8/31/20

1. API key vs token
2. Http Header’s fields
3. How to create json file, and how to parse json file.
4. Sniff HTTP requests:

* <https://www.telerik.com/fiddler>

1. **HTTP Sniffer and Protocol Analyzer**
2. HTTP cache (browser vs server)
3. REST API cache: Everything needs to be done on client side since no browser help for client request.

My son was using epic through school code scu9955. Since Epic school puts a limit on how much time my son can read with the free school version, it offers an annual subscription for $71.99. I made the subscription. But I just realized I can't manage my subscription since I got in the epic through school code.

Please either cancel my subscription or offer a way to manage my account.

Thanks,

Jian

1. How files are uploaded in http

<form enctype="multipart/form-data" action="http://localhost:3000/upload?upload\_progress\_id=12344" method="POST">

<input type="hidden" **name="MAX\_FILE\_SIZE" value="100000**" />

Choose a file to upload: <input name="uploadedfile" type="file" /><br />

<input type="submit" value="Upload File" />

</form>

Instead of URL encoding the form parameters, the form parameters (including the file data) are sent as sections in a multipart document **in the body of the request**.

In the example above, you can see the input MAX\_FILE\_SIZE with the value set in the form, as well as a section containing the file data. The file name is part of the **Content-Disposition header**.

**Host**: localhost:3000

**Content-Length**: 1325

Origin: http://localhost:3000

... other headers ...

**Content-Type**: multipart/form-data; boundary=----WebKitFormBoundaryePkpFF7tjBAqx29L

**------WebKitFormBoundaryePkpFF7tjBAqx29L**

Content-Disposition: form-data; name="MAX\_FILE\_SIZE"

100000

**------WebKitFormBoundaryePkpFF7tjBAqx29L**

**Content-Disposition**: form-data; name="uploadedfile"; filename="hello.o"

Content-Type: application/x-object

**... contents of file goes here ...**

------WebKitFormBoundaryePkpFF7tjBAqx29L—

1. **HTTP cache**

* Browser-side cache
* Server-side cache
* HTTP cache needs to solve two main problems:

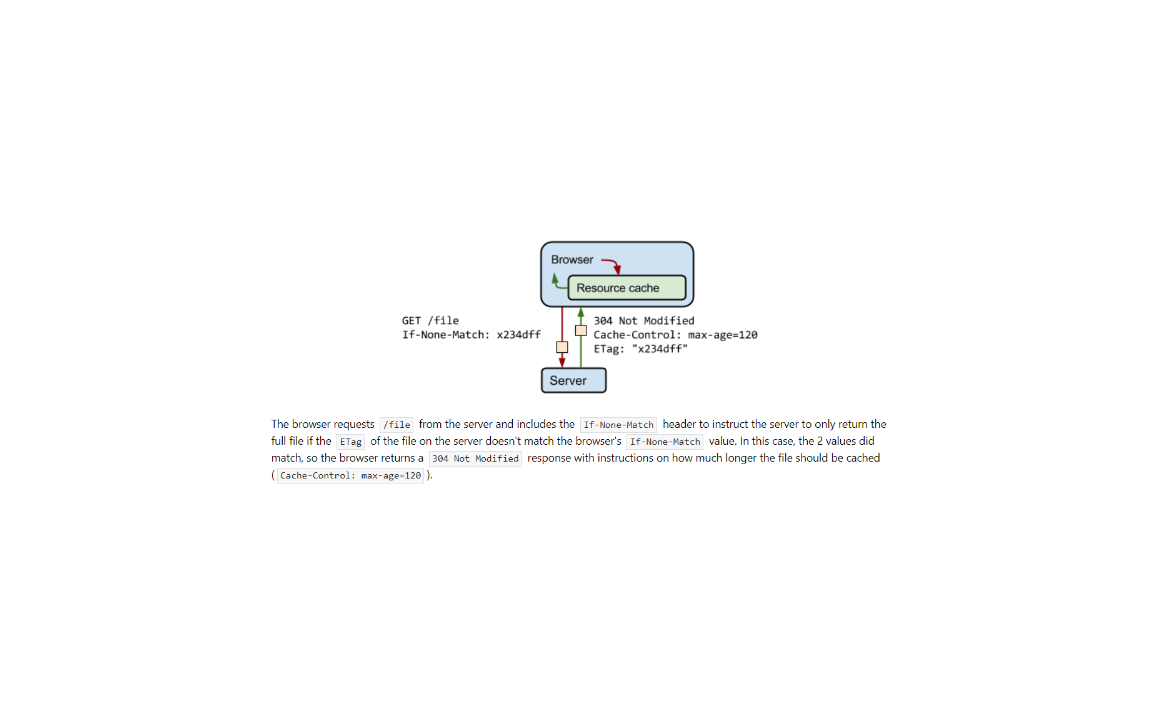
1. Cache or NOT
2. If cache, how long can it be cached.
3. Expires or Cache-Control: max-age

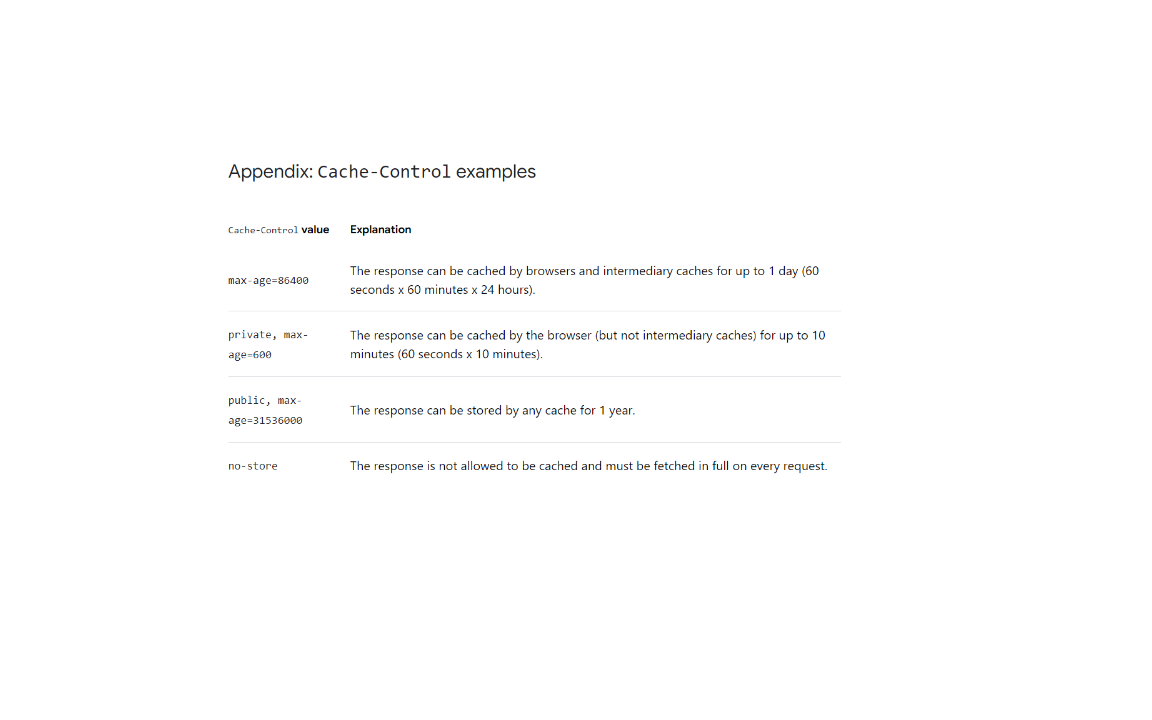
* Try to understand how browser and server manage : ETAG, Expires, max-age, no-cache, and no store. How do server and browser work together?
* Server dictates: cache or no cache; When the cache will expire; When the cache need to be re-validated.

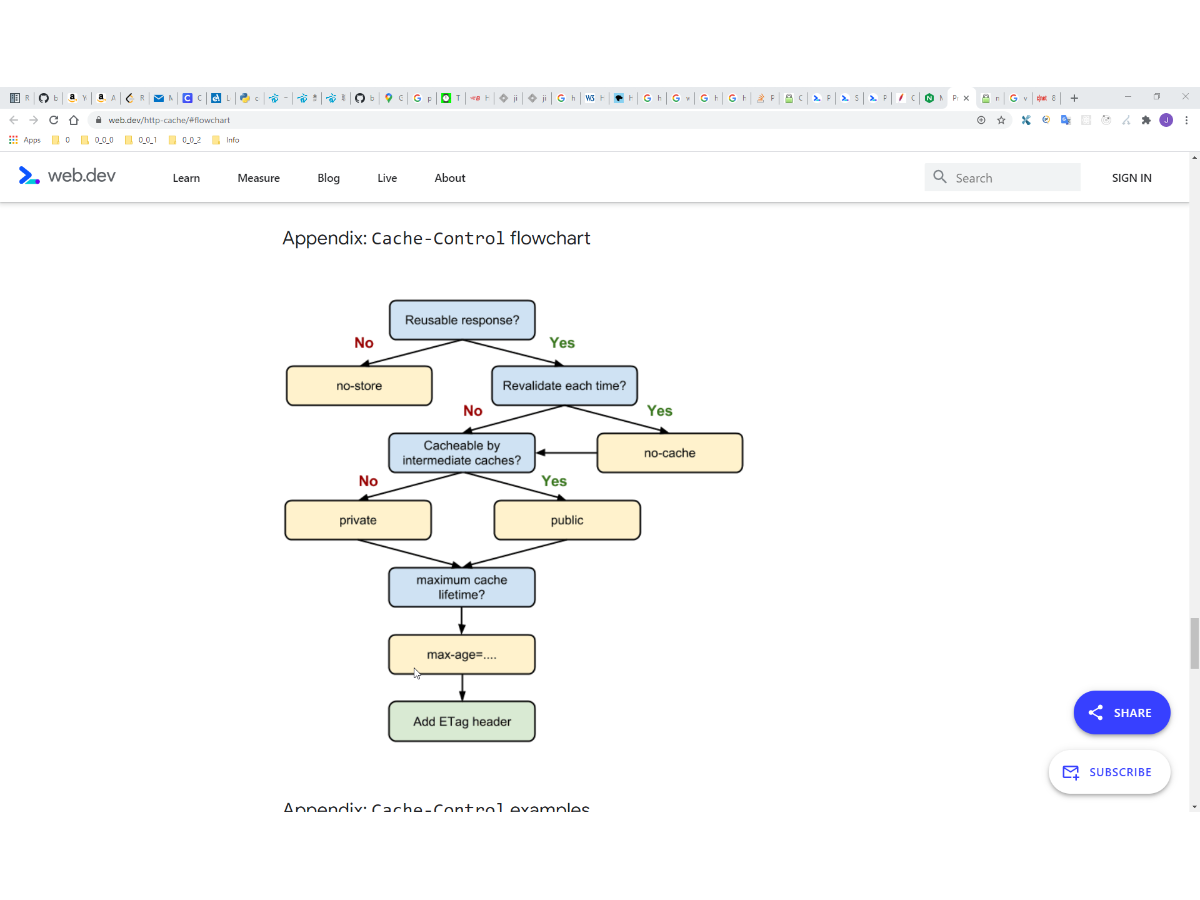
Assume that 120 seconds have passed since the initial fetch and the browser has initiated a new request for the same resource. First, the browser checks the HTTP Cache and finds the previous response. Unfortunately, the browser can't use the previous response because the response has now expired. At this point, the browser could dispatch a new request and fetch the new full response. However, that's inefficient because if the resource hasn't changed, then there's no reason to download the same information that's already in the cache! That's the problem that validation tokens, as specified in the ETag header, are designed to solve. The server generates and returns an arbitrary token, which is typically a hash or some other fingerprint of the contents of the file. The browser doesn't need to know how the fingerprint is generated; it only needs to send it to the server on the next request. If the fingerprint is still the same, then the resource hasn't changed and the browser can skip the download.

By setting ETag or Last-Modified, you'll end up making the revalidation request much more efficient. They end up triggering the [If-Modified-Since](https://developer.mozilla.org/docs/Web/HTTP/Headers/If-Modified-Since) or [If-None-Match](https://developer.mozilla.org/docs/Web/HTTP/Headers/If-None-Match) request headers that were mentioned in [Request headers](https://web.dev/http-cache/#request-headers).

When a properly configured web server sees those incoming request headers, it can confirm whether the version of the resource that the browser already has in its HTTP Cache matches the latest version on the web server. If there's a match, then the server can respond with a [304 Not Modified](https://developer.mozilla.org/docs/Web/HTTP/Status/304) HTTP response, which is the equivalent of "Hey, keep using what you've already got!" There's very little data to transfer when sending this type of response, so it's usually much faster than having to actually send back a copy of the actual resource being requested.







The following Cache-Control configurations are a good start:

* Cache-Control: no-cache for resources that should be revalidated with the server before every use.
* Cache-Control: no-store for resources that should never be cached.
* Cache-Control: max-age=31536000 for versioned resources.

And the ETag or Last-Modified header can help you revalidate expired cache resources more efficiently.