10 Caching

GET is one of the most optimized pieces of distributed systems plumbing in the world.
– Don Box, Webservices-Guru und SOAP-Miterfinder

Die Unterstützung von Caching und das *conditional GET* sind zentrale Voraussetzung für die Skalierbarkeit der Webarchitektur: Nichts ist effizienter als Anfragen, die gar nicht erst gestellt werden. Bereits seit HTTP 1.0 gibt es daher Mechanismen, die diese Anforderung effizient umsetzen, und Caching wird auch in Fieldings Dissertation als wesentliches Element des REST-Stils definiert.

Die HTTP-Spezifikation definiert dazu zwei Modelle. Im Expirationsmodell liefert der Server in den Metadaten – also in HTTP-Headern – Informationen über die Gültigkeitsdauer einer Antwort. Innerhalb dieses Zeitraums kann der Client oder ein zwischen Client und Server angesiedelter externer Cache die Antwort für zukünftige Anfragen wiederverwenden, ohne den Server erneut kontaktieren zu müssen: Die Client/Server-Interaktion wird so komplett vermieden. Im Validierungsmodell fragt der Client (oder ein Cache) beim Server an, ob eine bereits vorhandene – gecachte – Antwort wiederverwendet werden kann: So wird zwar nicht die Interaktion selbst, aber unter Umständen die Übertragung der Daten vermieden. Beide Modelle lassen sich kombinieren: Ein Cache kann eine gemäß Ablaufdatum nicht mehr gültige Kopie validieren.

10.1 Expirationsmodell

Betrachten wir zunächst das Expirationsmodell. Ein Client fordert von einem Server die Repräsentation einer Ressource per GET an. Der Server liefert die Repräsentation zurück und setzt einen Header, der angibt, wie lange die Antwort gültig ist, in diesem Fall 60 Sekunden:

```
curl -i http://example.com/orders/

HTTP/1.1 200 OK
Cache-Control: max-age=60
Connection: Keep-Alive
```

```
Allow: HEAD, GET, POST
Date: Sun, 22 Feb 2009 09:57:32 GMT
Content-Type: text/html; charset=utf-8
Etag: "f0c78b3bc6bac6df9b36061d72827f77"
Content-Length: 1953

...
```

Der Server teilt dem Client dadurch mit, dass es sich innerhalb von 60 Sekunden nach dieser Anfrage nicht lohnt, die Repräsentation erneut abzufragen – es würde nur das gleiche Ergebnis zurückgeliefert. Der Client kann die Antwort also innerhalb dieses Zeitraums weiterverwenden. Anstelle des »max-age«-Wertes im »Cache-Control«-Header kann auch ein »Expires«-Header verwendet werden. Dieser gibt einen absoluten Zeitpunkt an – sozusagen ein Mindesthaltbarkeitsdatum für die Ressource:

```
Expires: Sun, 22 Feb 2009 11:57:32 GMT
```

In beiden Fällen lässt sich die Anzahl der Anfragen, die der Server verarbeiten muss, drastisch reduzieren. Gleiches gilt für die Wartezeiten im Client: Kann dieser die Anfrage aus seinem lokalen Cache bedienen, eventuell aus dem Hauptspeicher, wird sie um mehrere Größenordnungen schneller beantwortet. Noch offensichtlicher wird der Effekt beim Einsatz eines Cache-Servers, der zwischen dem eigentlichen Server (in HTTP-Terminologie: »Origin Server«) und den Clients sitzt. Dabei kann eine Vielzahl von Anfragen vom Cache beantwortet werden, ohne dass das Backend involviert werden muss.

Allerdings muss sich der Client nun mit einer unter Umständen nicht mehr ganz aktuellen Antwort zufriedengeben. Dies erscheint im Umfeld von Enterprise-Anwendungen zunächst völlig inakzeptabel. Wie so häufig allerdings lohnt es sich, die reflexartige Reaktion zu hinterfragen: Wirklich aktuell ist eine Antwort bei einer verteilten Anwendung ohnehin nie, insbesondere dann nicht, wenn Antworten eine Interaktion mit mehreren Systemen erfordern oder Datenbanken repliziert werden. Für viele Anfragen ist eine Gültigkeit von zumindest 5, 10 oder 30 Sekunden völlig akzeptabel. Ob sich das lohnt, hängt von der potenziellen Anzahl der Anfragen ab. In vielen Fällen ergibt sich die Gültigkeitsdauer bzw. das Ablaufdatum auch aus anderen Randbedingungen: Werden Daten mit einem anderen System zum Beispiel nur einmal am Tag ausgetauscht, etwa in einem nächtlichen Batchlauf, wissen Sie als Entwickler der Serveranwendung, dass sich eine erneute Anfrage bis zum nächsten Datenabgleich nicht lohnt.

Viele Antworten sind aber nicht nur einige Sekunden, Minuten oder Stunden, sondern unendlich gültig. Das gilt für alle Ressourcen, die einen klaren Zeitbezug haben, der in der Vergangenheit liegt, ebenso wie für solche, die sich aufgrund ihres Status nicht mehr ändern können. Einige Beispiele: Die Ressource für alle Bestellungen, die am 21.2.2009 zwischen 10 und 11 Uhr eingegangen sind; die Bestellung 4711, die bereits abgeschlossen ist und sich daher nicht mehr ändern kann; die Umsatzstatistik für 2007 nach Veröffentlichung des Jahresabschlusses

des Unternehmens. In solchen Fällen ist es vollkommen unsinnig, den Server mit immer neuen Anfragen zu belasten, die doch immer das gleiche Ergebnis zurückliefern.

Als Entwickler einer REST-Anwendung müssen Sie auf Basis der fachlichen Anforderungen und der nicht funktionalen Rahmenbedingungen entscheiden, ob und welche Gültigkeit Sie für Ihre Ressourcen angeben. Wenn Sie diesem Ansatz folgen und der Haupzugriffsweg auf Ihre Dienste das RESTful-HTTP-Interface ist, können Sie sich die Implementierung eines anwendungsspezifischen Cache sparen.

10.2 Validierungsmodell

Das Expirationsmodell ist sehr einfach, sowohl für denjenigen, der den Server implementiert, als auch für den Nutzer des Dienstes. In vielen Fällen ist es jedoch nicht ausreichend, da im Vorhinein schwer zu entscheiden sein kann, wie lange oder bis wann eine Ressource gültig ist. Für diese Fälle bietet HTTP als Alternative ein Validierungsmodell.

Bei dieser Variante entscheidet sich erst bei einem erneuten Zugriff, ob die gecachte Repräsentation einer Ressource noch gültig ist oder nicht – die Abfrage erfolgt als *bedingt*. Eine Möglichkeit dazu ist die Angabe eines Datums, seit dem sich die Ressource verändert haben muss, um für den Client interessant zu sein:

```
GET /orders/ HTTP/1.1
Host: example.com
If-Modified-Since: Sun, 22 Feb 2009 10:03:39 GMT
```

Der Server antwortet darauf entweder mit der vollständigen Repräsentation oder aber mit einer kurzen Information, dass sich nichts verändert hat:

```
HTTP/1.1 304 Not Modified
Cache-Control: private, max-age=0, must-revalidate
Date: Sun, 22 Feb 2009 10:54:23 GMT
Content-Type: text/html; charset=utf-8
Etag: "635cef4f49602b621b129e8fb27e11f8"
```

Es findet also eine Client/Server-Interaktion statt, allerdings ohne eine überflüssige erneute Übertragung der Daten.

Alternativ zur Validierung auf Basis eines Modifikationszeitpunkts kann der Client den Server auch fragen, ob eine lokal verfügbare Kopie der Repräsentation einer Ressource identisch zur aktuellen ist. Die Daten dazu zum Server zu übertragen wäre eine unlogische und verschwenderische Lösung. Stattdessen liefert der Server als Teil der initialen Antwort ein Entity-Tag *(ETag)*, einen Hashwert, der sich bei jeder Änderung an den Daten ebenfalls ändert; der Client kann diesen Wert als Teil einer bedingten Abfrage an den Server senden:

```
GET /orders/ HTTP/1.1
Host: example.com
If-None-Match: "635cef4f49602b621b129e8fb27e11f8"
```

Auch hier antwortet der Server entweder mit einer aktualisierten Repräsentation (und einem neuen ETag) oder einem »304 Not Modified«. (Wichtig: Wenn Sie den ETag-Mechanismus zur Unterstützung des Cachings verwenden, müssen Sie damit rechnen, dass Clients auch bedingte PUT-, POST- oder DELETE-Requests senden – siehe dazu Abschnitt 13.4.)

Die Bedenken, dass ein Client mit möglicherweise nicht mehr aktuellen Daten arbeitet, greifen beim Validierungsmodell nicht. Dafür findet jedoch in jedem Fall eine Kommunikation zwischen Client und Server statt.

Die Konsequenzen hängen dabei stark von der serverseitigen Implementierung ab. Wenn die Anwendung die vollständige Antwort ermittelt – z. B. durch diverse Datenbankzugriffe und komplexe Verarbeitungslogik – wird beim Validierungsmodell »nur« die übertragene Datenmenge reduziert. Ist es jedoch möglich, das (logische) Datum der letzten Modifikation oder ein ETag schneller zu ermitteln als die komplette Antwort, kann auch die Last auf dem Server deutlich reduziert werden. In diesem Fall spricht man von »Deep ETags«.

Dies ist im Rahmen einer generischen Lösung, also zum Beispiel in einem Webframework oder einem vorgeschalteten Proxy-Server, nicht möglich, da dazu anwendungsspezifisches Wissen notwendig ist. Es ist also Ihre Aufgabe als Entwickler einer REST-Anwendung, sich darüber Gedanken zu machen.

Sehen wir uns dazu zwei Beispiele an. Im ersten Fall liefert ein Finanzsystem Analyseinformationen über Bewegungen an den weltweiten Börsen. Die Daten, aus denen diese Informationen berechnet werden, bezieht es von diversen nachgelagerten Systemen, die ihre Informationen aktiv (also in einem »Push«-Modell) übermitteln. Ressourcenrepräsentationen werden dynamisch auf Basis der vorliegenden Daten berechnet und an die anfragenden Clients zurückgemeldet. Dabei wird vom eingesetzten Webframework automatisch ein Zeitstempel in den »Date«-Header gesetzt. Soll eine bedingte Anfrage eines Clients beantwortet werden, bei der dieser einen »If-Modified-Since«-Header gesetzt hat, kann das System überprüfen, ob von einem der nachgelagerten Systeme seit diesem Zeitpunkt überhaupt Daten geliefert wurden. Ist dies nicht der Fall, kann direkt der Statuscode »304 Not Modified« zurückgemeldet werden, ohne dass irgendeine Berechnung stattfinden muss.

Im zweiten Beispiel liefert der Server auf Anfrage des Clients eine komplexe Repräsentation einer Kundenakte zurück, die diverse Informationen über den Kunden aggregiert. Die Erstellung der Repräsentation besteht dabei aus zwei nachgelagerten Schritten: der Zusammenstellung der notwendigen Informationen und der anschließenden layoutkonformen Aufbereitung. Der Informationsgehalt hängt nur vom ersten Schritt ab; der Server kann also auf dieser Basis ein ETag berechnen und dieses mit dem vom Client übermittelten vergleichen. Ist es

identisch, kann der Layoutprozess übersprungen und ebenfalls der Statuscode 304 zurückgegeben werden. (In diesem speziellen Fall ist es möglicherweise ratsam, das ETag als »weak« zu kennzeichnen; mehr dazu in Abschnitt 10.5.)

Die beiden Modelle – Expiration und Validierung – können miteinander kombiniert werden: Anstatt dass ein Client oder ein Cache nach Ablauf des Gültigkeitszeitraums die Ressource neu anfordert, kann er eine bedingte Anfrage stellen und die Gültigkeit seiner lokalen Kopie verlängern. Eine einfache grafische Darstellung des Caching-Verhaltens liefert Ryan Tomayko in [Tomayko2008].

10.3 Cache-Topologien

Ein Cache kann als Bestandteil eines Clients implementiert sein; in diesem Fall liegt der Hauptnutzen in der Reduktion von Anfragen, die ein und derselbe Client an die gleichen Ressourcen stellt: Verschiedene Clients, die auf dieselben Ressourcen zugreifen, haben jeweils ihre eigene Kopie. Bei gemeinsam genutzten (geteilten oder »shared«) Caches greifen unterschiedliche Clients auf einen dem Server vorgelagerten, separaten Dienst zu. Daraus ergibt sich eine Reihe von unterschiedlichen Topologien für Caching im HTTP-Umfeld, die wir uns im Folgenden genauer ansehen.

Bei einem rein clientseitigen Caching werten die Clients die Cache-relevanten Header aus, die der Server ihnen übermittelt, bedienen sich aus ihrem lokalen Cache und stellen bedingte Anfragen. Spezifisch auf den Client zugeschnittene Repräsentationen können dabei ebenso gecacht werden wie verschlüsselte Daten. Der Nutzen hängt davon ab, wie häufig ein und derselbe Client wiederholt auf dieselbe Ressource zugreift (und natürlich davon, wie häufig sich die Daten ändern – aber das gilt für jede Topologie). Ein solcher clientseitiger, »privater« Cache ist in jeden Webbrowser eingebaut und standardmäßig aktiviert.

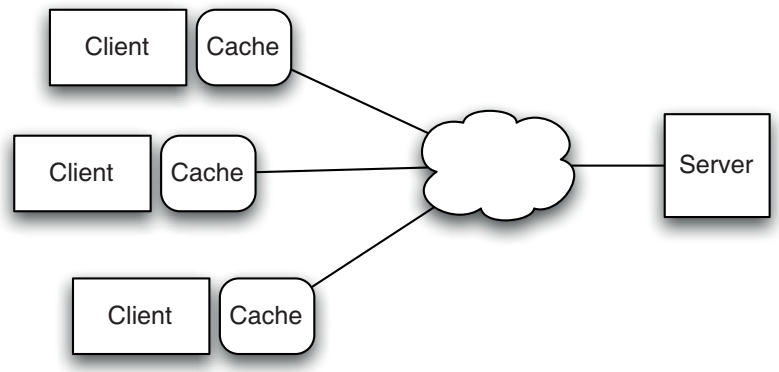

Abb. 10–1 *Rein clientseitiges Caching*

Mehrere Clients können sich einen Cache teilen; Anfragen werden bei Vorhandensein einer (noch) gültigen Antwort daraus bedient. Gecacht werden können dabei Antworten, die für alle Clients gleich sind (also ohne personalisierte Inhalte). Besonders vorteilhaft ist diese Topologie, wenn unterschiedliche Clients auf die gleichen Ressourcen zugreifen. Der Shared Cache selbst kann gecachte Repräsentationen beim jeweils angesprochenen Server validieren. Ein solches Modell ist vor allem in Unternehmen anzutreffen, die sämtliche Anfragen aus dem internen Firmennetz durch einen Proxy-Server leiten, der Caching implementiert. Beispiele für solche Server sind Squid oder Varnish (siehe Anhang C.3). Da der clientseitige Shared Cache aus historischer Sicht das erste breit unterstützte Modell war, bezeichnet man diese Topologie gemeinhin auch einfach nur als »Proxy-Cache«.

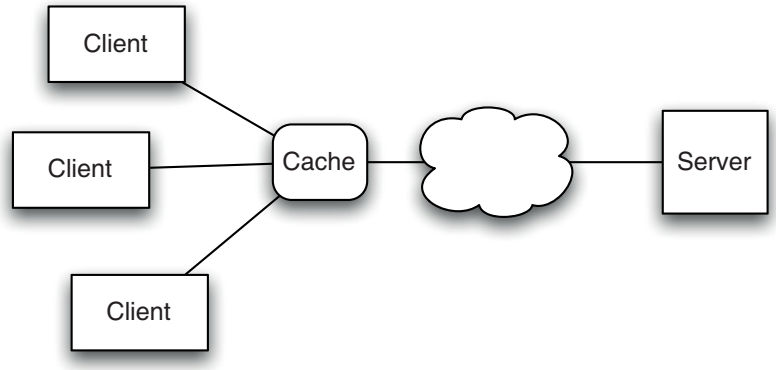

Abb. 10–2 *Clientseitiger Shared Cache*

Ein Cache kann auch auf der Serverseite angesiedelt werden. Sämtliche Anfragen an den Server laufen damit durch einen zentralen Cache; gecachte Antworten werden aus dem Cache bedient und der eigentliche Server (das Backend) damit entlastet. Dabei können sowohl personalisierte als auch übergreifend gültige Inhalte zwischengespreichert werden. Dieses Modell nennt man auch »Reverse Proxy Cache«, um den Unterschied zum clientseitigen Shared Cache deutlich zu machen.

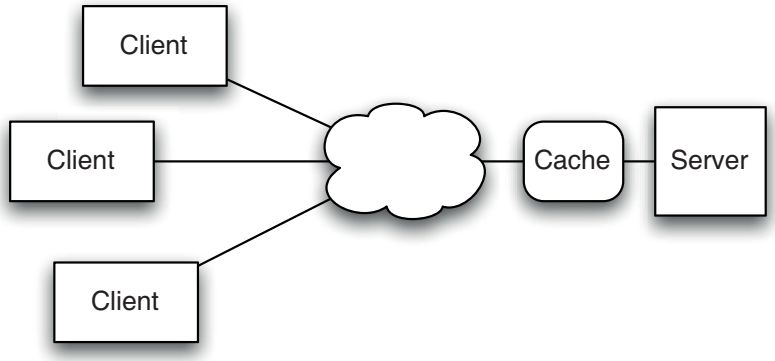

Abb. 10–3 *Server-Cache*

Die folgende Abbildung zeigt eine zufällig zusammengestellte, komplexe Cache-Topologie als Kombination aus den hier beschriebenen Möglichkeiten:

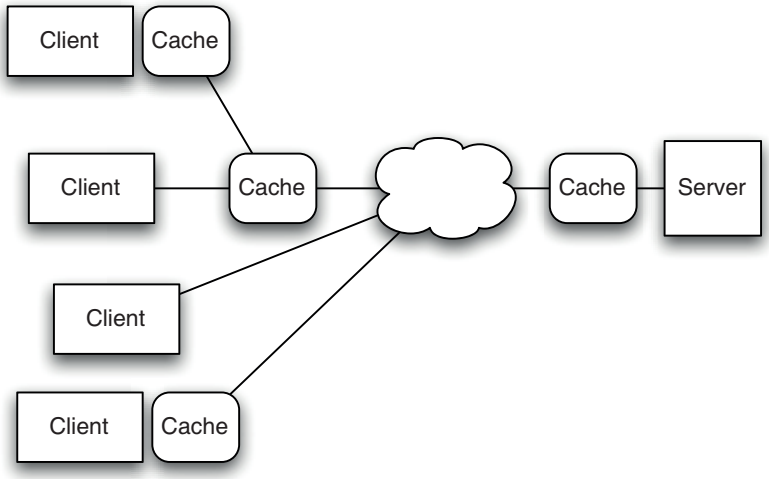

Abb. 10–4 *Kombination*

Anhand dieser Kombination lässt sich der größte Vorteil von Caching in HTTP illustrieren: Jede dieser Topologien lässt sich vollständig unabhängig von der Implementierung der Clients und Server aufsetzen, solange diese sich an die Cache-Vorgaben des HTTP-Protokolls halten. Ob Sie eine öffentlich zugängliche Webanwendung implementieren oder ein Enterprise-Szenario unterstützen müssen, ob die Clients Webbrowser oder andere Anwendungen sind: Wenn Sie die Cache-relevanten Header korrekt und effizient füllen bzw. auswerten, profitieren Sie am meisten von der darauf optimierten Webarchitektur.

10.4 Caching und Header

Clients und Server können das Caching beeinflussen, indem sie dafür relevante HTTP-Header verwenden. Der wichtigste dieser Header ist »Cache-Control«, der von Client und Server verwendet werden kann (den vom Server gesetzten Parameter »max-age« haben wir schon gesehen). Im Folgenden finden Sie eine Übersicht der wichtigsten Cache-Control-Parameter und anderer Caching-relevanter Header, getrennt nach Verwendung in Anfrage und Antwort.

10.4.1 Response-Header

Der wichtigste Header für die Kontrolle des Caching-Verhaltens in HTTP 1.1 ist »Cache-Control«. Über diesen kann der Server dem Client mitteilen, ob eine Antwort gecacht werden darf, und wenn ja, ob dies in einem von mehreren Clients genutzten Cache oder nur im Client selbst erfolgen darf:

Header-Beispiel	Bedeutung
Cache-Control: private, max-age=60	Die Antwort darf nur im Client-Cache gespeichert werden (und ist 60 Sekunden gültig)
Cache-Control: public, max-age=60	Verwendung von Shared Caches ist erlaubt (public ist hier Voreinstellung)
Cache-Control: no-cache	Die gesamte Antwort darf gecacht werden, muss vor der Auslieferung an den Client aber neu validiert werden
Cache-Control: no-cache="Set-Cookie"	Die Antwort darf gecacht werden, mit Ausnahme des genannten Headers

Tab. 10–1 *»Cache-Control«-Header in Serverantworten*

Es gibt eine ganze Reihe weiterer Direktiven, die im »Cache-Control«-Header vom Server als Teil einer Antwort gesetzt werden dürfen:

- Die Direktive »max-age=...« ist uns bereits diverse Male begegnet und gibt an, wie lange eine Antwort gültig ist und von einem Cache ohne erneute Validierung zwischengespeichert werden soll.
- Mit »s-maxage=...« können Sie die über den Expires-Header oder max-age gesetzte Zeit für einen Shared Cache anders festlegen.
- Über »must-revalidate« können Sie erzwingen, dass eine gecachte Antwort auf gar keinen Fall ohne eine erneute Validierung erfolgt, auch wenn ein Cache anders konfiguriert ist oder der Client die »max-stale«-Direktive (s.u.) verwendet.
- »proxy-revalidate« funktioniert wie »must-revalidate«, bezieht sich allerdings nur auf gemeinsam genutzte Caches.

- Mit »no-store« (verwendbar sowohl in der Anfrage als auch der Antwort) signalisiert der Sender, dass die in der Nachricht enthaltenen Daten nicht zwischengespeichert werden sollen, was z. B. für vertrauliche Informationen sinnvoll sein kann.

Neben den bereits erwähnten »Date«-, »Expires«- und »ETag«-Headern ist außerdem noch der »Vary«-Header Caching-relevant. So könnten Sie zum Beispiel aus Ihrem Server abhängig vom Typ des Clients unterschiedliche Inhalte ausliefern und diese durch einen »Vary«-Header bekannt geben:

```
Vary: User-Agent
```

Für einen Cache bedeutet das, dass er einem Client, der den User-Agent-Header auf »A« setzt, keine gecachte Repräsentation senden darf, die für User-Agent »B« zurückgeliefert wurde.

10.4.2 Request-Header

Der Client kann in seiner Anfrage ebenfalls in Cache-Control-Direktiven definieren, ob und unter welchen Bedingungen er Daten aus Caches akzeptieren möchte:

- »max-age=… « definiert das maximale Alter (in Sekunden), das eine Antwort haben darf. Mit max-age=0 kann ein Client erzwingen, dass alle Caches auf dem Weg zum Server ihre Einträge validieren und gegebenenfalls aktualisieren.
- Über »min-fresh=… « legt der Client fest, wie lange eine Antwort mindestens noch gültig sein muss.
- Mit »max-stale=...« kann der Client angeben, dass er auch mit nicht mehr aktuellen Antworten zufrieden ist.
- Sendet ein Client »Cache-Control: no-cache«, wird die Anfrage auf jeden Fall erneut zum Server geleitet und nicht aus einem Cache bedient.
- Mit »only-if-cached« kann der Client angeben, dass er die Antwort nur dann möchte, wenn sie gecacht ist. Dieser Header wird äußerst selten verwendet.

Zusätzlich kann der Client die bereits beschriebenen »If-Modified-Since«- und »IfNone-Match«-Header verwenden, um konditionale Abfragen zu formulieren.

10.5 Schwache ETags

In unserer Betrachtung von ETags haben wir noch ein Detail ignoriert – die genaue Definition, wann zwei Repräsentationen einer Ressource dasselbe ETag haben sollten. In der HTTP-Spezifikation werden dazu zwei Varianten unterschieden:

Starke *(»strong«)* **ETags** beziehen sich auf die vollständige Repräsentation. Nur wenn zwei Repräsentationen Byte für Byte identisch sind, darf ein Server für