come in. Let’s look at how to avoid this situation in the first place.

**How to Avoid This**

It may seem obvious, but the way to avoid this in the first place is to have a conversation with your partner teams. Review their documentation, and how they handle errors, and organize war games where you simulate outages up and down the dependency chain.

At the very least this will make you aware that this situation could occur, but ideally you will be able to convince or influence your partner teams to respect and surface the HTTP status codes of their upstream services. Anecdotally this is the most prevalent issue I’ve run into with service providers, and it’s usually either just a philosophical difference of opinion (why would I return an error; my service isn’t erroring out) or because the team has been incentivized through goals or an SLA to avoid anything resembling an error. These are all solved through communication and compromise.

**Problem: Storing Private Content in Shared Cache**

Here is another scenario: imagine you have a site that calls an API that loads in user data via an API call. This user data is presented to the user on your homepage, maybe innocuously as a welcome message, or more seriously as account, billing, or order information.

**Cause**

This is private data, intended for the specific user. But you just deployed an update and didn't realize that your CDN by default applies a 15-minute cache to all content unless otherwise specified, including the new end point you just made to make the user data call.

A few minutes after you deploy the new end point, the calls start coming in, with customers complaining that they are seeing the wrong user information. The data from the call is being cached, so users are seeing previous users’ session information (see Figure 4-2).
How to Avoid This

There are two big things you can do to avoid this issue:

1. **Education**: educate everyone on your team about how your CDN works, how cache works, and how important customer privacy is.

2. **Test in your CDN**: chances are your lower environments are not in your CDN, which means that they don’t necessarily have the same cache rules, which then means that your testing will miss this test case completely. You can solve for this by having either a testing environment or ideally a dark portion of your production nodes in your CDN to test on.

Problem: GTM Is Ping-Ponging Between Data Centers

You’ve offloaded your scale to the frontend, and you are maintaining a lean, minimal backend infrastructure. Everything is lovely, deploy-
ments are fast because files don't have to be cascaded across hundreds of machines, your operating costs are low, performance is beautiful, and all of your world is in harmony.

Except your application performance monitoring software has started sending throughput alerts. You take a look and see that your throughput is spiking high in one cluster of nodes for a little while, then drops to nothing, then the same happens in a different cluster of nodes. These groupings of nodes correspond to data center allocation. Essentially the Global Traffic Management service in your CDN is ping-ponging traffic between data centers.

You look at your nodes and they range from being up, down, starting back up, and having HTTP requests spiking.

**Cause**

What's happened is that you have either deployed a change or made a configuration change that accidentally turned off caching for certain assets. In addition you have made your infrastructure too lean, so that it can't withstand your full traffic for any significant amount of time.

The sheer number of requests coming in has caused your HTTP request pool to fill up, requests are stacking up, and your responses are drastically slowing down. Some machines are even running out of memory and are being unresponsive. Your CDN's GTM service notices that the health checks for a given data center are not loading, and it sees your data center as being down, so it begins to direct all incoming requests to your other data center(s). This exacerbates the problem, and soon those machines are becoming unresponsive. The GTM service looks for a better data center to point traffic to, and it just so happens that was enough time for some of the requests in your downed data center to drain off. Therefore, it has started to become responsive again, so the GTM begins to point traffic there.

And the cycle continues.

**How to Avoid This**

Once again, education and awareness of how cache works and impacts your site should head off the change that started this in the first place. But there is a more insidious problem present: your infrastructure is too lean.
Capacity testing and planning ahead of time without cache turned on would have let you know your upper limits of scale.

**Solution: Invalidating Your Cache**

Once you are in the middle of most of the given scenarios, your path forward is to fix the underlying issue, then invalidate your cache.

![A Word of Caution]

Invalidating your cache forces your cache layer to refresh itself, so it is imperative to first make sure that the underlying issue is fixed.

In the case of the API returning an HTTP 200 with a body that contains an error, either get the team to surface the HTTP response of the upstream service or just don’t cache that API call. For the cached user data call, either don’t cache that call, or add a fingerprint to the URL.

**Fingerprint the URLs**

Fingerprinting the URL involves adding a unique ID to a URL. This can be done several ways for different reasons:

*Build life*

As part of your application build-and-deploy process, you can have your build life added to the path to your static files as well as the links to them in your pages. This creates a brand new cache key and guarantees that previous versions are no longer referenced even in a user’s local browser cache.

*Hashed user ID*

In the case of private data API calls, you may still want to cache the call. In that case you can add an ID that is unique to the user as part of the user call.

**Kill Switch**

When all else fails your CDN should provide you with a way to purge your edge cache. This is will systematically remove one or more items from cache. This is the nuclear option to the situation
because it will cause a cache refresh to your origins for all incoming requests for that item(s).

**Summary**

We have just talked about a number of potential pitfalls you might run into, and all were either due to misconfigured cache or from not following the HTTP specification. The key to recovering from these issues is having a way to invalidate cache.

Armed with the information in this chapter, from identifying issues to their fixes, you should be prepared to handle these issues when they arise in your own environments.
Now that you understand the concepts, how do you apply them to your own site? In this chapter we look at some beginning tactics for implementing the strategies we’ve covered in this book.

**Evaluate Your Architecture**

Look at your application architecture and note which direction your architectural philosophy is leaning. Are you backend heavy? Does every click on your page result in a page refresh and HTTP round-trip?

If so you’ll get minimal benefits scaling to the frontend as is. Your first step would be to shift to a more modern, web-friendly architecture where you can take advantage of innovations that have taken place over the last five to eight years, like asynchronous loading of content and RESTful APIs.

This will allow you to make your frontend highly cacheable, and you only would need to make round trips to call APIs asynchronously without refreshing your page and interrupting the experience.

**Cache Your Static Content**

Look at each page or section of your site with the Network tab open in Developer Tools. Look at the status codes being returned. Do you see any 304s? Why not?
You can also run your site through most popular web performance evaluation tools. Webpagetest.org is a personal favorite of mine. Along with a performance evaluation, it will explicitly call out assets by URI that are not being cached (see Figure 5-1).

Leverage browser caching of static assets: 34/100

Evaluate a CDN

If you aren’t using a CDN yet, most offer a free trial account so that you can try out their services and see the benefits for yourself. All you would need to do is sign up for the free account, deploy your application to the CDN, then either just:

- Traffic shape a segment of your traffic at the CDN hosted application.
- Add the CDN as of one of your data centers in your round-robin rotation.
- Or just go all in and point all of your traffic at the CDN.

Once you have traffic pointing to your CDN, compare your data. Look at your web performance, and look at the CPU usage for your origins with and without the CDN.
Summary

At this point you should have all the tools you need to get started implementing client-side scaling tactics for your own site. Every architecture and business case is different, so be sure to evaluate your own needs and see what works and what doesn’t work for you. As with everything in life, take what you like and leave the rest.
About the Author

**Tom Barker** is a software engineer, an engineering manager, a professor, and an author. Currently he is Director of Software Engineering and Development at Comcast and an Adjunct Professor at Philadelphia University.