XMLHttpRequest Caching Tests in FireFox

3 October 2008

Bjarne Geir Herland (bjarne@runitsoft.com)

This document contains an analysis of why FireFox fails a number of tests on the web page located at http://www.mnot.net/javascript/xmlhttprequest/cache.html . The approach is reactive, meaning that only tests reported to fail are analysed. In light of some of the findings, a more proactive approach (analysing all tests, also those reported to be successful) might be recommended.

Throughout this document, the term “the test” means tests on the mentioned page, and the term “the server” means the HTTP-server hosting the mentioned page and the server-side scripts called by the test. When running the test, Wireshark was used to capture the entire network-conversation, which is referred to as “the dump”.

The test is run using Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.0.3) Gecko/2008092510 Ubuntu/8.04 (hardy) Firefox/3.0.3 but results are similar when running builds from the latest trunk.

By request, the analysis is inlined in a copy of the test. In order to separate text from the original testpage and the analysis, a different font (the current font) is used.

XMLHttpRequest Caching Tests

nearby: XMLHttpRequest Tests

2006-06-01

This is a set of functional tests for determining how client-side HTTP caches operate.

Note that while they are implemented using XMLHttpRequest (a JavaScript HTTP client), most implementations should use the same cache as for "normal" requests. However, there may be variations.

Each group of tests explains what is being tested and what the implications of failure are. Although many of the tests are automated, some may require user interaction, via a "run test" button. Be sure to follow any *instructions* carefully.

These tests may have unpredictable results if you instructed your browser to force-reload the cache (e.g., by hitting shift-reload or pressing return in the address field), or if there is a proxy cache between your browser and the server. For best results, clear your cache before loading this page.

See this blog entry for more information.

Testing Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.0.3) Gecko/2008092510 Ubuntu/8.04 (hardy) Firefox/3.0.3

Request Cache-Control

Cache-Control headers in HTTP requests allow clients to control how downstream caches operate. In APIs like XMLHttpRequest, they can also be used to control how the local (i.e., in-browser) cache operates.

Request Header Pass-Through

Request cache-control headers can be used by page authors to control the behaviour of the local as well as downstream caches. For example, an author may want to bound the freshness of a response based on the request context, or invalidate the cache from code.

If the browser cache does not pay attention to these directives, it can't be controlled by authors.

1. Cache-Control: no-store request header honoured: fail (Cache-Control: no-store not honoured.)
2. Cache-Control: no-cache request header honoured: fail (Cache-Control: no-cache not honoured.)
3. Pragma: no-cache request header honoured: fail (Pragma: no-cache not honoured.)

In this section, FF fails all three sub-tests. Each sub-test loads a URI three times, setting a request-header (not to be confused with headers in the response) on request#2. The sub-tests all expect to see the server-response from request#2 also for request#3. Thus, the test expects behaviour as outlined in the tables below. The column *Server response* contains the response by the server if the request goes to it and the column *Cacheable* indicates whether such response can be cached. The column *Expected* contains the expected value in the sub-tests.

Request#	Header	Server-resp	Cacheable	Expected	Comment
1	-	1	Yes	-	Initial load
2	**no-store**	2	No	-	Forced reload
3	-	3	Yes	2	Wrong. Should expect 3

This sub-test is wrong in expecting “2” since that value can not be cached due to the no-store directive in request #2.

Request#	Header	Server-resp	Cacheable	Expected	Comment
1	-	1	Yes	-	Initial load
2	**no-cache**	2	Yes	-	Forced reload
3	-	3	Yes	2	Ok – value from cache

This sub-test is ok, but FF does not handle it. The test for *Pragma: no-cache* behaves (and should do so) like this sub-test so I don't repeat it here.

The proper behaviour, according to my understanding, should be like below (again not showing *Pragma: no-cache*) :

Request#	Header	Server-resp	Cacheable	Expected	Comment
1	-	1	Yes	1	Initial load
2	no-store	2	No	2	Forced reload
3	-	3	Yes	3	Prev not cached, loaded
4	no-cache	4	Yes	4	Forced reload
5	-	5	Yes	4	Value from cache

Observe the difference between no-store and no-cache in the *Cacheable* column, and for the subsequent request in the *Expected* column. This behaviour has been implemented in a unit-test.

Conclusion : The first sub-test is wrong, but FF does not handle any of these correctly and should be fixed in any case. It should be noted that I don't find anything in RFC 2616 to back up my claim that **no-store** should force a reload from the server, but the first sub-test would be wrong even if this was not the case.

Related bugs : I have not found any

Automatic Request Cache-Control

Some implementations try to automatically bust upstream caches for XmlHttpRequests, because many people use XHR as an RPC mechanism. However, doing so takes control away from calling code that might want to take advantage of the cache, and reduces cache efficiency.

Implementations that fail this test have automatically appended Cache-Control request headers.

1. Cache-Control request headers automatically appended: success

Validation

Validation is one of the primary caching mechanisms in HTTP; it allows a cache to see if it can reuse an entity it already has, by asking the server if it's still fresh.

Conditional Request Headers

HTTP defines request headers that can be used to make conditional requests, which are used to validate cached representations on the server. Implementations that fail these tests don't send the appropriate request headers that trigger validation.

1. If-Modified-Since request header used: success
2. If-None-Match request header used: success

304 Not Modified Handling

The 304 Not Modified status code indicates that a cached representation is still fresh. Generally, it isn't useful to expose this to authors; a 200 OK response should instead be constructed from the cache.

1. 304 status code automatically handled: success

Freshness

Another key caching technique is allowing the server to specify that a representation is fresh for a given amount of time, so that caches can avoid round-trips to check if it's fresh altogther.

Basic Freshness

Servers can instruct caches to use a stored response without validation. Additionally, clients can use a heuristic (usually based on the Last-Modified header) if no explicit freshness information is present.

Implementations that fail this test do not take advantage of these hints.

1. Heuristic freshness: success
2. Cache-Control max-age response freshness: success
3. Expires response freshness: success

Query Freshness

HTTP requires that URIs with a query string (i.e., those containing a "?") not be cached, unless the server gives explicit freshness information.

1. Heuristic freshness w/ "?": fail (Stale representation used)
2. Cache-Control max-age response freshness w/ "?": success
3. Expires response freshness w/ "?": success

Note that failing the first test is an indication that a freshness heuristic may not be used, which may be desireable behaviour for some applications.

In this section, FF fails the first sub-test (Heuristic freshness w “?”). The sub-test checks that a URI containing a query-string (i.e. a “?”) is not cached UNLESS the server provides “explisit expiration time” (RFC 2616 section 13.9, second paragraph). (The test-page uses the term “explisit freshness information” but this should be equivalent.) The problem is that the dump shows that the server in fact does provides an Expires-header, thus the first sub-test does not check what it claims to be doing.

Conclusion : The first sub-test is wrong. FF behaviour for a working test is unknown.

Related bugs : I have not found any

Redirect Caching

Certain 3xx redirects are allowed to be cached. 301 Moved Permanently is cacheable by default, while 302 Found and 307 Temporary Redirect both need explicit information. 303 See Other is not cacheable.

1. 301 w/ freshness information: fail (301 w/ freshness not cached.)
2. 301 w/o freshness information: fail (301 w/o freshness not cached.)
3. 302 w/ freshness information: fail (302 w/ freshness not cached.)
4. 302 w/o freshness information: success
5. 303 w/ freshness information: success
6. 303 w/o freshness information: success
7. 307 w/ freshness information: fail (307 w/ freshness not cached.)
8. 307 w/o freshness information: success

This section checks caching of certain responses in the 300-series. Each sub-test loads a certain resource twice and on the second load, depending on the sub-test and response-headers, either expects or not to see the cached response. Those sub-tests not expecting to see a cached response (4,5,6 and 8) are marked successful, the others are marked as failed.

The dump shows that the server sends no 30x-responses, only “200 OK” with no freshness information. Thus none of these resources are cached, and the effect that sub-tests not expecting to see a cached response are considered successful can be expected. But this sub-test doesn't really test how FF handles 30x responses.

Conclusion : The sub-test is wrong. FF behaviour for a working test is unknown.

Related bugs : 415086, 341189

Private Caches

HTTP distinguishes between private and public caches; a browser cache is private, and should cache responses marked as such. Implementations that fail this test don't recognise that they're a shared cache, and therefore can't be targetted as one (which is a useful technique for separating browser-

cacheable content from proxy-cacheable content).

1. Private responses cached: success

Directive Precedence

HTTP has many caching directives that might conflict, such as HTTP 1.0's `Pragma` and `Expires`, as well as HTTP 1.1's `Cache-Control: max-age`.

Sometimes, it is useful to direct caches that understand HTTP/1.1 caching directives (like Cache-Control) to do one thing, while directing those that don't to do something else. For example, if you're taking advantage of advanced features, you might want to allow more capable devices to cache something, while making sure that older ones don't.

HTTP accommodates this by specifying that the Cache-Control: max-age directive takes precedence over the Expires header; a response that contains both of them should be cached according to the Cache-Control header by a device that understands it, while those that don't will honour the Expires header.

There are a number of other situations where directives may conflict; in most cases, caches should follow the most conservative directive (i.e., something that says not to cache) present.

Browsers that don't correctly handle directive precedence will make it difficult to target directives at caches with different levels of conformance.

Tests where there are two results test each combination with a different ordering of headers; the results *should* be the same.

1. Cache-Control: max-age and stale Expires is fresh: fail (Fresh representation not used), fail (Fresh representation not used)
2. Cache-Control: max-age and Pragma: no-cache is stale: success, success
3. Cache-Control: no-cache and fresh Expires is stale: success, success
4. Cache-Control: no-store and fresh Expires is not cached: success, success

```
This test checks various precedence between conflicting
freshness-related headers. The failing sub-test if the first one
and it specifically checks that the “Cache-control: max-age”
directive takes precedence over an Expires-header. It performs
the check twice, switching the order of the headers, to ensure
that the order in which the headers come does not make a
difference. FF fails the precedence regardless of the order of
the headers.

Conclusion : FF fails this test

Related bugs : 203271
```

Variant Caching

HTTP allows responses to requests for the same URI to vary based on the values of request headers; this is called server-driven content negotiation. This has special implications for caches, because they usually store

representations based on their URIs, but HTTP 1.1 requires them to also consider the content of the Vary header, which indicates what other headers form part of the cache key.

The first test checks to see that a negotiated response will be cached; the second makes sure that two different variants aren't cached as the same thing. The third test sees if the cache is smart enough to ignore the case of the request header-names specified by the Vary response header. Finally, the fourth test specifically checks to see if responses that vary on the Accept-Encoding header (which commonly implements compression in HTTP) work.

This last test is important, because if compressed content isn't cached by the browser, this popular technique may become less useful.

1. Arbitrary Negotiated response caching: success
2. Arbitrary Negotiated response differentiation: success
3. Arbitrary Negotiated response canonicalisation: success
4. Encoded Negotiated response caching: success

POST Caching

Contrary to conventional wisdom, HTTP does allow caching of the response to a POST request. From RFC2616, section 9.5;

```
   Responses to this method are not cacheable, unless the response includes
appropriate Cache-Control or Expires header fields. However, the 303 (See
Other) response can be used to direct the user agent to retrieve a cacheable
resource.
```

This means that if a POST response includes Cache-Control or Expires (or, given a strict reading of section 13, even a validator like Last-Modified or ETag), the response can be used to satisfy future GET requests.

For example, a blog entry page could accept comments by taking a POST with the comment text; if the response were cacheable, the commenter would immediately see their comment on the page, even if they reloaded from cache (because their cache would contain the POST response).

Implementations that pass this test will cache a POST response.

1. POST freshness: fail (POST not cached)

```
This test checks whether responses to POST-requests are cached
for subsequent GET-requests if the server provides appropriate
Expires or Cache-control headers. It is done by first POSTing,
then GETting the same URI.

Conclusion : FF fails this test, probably because POST and GET
URLs are encoded into different cache-keys and the GET would not
find a cache-entry for the URL because of the different key.

Related bugs : 417841
```

Cache Invalidation

Even if an implementation doesn't actively cache POST responses, it needs to invalidate the cache when a POST is made. Otherwise, the wrong response may be sent from cache.

For example, if the blog entry page above used the same URI for POSTing comments and GETting the latest version of the entry, an implementation that doesn't invalidate the cache upon a POST will show an old version of the page (out of cache) upon a subsequent GET.

HTTP talks about this in RFC2616, section 13.10;

```
   Some HTTP methods MUST cause a cache to invalidate an entity. This is

either the entity referred to by the Request-URI, or by the Location

   or Content-Location headers (if present). These methods are:

      - PUT

      - DELETE

      - POST

   In order to prevent denial of service attacks, an invalidation based

   on the URI in a Location or Content-Location header MUST only be

   performed if the host part is the same as in the Request-URI.

   A cache that passes through requests for methods it does not

   understand SHOULD invalidate any entities referred to by the

   Request-URI.
```

POST Invalidation

Implementations that pass this test will invalidate the appropriate cache entry (the Request-URI, the Content-Location and Location) upon a POST. Implementations that fail this test are not conformant to RFC2616, and will serve the incorrect entry from cache.

- Request-URI: fail (POST to Request URI did not invalidate the cache.)
- Content-Location: fail (POST to Content-Location did not invalidate the cache.)

- Location: fail (POST to Location did not invalidate the cache.)

PUT Invalidation

Implementations that pass this test will invalidate the appropriate cache entry (the Request-URI, the Content-Location and Location) upon a DELETE. Implementations that fail this test are not conformant to RFC2616, and will serve the incorrect entry from cache.

- Request-URI: fail (PUT to Request URI did not invalidate the cache.)
- Content-Location: fail (PUT to Content-Location did not invalidate the cache.)
- Location: fail (PUT to Location did not invalidate the cache.)

DELETE Invalidation

Implementations that pass this test will invalidate the appropriate cache entry (the Request-URI, the Content-Location and Location) upon a DELETE. Implementations that fail this test are not conformant to RFC2616, and will serve the incorrect entry from cache.

- Request-URI: fail (DELETE to Request URI did not invalidate the cache.)
- Content-Location: fail (DELETE to Content-Location did not invalidate the cache.)
- Location: fail (DELETE to Location did not invalidate the cache.)

Unknown Method Invalidation

RFC2616, section 13.10 goes on to say how unrecognised methods should be handled;

```
A cache that passes through requests for methods it does not

understand SHOULD invalidate any entities referred to by the

Request-URI.
```

- Request-URI: fail (Unknown method to Request URI did not invalidate the cache.)

These tests check whether mutating methods (POST, PUT or DELETE) as well as unknown methods invalidates cached entries for the URI. Please see https://bugzilla.mozilla.org/show_bug.cgi?id=327765 for details, testcase and patch.

There is an issue with these tests though : Since they all load the same URI, a properly cached response for the POST sub-test will make the subsequent sub-tests (PUT, DELETE and Unknown) successful as well, even though only the POST-method has been fixed. Similarly, if only the DELETE-method was fixed, the POST

and PUT sub-tests would fail, but DELETE and the subsequent Unknown sub-test would be successful. The unit-test attcahed to bug 327765 avoids this by loading one URI for each method.

Conclusion : FF fails this test but a patch is available. The test have a subtle issue which can lead to false positives.

Related bugs : 327765, 200948