Maximilian Bloch

BROWSER-CACHING – DER HEIMLICHE FREUND DEINES USERS

Pagespeed ist nicht nur im SEO ein Dauerbrenner. Von Google als Rankingfaktor erklärt, von UX-lern als selbstverständlich erachtet und kaum eine Fachkonferenz, auf welcher die Ladezeit-Optimierung kein Thema ist. Dabei fällt auf, dass ein bedeutender Hebel oft nur wenig Beachtung erfährt: das clientseitige Caching, also das Caching im Browser. Synthetische Performance-Tests, welche sich vornehmlich auf den ein- und erstmaligen Aufruf einer Seite konzentrieren, tun ihr Übriges dazu, dass diesem Faktor oft nicht die Aufmerksamkeit geschenkt wird, welche er eigentlich verdient. Bei solchen Tests zeigen Optimierungen des Caching-Verhaltens keine Steigerung der gemessenen Geschwindigkeit. Allerdings bildet dies oft nicht die Realität ab.

Das Internet ist nicht ausschließlich von Onepager besiedelt, sondern im besten Fall bleibt uns der User treu und navigiert über mehrere Seiten. Und genau hier spielt das Caching im Browser seine Stärken aus.

Wie in so vielen Bereichen gilt jedoch auch hier: Ohne fundierte Kenntnisse und das Wissen um die Relevanz einzelner Werte verleiten Messergebnisse, wie in Abb. 1 gezeigt, gerne zu fehlgeleiteten Optimierungsansätzen.

Gefühlt gibt es beim Verständnis von Performance-KPIs bereits bedeutende Fortschritte. Es wird nicht mehr so eng auf Zeiten wie die Fully Loaded Time geschaut, stattdessen haben Rendering-sensible Werte wie First Contentful Paint (FCP) oder First Meaningful Paint (FMP) an Bedeutung gewonnen. Jedoch rückten mit dieser sehr sinnvollen Entwicklung auch vermehrt Versuche in den Vordergrund, das Rendering bzw. dessen KPIs zu optimieren.

„Eliminate render-blocking resources“, so melden es Pagespeed Insights und der Lighthouse-Report von Google.

Das macht auch absolut Sinn! Nur eben nicht um jeden Preis. Ohne ein Auge auf damit einhergehende Wechselwirkungen zu haben, kann die Optimierung mit starkem Fokus auf Rendering-Zeiten in Summe durchaus auch Nachteile mit sich bringen. Beispielhaft ist hier das „Inlining“ des gesamten CSS zu nennen, also die

Foto: beastfromeast / gettyimages.de

DER AUTOR

Maximilian Bloch ist auf technische Suchmaschinen- und Ladezeitoptimierung spezialisiert. Er ist Inhaber des SEO-Ateliers und assoziierter Berater bei der elaboratum GmbH.

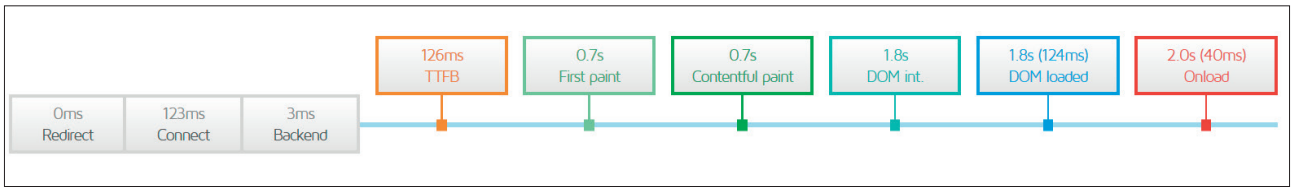


Abb. 1: Performance-Werte aus dem Tool gtmetrix.com

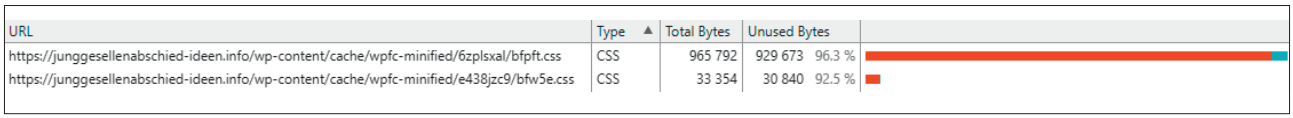


Abb. 2: Chrome Entwickler-Tools, Coverage-Report – zeigt die Abdeckung des genutzten Codes in Blau

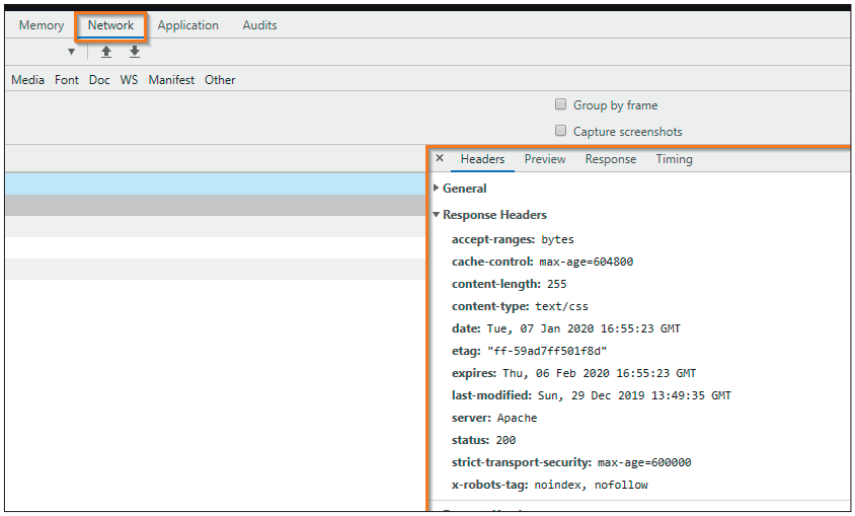


Abb. 3: Chrome Entwickler-Tools – HTTP-Header finden sich unter dem Tab „Network“

Platzierung des vollständigen CSS im HTML-Head. Im Falle von CSS handelt es sich um eine Ressource, welche das Rendering blockiert, sprich: Der Browser muss auf die Datei warten, bevor er die Darstellung fortsetzen kann.

Da mit dem Inlining kein zusätzlicher Request mehr zum Abruf des Styles benötigt wird, senkt dies (vorausgesetzt, es blockieren nicht zeitgleich andere Ressourcen) die Werte der FCP und FMP beim initialen Aufruf einer Seite. Alles gut also? Nein, denn dabei wird häufig außer Acht gelassen, dass das CSS mit Integration in das HTML nicht mehr im Browser „gecacht“ werden kann. Beim Aufruf einer zweiten Seite der Website müsste der User das gesamte CSS erneut herunterladen. Spätestens beim Besuch weiterer Seiten ergibt diese Technik also Nachteile, welche meist mit der Senkung der erstmaligen Ladezeit nicht mehr auszugleichen sind.

Vor- und Nachteile solcher Optimierungen müssen abgewogen werden und im Einklang mit dem User-Verhalten stehen. Beispielsweise müssen hier die durchschnittliche Anzahl von Seitenaufrufen je Sitzung und die wiederkehrenden Besucher mit in die Betrachtung einbezogen wurde. Auch die Anzahl der Seiten spielt eine Rolle sowie die Größe des CSS selbst. Gerade die Wordpress-Gemeinde ist oft von solchen Entscheidungen betroffen. Es wird dem Webmaster auch einfach gemacht: Mit einem Klick im richtigen Caching-Plug-in findet sich das gesamte CSS im HTML wieder. Und Pagespeed Insights meldet unter Umständen einen Erfolg. Immer mehr Wordpress-Systeme basieren zudem auf sehr umfangreichen Themes und Frameworks. Diese müssen allerdings aufgrund ihrer breiten Flexibilität auch viel ungenutzten Code mitbringen. Die Verwertung des gelieferten Codes zeigt dann häufig Werte wie in Abb. 2.

Um zumindest einen Teil dieses CSS „cachbar“ zu machen und dennoch die Vorteile des „Inlinings“ nutzen zu können, wird wiederum oft zu einer Methode gegriffen, bei welcher lediglich das CSS für die Darstellung „Above the fold“ (also der primär, ohne zu scrollen, sichtbare Inhalt) inline eingebunden wird, während das restliche CSS per JavaScript nachgeladen wird.

Aber mal ehrlich: Das funktioniert nur in den seltensten Fällen wirklich gut und führt sehr häufig zu sog. „Flash of Unstyled Content“ (kurz „FOUT“, einem sichtbaren „Springen“ der Darstellung der Seite), was gefühlt noch grausamer ist als eine marginal langsamere Seite.

Die oben genannten Methoden sollen keinesfalls schlechtgeredet werden. In Bezug auf den einzelnen Seitenabruf lassen sich damit Websites sehr effizient beschleunigen. Allerdings finden bei aller gerechtfertigten Optimierung der initialen Darstellung oftmals die Folgeschritte zu wenig Beachtung.

Die Performance einer Website lässt sich endlos optimieren, jedoch hat man mit einer akzeptablen Antwortzeit des Servers und einem funktionierenden Browser-Caching schon viel richtig gemacht. Allerdings kann man auch beim Set-up des Browser-Cachings nicht nur gewinnen. Falsche Einstellungen können mitunter zu Problemen führen, welche im schlimmsten Fall die gesamte Funktionalität der Website gefährden.

INFO

Etags werden in einen schwachen und einen starken Etag unterschieden. Den Wert eines schwachen Etag dürfen mehrere identische Ressourcen tragen. Die Werte von schwachen Etags sind mit dem Präfix „W/“ gekennzeichnet.

Browser-Caching: Was und wie?

Beim Caching handelt es sich um ein Zwischenspeichern bereits übertragener Informationen bzw. Dateien. Der Browser muss somit eine bereits heruntergeladene Datei kein zweites Mal vom Server abrufen und empfangen.

Ein effizientes Caching bringt nicht nur Vorteile, sofern der User die gleiche Seite erneut aufruft. Auch wenn er von der ersten auf die zweite Seite der gleichen Website wechselt, muss z. B. das Logo nicht erneut heruntergeladen werden und kann direkt aus dem Speicher des Browsers bezogen werden. Ganz wichtig ist hierbei: Das Logo in unserem Beispiel hat sich zwischenzeitlich nicht verändert. Dieser Umstand wird noch eine Rolle spielen.

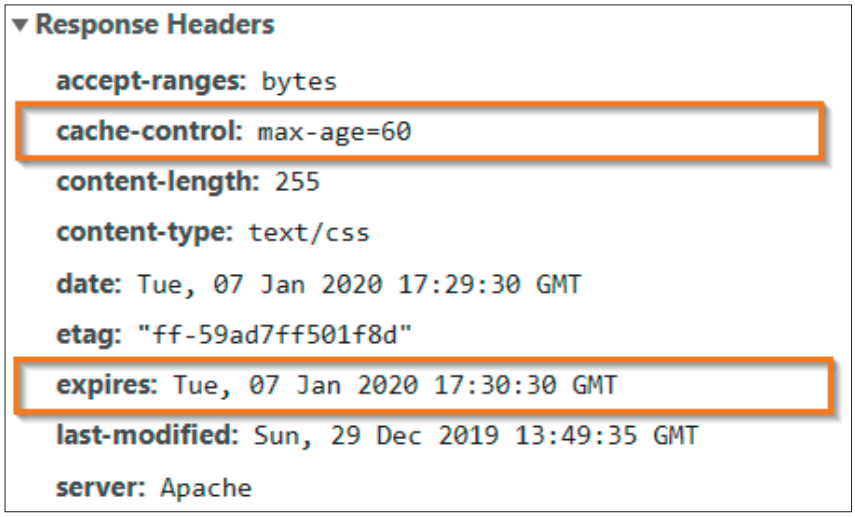


Abb. 4: Chrome Entwickler-Tools – mit diesen Headern kann die Ablaufzeit von Ressourcen bestimmt werden

Gesteuert wird das Caching im Browser über HTTP-Header. Diese erlauben es, Client und Server zusätzliche Informationen bei Request oder Response zu übergeben. Zu finden sind sie in den Developer-Tools im Browser unter dem Network-Tab (Abb. 3). Mit Klick auf eine einzelne Request-Zeile lassen sich zusätzliche Daten anzeigen, u. a. auch die Request- und Response-Header.

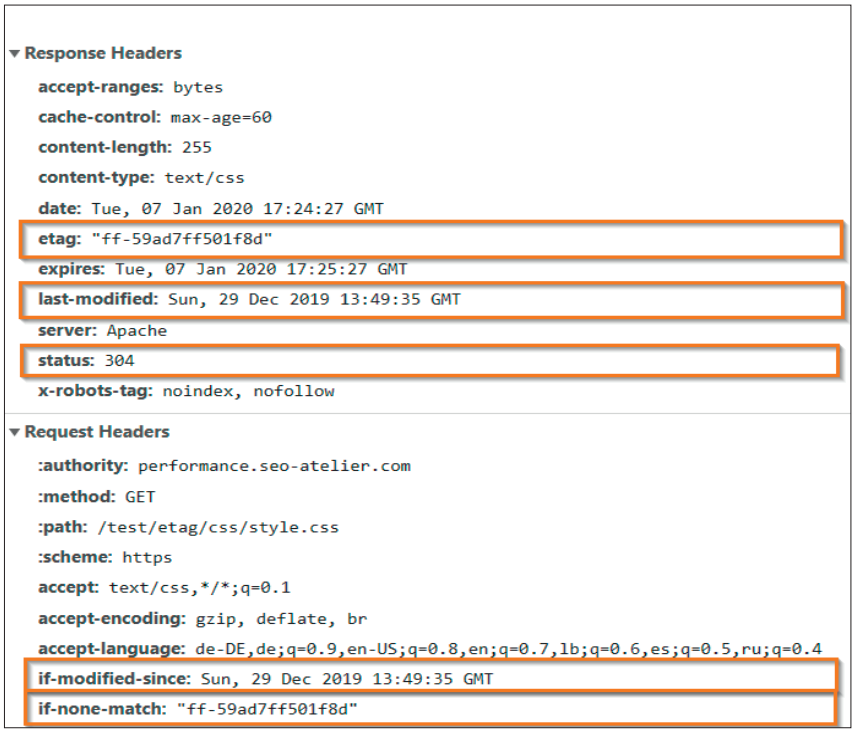


Abb. 5: Chrome Entwickler-Tools – nach Ablauf der TTL wurde mit einem Conditional-Request die Datei validiert; der Server hat hierzu mit dem Status 304 geantwortet

Gesteuert werden können die Header und deren Werte auf dem Server entweder über die .htaccess-Datei oder die Server-Konfiguration. Weiter können Header auch programmseitig, beispielsweise über PHP, hinzugefügt werden. Nutzer eines CDN haben außerdem meist die Möglichkeit, diese Angaben direkt über die CDN-Einstellungen zu steuern.

Caching-Header und ihre Wirkung

Für die Steuerung der Caching-Dauer (der sog. TTL = „Time to live“) stehen zwei Header zur Verfügung (Abb. 4). Der Cache-Control-Header entstammt dem HTTP1.1-Protokoll, während es den Expires-Header bereits unter HTTP1.0 gab. Der Pragma-Header wird lediglich zur Abwärtskompatibilität mit HTTP1.0-Caches benötigt und deswegen vorerst nicht weiterverfolgt.

Beide funktionieren im Kern gleich, der Cache-Control-Header kann jedoch einfach etwas mehr als der Expires-Header. Im Beispiel unter Abbildung 4 sind jedoch beide äquivalent zueinander und stellen somit eine redundante Angabe dar, was grundsätzlich kein Problem ist. Der Expires-Header wird in diesem Fall einfach ignoriert und der Browser gibt dem Cache-Control den Vortritt.

In unserem Fall haben wir die TTL mit der Angabe „max-age=60“ im Cache-Control-Header auf 60 Sekunden gesetzt. Der „Expires“-Header rechnet hingegen einfach 60 Sekunden auf den

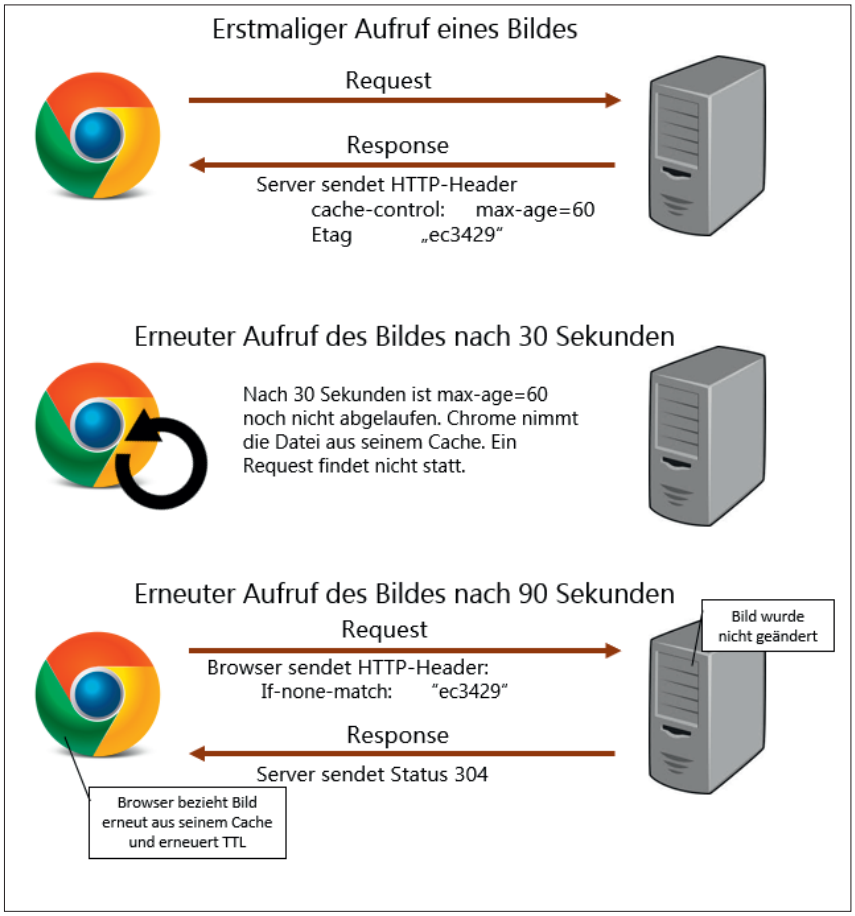


Abb. 6: Caching & Validierungsverhalten bei Chrome – so arbeiten Browser und Server zusammen

aktuellen Zeitpunkt in die Zukunft und gibt somit ein genaues Ablaufdatum mit Uhrzeit an. Die Ressource, in diesem Fall eine CSS-Datei, wird also für eine Minute im Client gespeichert und so lange ohne erneuten Download wiederverwendet.

Conditional-Header und ihre Funktion

Im Beispiel der Abb. 4 finden sich jedoch noch weitere Caching-relevante Angaben: die sogenannten Conditional-Header. Hierzu zählen der Entity-Tag (Etag) und der „Last-Modified"-Header.

Analog zu Cache-Control- und Expires-Header verfolgen auch diese beiden Angaben im Grunde den gleichen Zweck. Last-Modified ist hier wieder der ältere Header (ihn gab es bereits unter HTTP1.0), während der Etag erst mit HTTP1.1 verfügbar war.

Der Etag stellt, vereinfacht gesagt, einen individuellen Fingerabdruck der jeweiligen Datei dar, quasi eine Quersumme. Ändert sich die Datei, ändert sich auch der Etag. Last-Modified gibt hingegen einfach den Zeitpunkt der letztmaligen Aktualisierung an und ist damit etwas weniger genau.

Die Funktion des Etag bzw. Last-Modified lässt sich am einfachsten anhand eines typischen Caching-Ablaufs erklären. Im obigen Beispiel wurde die TTL auf 60 Sekunden gesetzt. Was aber, wenn diese Zeit abgelaufen ist? Muss sich der Browser dann die Datei erneut herunterladen? Nicht unbedingt. Ein valider Etag oder eine stimmige Last-Modified-Angabe ermöglichen es dem Browser, eine Anfrage an den Server zu stellen, deren einziger Zweck es ist, zu prüfen, ob sich die Datei seit dem letzten Download geändert hat – die sog. Validierung. Dazu sendet der Browser einen sog. Conditional-Request mit den zuvor empfangenen Etag- bzw. Last-Modified-Daten.

Im Beispiel von Abbildung 5 wird if-none-match und der Wert des vorher empfangenen Etags gesendet. Stimmen diese Angaben noch mit der aktuellen Etag-Berechnung des Servers überein, muss die im Browser gespeicherte Version noch aktuell sein. Der Server antwortet mit dem Statuscode 304 (not modified) anstelle von 200. Dieser Status veranlasst den Browser wiederum, die bereits im Cache liegende Datei erneut zu verwenden. Gleichzeitig wird die TTL der ursprünglichen Response erneuert. Heißt: In unserem Fall würde die Datei also erneut 60 Sekunden direkt aus dem Cache bezogen.

Etag – Achtung bei großen Set-ups

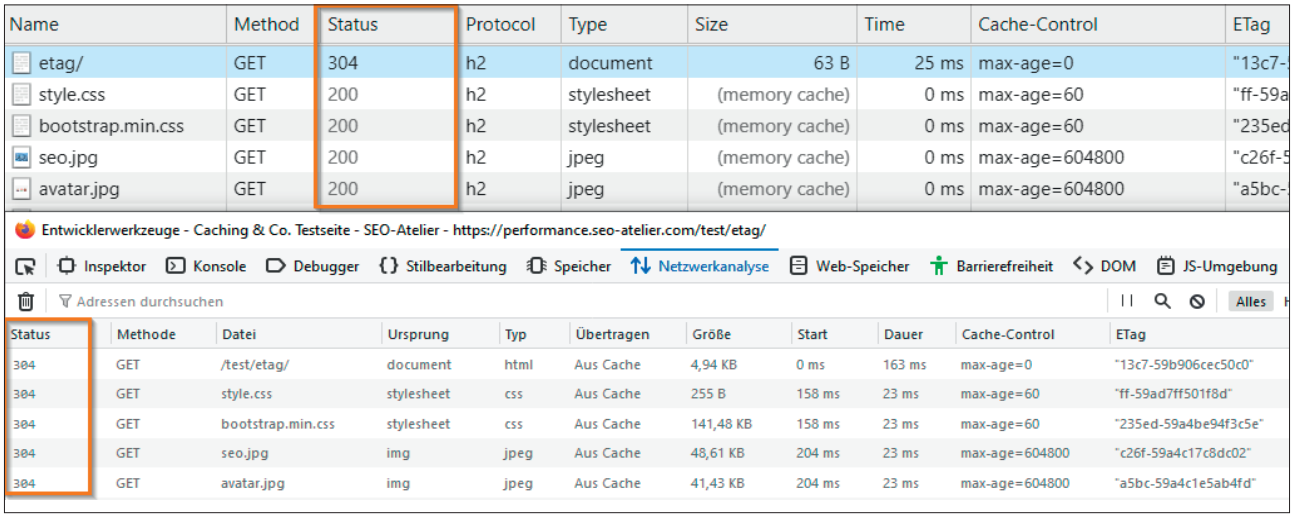
Ein funktionierender Etag ist großartig. Dennoch stellt man häufig fest, dass der Etag bei vielen Systemen per Default deaktiviert ist. Aber warum? Schließlich ermöglicht dieser doch ein effizienteres Caching.

Dies liegt daran, dass die Art, wie ein Etag generiert wird, oftmals nicht mit der zugrunde liegenden Infrastruktur harmoniert. Nehmen wir an, dass es sich in unserem Beispiel um keine kleine Website handelt und diese von mehr als einem Server ausgeliefert wird, vielleicht, weil die Requests über einen Load-Balancer auf verschiedene Server verteilt werden.

Etags werden üblicherweise aus drei Faktoren errechnet:

- **Inode** (wo liegt die Datei auf dem Server im Dateisystem?)
- **Timestamp** (wann wurde die Datei dort abgespeichert?)
- **Size** (Größe der Datei?)

Die Verteilung auf die hinter dem Load-Balancer liegenden Server hätte jedoch zur Folge, dass ein Heranziehen der Inode-Information nicht mehr valide wäre. Diese ist nur innerhalb eines Systems gültig. Sofern uns beim Seitenwechsel die Response von einem anderen Server beantwortet wird, würde sich mindestens die Inode-Information und damit auch der Etag der Datei ändern.



Name	Method	Status	Protocol	Type	Size	Time	Cache-Control	ETag
etag/	GET	304	h2	document	63 B	25 ms	max-age=0	"13c7-
style.css	GET	200	h2	stylesheet	(memory cache)	0 ms	max-age=60	"ff-59a
bootstrap.min.css	GET	200	h2	stylesheet	(memory cache)	0 ms	max-age=60	"235ed
seo.jpg	GET	200	h2	jpeg	(memory cache)	0 ms	max-age=604800	"c26f-5
avatar.jpg	GET	200	h2	jpeg	(memory cache)	0 ms	max-age=604800	"a5bc-

Status	Methode	Datei	Ursprung	Typ	Übertragen	Größe	Start	Dauer	Cache-Control	ETag
304	GET	/test/etag/	document	html	Aus Cache	4,94 KB	0 ms	163 ms	max-age=0	"13c7-59b906cec50c0"
304	GET	style.css	stylesheet	css	Aus Cache	255 B	158 ms	23 ms	max-age=60	"ff-59ad7ff501f8d"
304	GET	bootstrap.min.css	stylesheet	css	Aus Cache	141,48 KB	158 ms	23 ms	max-age=60	"235ed-59a4be94f3c5e"
304	GET	seo.jpg	img	jpeg	Aus Cache	48,61 KB	204 ms	23 ms	max-age=604800	"c26f-59a4c17c8dc02"
304	GET	avatar.jpg	img	jpeg	Aus Cache	41,43 KB	204 ms	23 ms	max-age=604800	"a5bc-59a4c1e5ab4fd"

Abb. 7: Google Chrome meldet Status 200, während Firefox 304 anzeigt – beide Browser zeigen bei gleichen Headern ein unterschiedliches Caching-Verhalten

Anderer Server = andere Inode-Information = anderer Etag. Der Browser würde also sehr häufig die Information erhalten, dass die Datei nicht mehr dieselbe ist und er diese erneut herunterladen soll, obwohl nichts geändert wurde.

Die Berücksichtigung von Inode kann bei der Generierung des Etags relativ einfach deaktiviert werden. Was aber, wenn es sich bei unseren Servern beispielsweise um Caching-Server handelt, welche die Response selbst erst bei Bedarf von dahinterliegenden Servern beziehen?

Je nach Konfiguration der Server (z. B., wenn die Caching-Server selbst die entsprechenden Header setzen) könnte nun auch der Timestamp kein valider Faktor mehr sein, da ggf. auf dem zweiten Server die gleiche Datei erst später zwischengespeichert wird als beim ersten. Auch das Heranziehen dieses Faktors ließe sich bei der Berechnung des Etags ausschließen, allerdings wäre lediglich die Nutzung der Size ein zu ungenauer Indikator für die Etag-Generierung. Konkretes Beispiel: Würden wir auf einer Produktseite den Preis von 19,99 € auf 29,99 anpassen, würde dies der Client nicht mitbekommen, da sich die Dateigröße nicht verändert hätte. Um dieses Problem lösen zu können, müssen wir uns vorher noch genauer mit dem Browserverhalten auseinandersetzen.

INFO

Die Validierung beim Reload vor Ablauf der TTL kann im Firefox mit der zusätzlichen Angabe von „immutable“ abgeschaltet werden. Allerdings validiert dieser dann die Datei gar nicht mehr, auch nicht, sofern die Ablaufzeit überschritten ist, sondern lädt diese erneut herunter, weswegen die Verwendung von „immutable“ nur auf Seiten mit häufigen Reloads und in Verbindung mit langer Caching-Dauer Sinn macht. Facebook setzt diesen Header beispielsweise sehr häufig ein.

Besonderheiten: Seitentyp und Browser

Mit der oben beschriebenen Fehlerquelle hätten wir jedoch zumindest noch nichts verloren, sofern nach Ablauf der TTL der Browser zum erneuten Download angewiesen würde. Das Caching an sich würde innerhalb der TTL ja noch funktionieren. Erst nach deren Ablauf würde die Datei unter Umständen erneut heruntergeladen. Das wäre zumindest nicht schlechter als ein Szenario komplett ohne Etag.

Allerdings kommen nun maßgeblich der vorliegende Seitentyp und die Eigenheiten der Browser ins Spiel. Browser reagieren unterschiedlich darauf, ob der Request aufgrund einer Navigation über die Seiten (beispielsweise aufgrund des Klicks auf einen Link) getätigt wird oder die Seite vom User manuell (beispielsweise mit F5) aktualisiert wird. Startseiten von News-Publishern oder Seiten mit User-Feeds werden beispielsweise tendenziell häufiger vom User aktualisiert als die typische Wikipedia-Seite.

Sofern man sich nun das Verhalten von Chrome und Firefox beim Reload genauer ansieht, lassen sich Unterschiede erkennen.

Die obere Ansicht in Abbildung 7 zeigt die Chrome-Developer-Tools, die untere die des Firefox. In beiden wurde die Seite per F5 aktualisiert. Beide Browser beziehen Bilder und CSS aus ihrem Cache, allerdings erhält der Firefox häufig eine Response mit dem Statuscode 304 (not modified), während Chrome den Status 200 empfängt.

Aber warum erhalten die Browser überhaupt Statuscodes? Immerhin war zum Abruf-Zeitpunkt die TTL der Ressource noch nicht abgelaufen, was wiederum bedeuten würde, dass gar kein Request an den Server gesendet werden sollte. Dieser sollte also auch gar nicht mit irgendeinem Statuscode antworten können. Ohne Anfrage keine Antwort.

Das hat zwei verschiedene Gründe: Die einfachere Begründung trifft hier auf das Verhalten von Chrome zu. Die Zeilen sind hier leicht ausgegraut, was bedeutet, dass der Request direkt aus dem Cache bedient wurde, ohne Anfrage an den Server. Es fand tatsächlich kein Request an den Server und

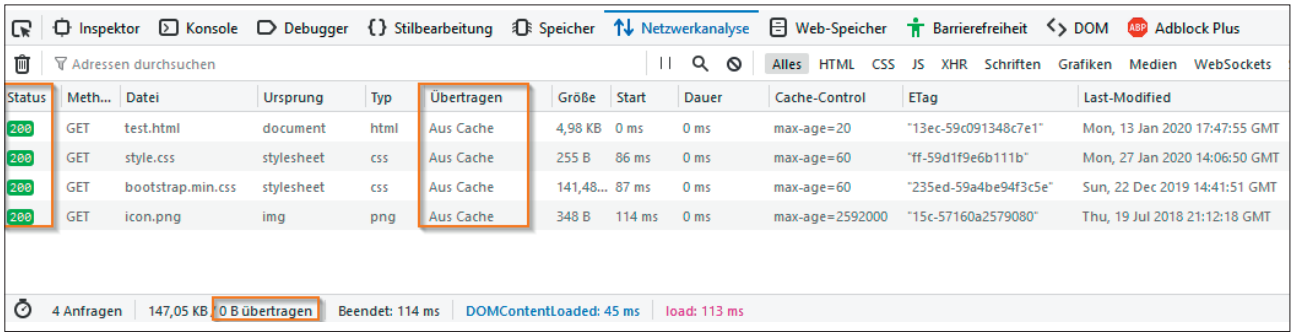


Abb. 8: Dieser Seitenaufruf kam über eine User-Navigation zustande – auf die gleiche Seite wurde innerhalb von 20 Sekunden erneut navigiert; in diesem Fall wird überall die TTL berücksichtigt und keine Bytes übertragen

```
<img src="img/seo.jpg" alt="Etags und Validierung" title="Validierung mittels Etag"/>
```

Abb. 9: Vor und nach Änderung des Bildes ist das der Aufruf aus dem HTMLkeine Bytes übertragen

keine Response von ihm statt. In der Spalte „Size“ meldet Chrome zusätzlich den Bezug der Ressource aus dem Cache.

Beim Firefox erhalten wir allerdings deutlich häufiger den Status 304, inklusive der Information, dass auch hier die Responses aus dem Cache bezogen wurden. Beim manuellen Reload unterscheiden sich die Browser im Verhalten, sofern der Cache-Control-Header nicht mit noch weiteren Angaben angereichert wird. Während Chrome relativ aggressiv cacht und die Aktualität von Ressourcen nur dann per Conditional-Request prüft, wenn die TTL abgelaufen ist, geht Firefox deutlich behutsamer vor. Denn Firefox prüft hier jede Ressource auf Aktualität, unabhängig davon, ob die TTL abgelaufen ist oder nicht. Erst wenn der Server eine 304-Response meldet, wird die Ressource erneut aus dem Cache bezogen.

Sofern nur max-age im Cache-Control-Header angegeben ist, bedeutet dies beim Reload:

» Chrome beachtet die TTL, validiert nach Ablauf dieser und erneuert die TTL anschließend wieder. Das HTML wird unabhängig von einer TTL validiert.
» Firefox ignoriert die TTL, validiert beim Reload jedes Mal mittels Etag oder Last-Modified und bezieht die Ressource erst dann aus seinem Cache.

Je nach Seitentyp würde uns also bei einer fehlerhaften Generierung des Etags unser Caching-Plan um die Ohren fliegen.

Betrachtet man Abbildung 7 weiter, lassen sich aber auch Gemeinsamkeiten im Browserverhalten feststellen. Beim Reload der Document-Dateien (also unserem HTML) haben beide Browser den Status 304 erhalten: Das HTML selbst wird in diesem Fall immer von beiden Browsern validiert, unabhängig davon, ob eine TTL angegeben ist.

Findet jedoch ein Seitenwechsel aufgrund eines Klicks statt, wird die TTL, unabhängig davon, ob HTML oder ein anderer Filetype vorliegt, von beiden Browsern beachtet. In Abbildung 8 findet sich ein solches Beispiel. Der User hat innerhalb der beim HTML angegebenen TTL (20 Sekunden) erneut auf die gleiche Seite navigiert. In diesem Fall wurde nichts validiert und es fand keine Datenübertragung statt.

HTML mit oder ohne Ablaufzeit: Aktualität vs. Caching-Effizienz?

An dieser Stelle sollten wir uns die Frage stellen, ob die Angabe einer anderen Ablaufzeit als „max-age=0“ beim HTML überhaupt Sinn ergibt. In der Regel und bis auf wenige Ausnahmen ist die Antwort hierauf: Nein.

Denn erstens bewirkt ein „max-age=0“ nicht, wie oftmals fälschlicherweise angenommen, „nicht cachen“. Sie bewirkt lediglich, dass quasi nach Ablauf von 0 Sekunden wieder validiert werden soll. Der Browser muss also die Aktualität immer per Etag oder Last-Modified prüfen, darf die Datei danach jedoch sehr wohl aus seinem Cache beziehen.

Zweitens fehlt uns bei öffentlichen HTML-Dateien ein großer Vorteil im Vergleich zu CSS-, JS usw.: Wir können die URL nicht regelmäßig und häufig ändern! Allein aus SEO-technischen Gründen hätte ein häufiger Wechsel der URLs mit unserem eigentlichen Inhalt katastrophale Folgen. Bei den restlichen Ressourcen sieht das allerdings anders aus.

Der Umstand, dass dies bei den übrigen Ressourcen möglich ist, erlaubt uns, hier eine maximale Caching-Dauer anzugeben, ohne Gefahr zu laufen, dass der User eine veraltete Information zu Gesicht bekommt.

INFO

Auch beim Einsatz von Accelerated Mobile Pages (AMP) sollte großer Wert darauf gelegt werden, dass die Cache-Validierung korrekt funktioniert. Schaut man sich in den Logfiles die Aufrufe einer AMP-Seite an, wird man sehr schnell feststellen, dass Google hier anders vorgeht als beim Abruf von normalen Seiten. Diese werden stets mit Status 200 heruntergeladen. Der AMP-Crawler nutzt hingegen stark Conditional-Requests, um zu prüfen, ob sich etwas auf der Seite geändert hat. Da dabei kein Datei-Transfer erfolgen muss, ermöglicht dieses Vorgehen dem Crawler, die Aktualität deutlich öfter zu prüfen als bei normalen HTML-Seiten.

TIPP

Wem selbst die Validierung des HTML zu lange dauert und wer trotzdem nicht massiv auf Aktualität verzichten bzw. ganz generell seine Time to first Byte für den User senken möchte, sollte sich das kompakte Skript Instant.page (URL: instant.page) ansehen. In der Standardkonfiguration wird damit bereits beim Maus-Hover das entsprechende Dokument per „preload“ vorgeladen, sodass der Browser es beim Klick aus seinem Cache beziehen kann. Auf mobilen Endgeräten wird die Zeit zwischen dem Berühren und Entfernen des Fingers dafür verwendet. Damit auch der Firefox nicht mehr validiert, sollte die TTL des HTML auf fünf Sekunden gesetzt werden. Etwaige Auswirkungen auf Logfile-basierende Analysen sollten hierbei berücksichtigt werden.

Stellen wir uns vor, wir haben für unsere Bilder eine TTL von einem Monat hinterlegt. Nun tauschen wir aus Designgründen das Bild „seo.jpg“ durch ein neues Bild aus. Da wir als SEO-affine Bediener allerdings auf die Dateinamen unserer Bilder achten, verwenden wir wieder „seo.jpg“. Der Aufruf des Bildes würde wie in Abb. 9 aussehen.

Da sich der Dateiname und damit der Pfad zu unserem Bild nicht geändert hat, würde der Browser die alte Version ohne Überprüfung aus seinem Cache beziehen und dem User somit im schlimmsten Fall fast einen ganzen Monat lang das falsche Bild anzeigen.

Firefox würde hingegen auch nur bei einem Reload auf der gleichen Seite den Etag des Bildes prüfen und somit ebenfalls nur selten die Aktualisierung mitbekommen. Aber was machen wir mit den restlichen Usern? Die falsche Darstellung einen Monat lang akzeptieren oder einfach kürzer cachen?

Nein, aus diesem Grund werden sog. Fingerprint-URLs verwendet. Alle URLs statischer Ressourcen werden mit einem automatisch generierten Hash ausgeliefert, der sich jedes Mal ändert, sofern die Datei geändert wurde. In

```
<img src="img/seo.jpg" alt="Etags und Validierung" title="Validierung mittels Etag"/>
```

Abb. 10: Referenz des gleichen Bildes mit Fingerprint in der URL; wird die Datei geändert, ändert sich auch der Fingerprint – unabhängig davon, ob die neue Datei den gleichen Dateinamen wie zuvor trägt, ändert sich damit die URL der Ressource

Abb. 10 findet sich der modifizierte Aufruf mit einem „Fingerprint“ in der URL.

Die Änderung der URL bewirkt, dass der Browser die Ressource nicht mehr aus dem Cache beziehen kann, da es sich für ihn um eine neue URL und damit unbekannte Ressource handelt, für welche natürlich auch kein Cache-Eintrag besteht. Sofern ein User nun beispielsweise einen Reload auf der gleichen Seite machen würde, spielt sich Folgendes ab:

» Die Browser versuchen, beim Reload das HTML zu validieren, und senden einen Header „if-none-match“ mit dem Wert des zuvor empfangenen Etags.
» Der Server gleicht den Wert des vom Client gesendeten Etags mit der aktuellen Berechnung ab. In unserem Fall hat sich die HTML-Datei geändert, da der Pfad des Bildes im HTML durch den Fingerprint aktualisiert wurde.
» Hierdurch hat sich wiederum der Etag geändert. Die „if-none-match“-Bedingung trifft nicht zu. Der Server antwortet mit dem Statuscode 200 und der erneute Empfang des aktualisierten HTML-Dokuments findet statt.
» Der Browser parst das HTML und entdeckt die Source-URL auf das Bild. Diese URL ist für ihn unbekannt, er hat dazu keine Informationen im Cache.
» Er lädt das aktuelle Bild herunter und zeigt diese Version dem User an.

Dies erklärt nun auch, warum die Validierung der HTML-Datei mit „max-age=0“ meist Sinn macht. Wir können die URL unseres HTML-Dokuments selbst ja schließlich nicht bei jeder Aktualisierung ändern. Um einen Vorteil aus einer längeren TTL zu schöpfen, müsste der User öfter über das gleiche Dokument navigieren. Da diese Angabe uns jedoch die Möglichkeit der Aktualisierung nehmen würde, könnte die TTL ohnehin nur relativ kurz sein, was wiederum die Wahrscheinlichkeit mindert, dass der gleiche User innerhalb der angegebenen Zeit ein zweites Mal über die gleiche Seite navigiert. Wie man sieht, macht der Validierungs-Request auf das HTML-Dokument also meistens Sinn und sollte somit in der Regel auch nicht verhindert werden. Gerade in Verbindung mit Fingerprint-URLs erlaubt uns dieses Vorgehen, die TTL der restlichen Ressourcen auf ein Maximum zu setzen, ohne Gefahr zu laufen, die Aktualität nicht mehr wahren zu können.

Oft findet man solche Fingerprints oder Versionierungen auch in Form von Get-Parametern. Das bloße Vorhandensein von Query-Strings veranlasst allerdings manche Proxy-Server dazu, die Ressource komplett vom Caching auszuschließen, weswegen die Variante des Fingerprints im Pfad der URL deutlich sinnvoller ist.

Wenn wir uns nun unser problematisches Szenario in Erinnerung rufen, bei welchem wir den Etag nicht durch Verwendung von Inode und Timestamp berechnen lassen konnten, bietet uns die Versionierung mit Fingerprint einen Ausweg bzw. zumindest die Option auf einen Workaround. Dieser würde so aussehen:

» Wir verzichten gänzlich auf den Etag und deaktivieren diesen beispielsweise durch den Eintrag „Header unset ETag“ in der .htaccess.
» Wir verwenden allerdings den Last-Modified-Header. Diesen Wert lassen wir jedoch nicht (wie eigentlich korrekt) vom Server generieren. Wir hinterlegen einfach ein fixes Datum in der Vergangenheit für alle Ressourcen, welche gecacht werden

sollen und bei welchen wir mit einem Fingerprint in der URL arbeiten können.
» Wir setzen die TTL für diese Dateien per Expires-Header auf ein ganzes Jahr.
» Bei HTML-Files lassen wir Etag/Last-Modified weiter korrekt vom Server generieren, setzen die TTL jedoch auf 0 Sekunden.

Aus diesem Set-up würde folgen, dass für unsere Ressourcen unabhängig vom ausliefernden Server stets eine positive Validierung erfolgen würde. Um die Validierungen, welche ebenfalls Zeit kosten, so selten wie möglich durchführen zu müssen, wurde die TTL (außer beim HTML) auf ein Jahr gesetzt. Sofern sich in diesem Zeitraum etwas ändert, versichern uns die Fingerprints in den URLs und die immer stattfindende Validierung auf dem HTML, dass der User stets die aktuelle Version zu Gesicht bekommt.

Gefahren beim Caching

Caching kann dann gefährlich werden, wenn Dinge gecacht werden, welche eigentlich nicht gecacht werden sollten. Dies fängt beim fehlerhaften 301-Redirect an, welcher gerne mal eine ganze Website für User mit befülltem Cache lahmlegen kann, und hört beim Umgang mit sensiblen Informationen auf.

Konstruieren wir hierzu wieder ein Szenario. Angenommen, wir betreuen eine Seite mit Log-in-Funktionalität. Hinter dem Log-in finden sich Informationen wie Kontostand, Name und Anschrift. In diesem Zusammenhang wird Caching problematisch, sofern Proxy-Caches ins Spiel kommen. Um die Netzwerklast zu reduzieren, werden, bei zwischengeschalteten Proxy-Caches, Anfragen des Clients nicht mehr an den Zielserver weitergeleitet, sondern als zwischengespeicherte Version vom Proxy bedient.

Sofern nun aber zwei User den gleichen Proxy nutzen, können falsche Caching-Einstellungen dazu führen, dass der zweite User die Informationen des ersten angezeigt bekommt, die gecachte Version vom Proxy. Ähnlich problematisch kann sich auch ein über falsche Header informiertes CDN verhalten. Im Verlauf des Artikels wurde bereits erwähnt, dass Cache-Control noch weitere Funktionen bietet als der Expires-Header. Diese Funktionen benötigen wir nun.

Cache-Control – mögliche Funktionen

Gemeint sind zusätzliche Angaben, welche in Kombination oder einzeln im Header hinterlegt werden können (Tabelle 1).

Weiter gibt es einige Angaben, welche nicht im Kern des HTTP-Standard-Dokumentes definiert sind, aber interessante Funktionalitäten bieten. Wichtig ist bei der Betrachtung, dass diese Cache-Header oft nicht nur von Browsern verwendet werden, sondern häufig auch Proxys oder ein ggf. vorhandenes CDN betreffen.

Cache-Control: stale-while-revalidate=Sekunden

Bei dieser Anweisung handelt es sich um eine sehr interessante Funktionalität, welche leider nicht von jedem Browser bzw. Cache unterstützt wird und noch relativ wenig Anwendung findet.

Sie gibt an, dass der Client bereit ist, für eine bestimmte Zeit eine abgelaufene Ressource zu verwenden. Sofern die TTL abgelaufen ist, wird im Hintergrund asynchron (also nicht blockierend) validiert. Die hier angegebenen Sekunden stellen die Zeit dar, welche der Client bereit ist, die abgelaufene Response zu akzeptieren. Siehe Abb. 11.

Nicht nur in Bezug auf Browser ist diese Funktion interessant: Gerade bei der Nutzung eines CDN kann diese Angabe einen näheren Blick wert sein und dafür sorgen, dass die vereinzelten langsamen Responses verhindert werden, welche dann auftreten, wenn z. B. die TTL von HTML-Seiten im CDN abgelaufen ist und der Request eigentlich an den Origin-Server weitergeleitet werden müsste. Dem anfragenden Client würde in diesem Fall noch einmal die veraltete, im CDN gecachte Response ausgeliefert, das CDN würde aber währenddessen im Hintergrund die Aktualität gegen den Ursprungsserver prüfen. Leider können auch noch nicht alle CDN-Anbieter damit umgehen.

max-age=Sekunden	Diese Angabe gibt die TTL der Ressource an und wurde weiter oben bereits beleuchtet. Die Angabe muss in Sekunden erfolgen.
must-revalidate	Veranlasst Caches dazu, die Response jedes Mal neu zu validieren, bevor sie aus dem Cache verwendet wird.
Private	Meist handelt es sich dabei um Informationen, die auf einen einzelnen User ausgerichtet sind. Die Informationen dürfen demnach nicht von Proxy-Caches gespeichert werden, vom Browser allerdings schon. Kann hinter einem Log-in sinnvoll sein.
Public	Die Response darf von jedem Cache gecacht werden.
no-cache	Heißt nicht, dass die Response nicht gecacht werden darf! Es zwingt Caches aber dazu, Requests an den Ursprungsserver durchzustellen. Proxys müssen den Request also an den Origin-Server zustellen. Achtung: Proxys können so konfiguriert sein, dass sie diesen Header ignorieren, weswegen bei Seiten mit sensiblen Informationen diese Angabe gerne mit „no-store“ bzw. „no-store, must-revalidate“ kombiniert wird. Die Angabe no-cache selbst führt zu einer Validierung. Erst im Anschluss wird die gecachte Version der Datei verwendet. Diese Angabe kann für eine Abwärtskompatibilität mit HTTP1.0 auch im „Pragma“-Header verwendet werden. Ein gesetztes „Pragma: no-cache“ führt im Browser ebenfalls vor Ablauf der TTL zur Validierung.
no-store	Diese Angabe gibt an, dass Caches nichts über Request oder Response speichern dürfen. Diese Funktion wird oftmals in den „No-Cache“-Header hineininterpretiert. Um ein Caching komplett zu verhindern, ist dies die richtige Angabe.
proxy-revalidate	Bewirkt für Proxys-Caches das Gleiche wie must-revalidate, wird aber von Browsern ignoriert.
immutable	Gibt an, dass sich eine Ressource innerhalb der TTL nicht ändern wird. Eine bedingte Abfrage, welche zu einer Validierung führen würde, soll explizit nicht durchgeführt werden, auch wenn der User F5 drückt.

Tabelle 1: Cache-Control – mögliche Funktionen

Zusammenfassung

- Auf Fingerprints sollte in den URLs nicht verzichtet werden, mit Ausnahme von HTML natürlich. Wenn möglich, sollte diese Art der Versionierung nicht als Parameter stattfinden.
- Mit Verwendung von Fingerprints sollte eine möglichst lange Caching-Dauer einhergehen.
- Die Angabe max-age=0 macht beim HTML meistens Sinn.
- Sofern häufige Aktualisierungen vom User zu erwarten sind, kann die Verbindung mit der Angabe „immutable“ sinnvoll sein. Dabei sollten jedoch Fingerprint-URLs zum Einsatz kommen.
- Sofern private Daten übermittelt werden, sollte mindestens „no-cache“ Verwendung finden. Hier macht es Sinn, auf Nummer sicher zu gehen und das Caching komplett abzustellen.
- Soll ein Caching komplett unterbunden werden, muss mit „no-store“ gearbeitet werden, die Angaben „no-cache“ oder „max-age=0“ reichen nicht.
- Ein genauerer Blick auf die Angabe „stale-while-revalidate“ könnte sich lohnen.