

Webentwicklung

Querschnittsthema: Optimierung

Inhalt dieser Einheit

0. **Optimierung:** Was und Warum?

1. **Ladezeiten:** Anwenderperspektive & Grundlagen

2. **Kritischer Request-Pfad**

3. **Request-Zahlen minimieren**

- Lazy Loading
- **HTTP-Caching:** Expiration & Validation
- Bündeln: CSS & JS, Grafiken
- HTTP/2: Server-Push

4. **Datenvolumen reduzieren**

- Minify & Dateiformate
- Payload-Komprimierung, Kontextabh. Bilder

5. **Nachbemerkungen**

Optimierung

Was und Wofür?

Web-Entwicklung: Bisher

- Ganz allgemein: Verteiltes System implementieren
 - damit Anwender *Ziele* erreichen können
 - Zerlegung in (mind.) 2 Komponenten: **Frontend & Backend**
 - Komponenten kommunizieren über HTTP
 - Randbedingung: *Frontend* läuft im Browser → HTML, CSS, JS

Web-Entwicklung: Wirtschaftlich

- **Anwender (A)**

1. *Ziel erreichen*

- funktionale Anforderungen

2. *Schnell, zufrieden, angenehm*

- nicht-funktionale Anf.
- Usability

- **Betreiber (B)**

1. *Anwender in*

Wertschöpfungskette

- meistens: **A**-Zielerreichung
- oft: **A**-Zufriedenheit

2. *Entwicklungs- und Wartungskosten*

- Personenstunden pro Feature/Fix

3. *Betriebskosten*

- Netzbandbreite, CPU-Zeit

Einordnung bisheriger Themen

- **Anwender (A)**

1. *Ziel erreichen* ✓

- PHP, JS (Funktionalität)
- HTML, HTTP (Darstellung)

2. *Schnell, zufrieden, angenehm* (✓)

- Gestaltung, CSS, JS
- Security
- AJAX

- **Betreiber (B)**

1. *Anwender in*

Wertschöpfungskette

- HTTP

2. *Entwicklungs- und Wartungskosten* (✓)

- Wiederverwendung (Komponenten, WebServices)
- Frameworks in Front- und Backend (jQuery, Bootstrap, Symfony)

3. *Betriebskosten* ✗

- -

Motivation: State of the Web

- 46% von 7,4 Mrd. Menschen online
 - 93% über Mobilgeräte
 - Mittlerer Anschluss: ~ 7MBit/s
 - 1-13 Arbeitstunden für 500 MB-Paket
- Websites:
 - Mittel: 3 MB; davon 1.7 MB Bilder, 400 KB JavaScript
- Mobilgeräte:
 - JS parsen & kompilieren: 2-5× langsamer

Quelle: <https://medium.com/@fox/talk-the-state-of-the-web-3e12f8e413b3>

Offene Punkte (Ausschnitt)

- **Anwender (A)**

- 2. *Schnell, zufrieden, angenehm*

- 1. Lade-/Wartezeiten
 - 2. Datenvolumen

- **Betreiber (B)**

- 2. *Entwicklungs- und Wartungskosten*

- 1. Aufwand CSS-Coding
 - 2. Aufwand JavaScript-Coding

- 3. *Betriebskosten*

- 1. Netzbandbreite
 - 2. CPU-Zeit

Optimierung

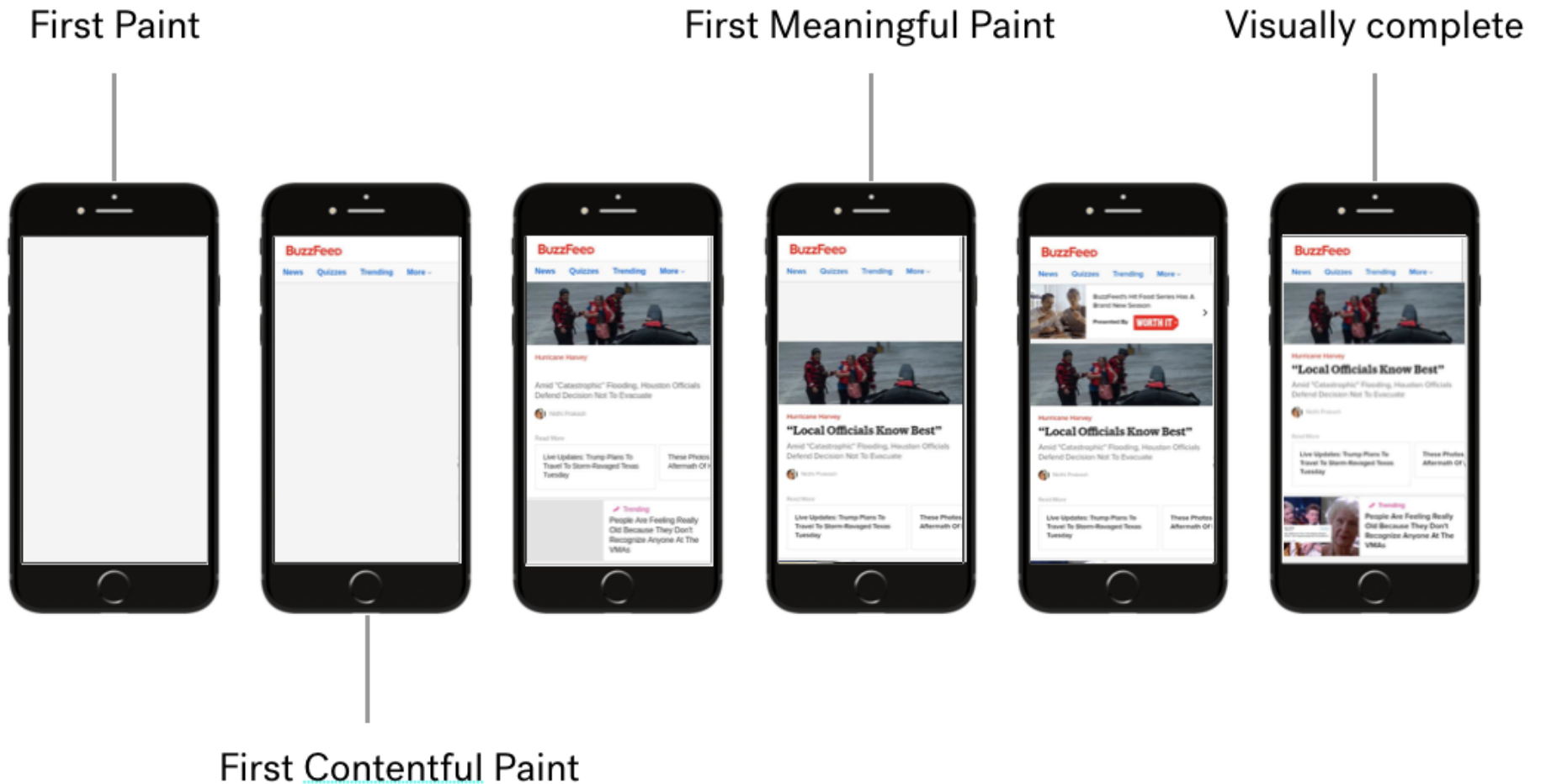
- Vielzahl an Möglichkeiten zur Optimierung
- Überblick:
 - Welche Technik setzt wo an, welches Problem soll gelöst werden?
 - Was sind die Konsequenzen daraus?
- Praktisch immer:
 - erhöhte Komplexität
 - Wechselwirkung mit anderen Problemen → Trade-Offs
- Heute: Fokus auf **Betrieb**
 - **Anwender**: Ladezeiten und Volumen
 - **Betreiber**: Datenvolumen und CPU-Zeit
- Nächste Woche: **Entwicklung und Wartung**

Ladezeiten

Anwenderperspektive

Kritischer Abfrage-Pfad

- kritische Abfrage: für Darstellung “above the fold”



Quelle: <https://medium.com/@fox/talk-the-state-of-the-web-3e12f8e413b3>

Relevante Zeitpunkte

- Aus Anwendersicht
 - **First Paint:** Wechsel von Weiß zu *etwas*
 - (First Contentful Paint: Logo u.ä. sichtbar)
 - **First Meaningful Paint:** Text, Bilder, Elemente sichtbar
 - **Visually Complete:** Inhalt im Viewport ist vollständig
 - **Time to Interactive:** Bereit zur Interaktion

Quelle: <https://medium.com/@fox/talk-the-state-of-the-web-3e12f8e413b3>

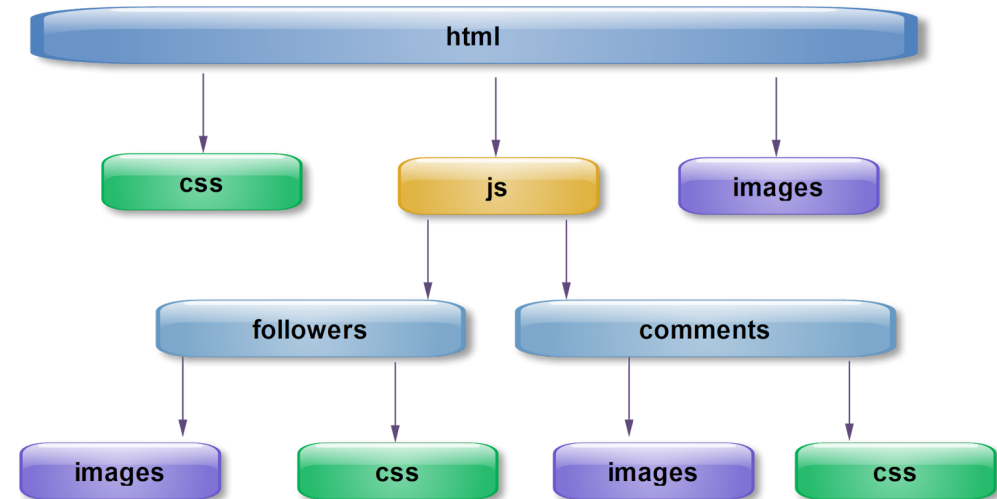
Ladezeiten

Technische Grundlagen

Ursache 1: Vielzahl an Requests

- Typische Folge von Anfragen:

- eingebundene Bilder
- eingebundene Stylesheets
 - verweisen auf Grafiken und Schriftarten
 - `@imports` weiterer Stylesheets
- eingebundene JavaScript-Dateien
 - laden jeweils weitere Ressourcen (Bilder, JavaScripts, ...)



Ursache 1: Vielzahl an Requests

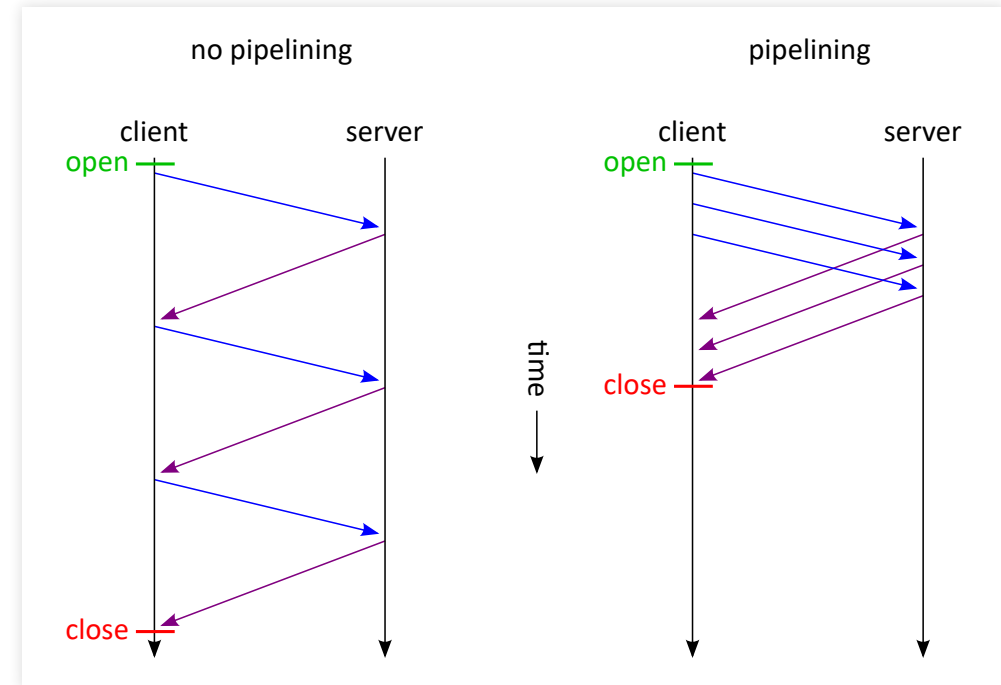
- viele Requests dauern länger als wenige
 - (zeit.de: insgesamt ca. 160 HTTP-GET-Requests)
- verschiedene Bauarten von Seiten:
 1. externe Elemente, die selbst externe Elemente laden
 - werden *zwangsläufig* in Stufen geladen
 2. Dokumente mit vielen externen Elementen
 - viele Bilder, mehrere Stylesheets
 - *im Prinzip* parallel ladbar
 - aber ...

HTTP: Parallele Abfragen

- pro Host: 4 bis 8 TCP-Verbindungen (früher: 2)
 - konkretes Limit ist Implementierungsdetail des Browsers
 - (Hintergrund: jede TCP-Verbindung kostet Server-RAM)
- pro Verbindung: Request, Response, Request, ...
 - d.h. 20 Ressourcen vom gleichen Server:
 - höchstens 8 Ressourcen gleichzeitig
 - Nr. 9 wird erst geladen, wenn erste Verbindung frei wird
- Flaschenhals:
 - Ressourcen werden nacheinander geladen
 - selbst wenn alle Ressourcen *im Prinzip* parallel ladbar wären

Beschleunigung der Abarbeitung

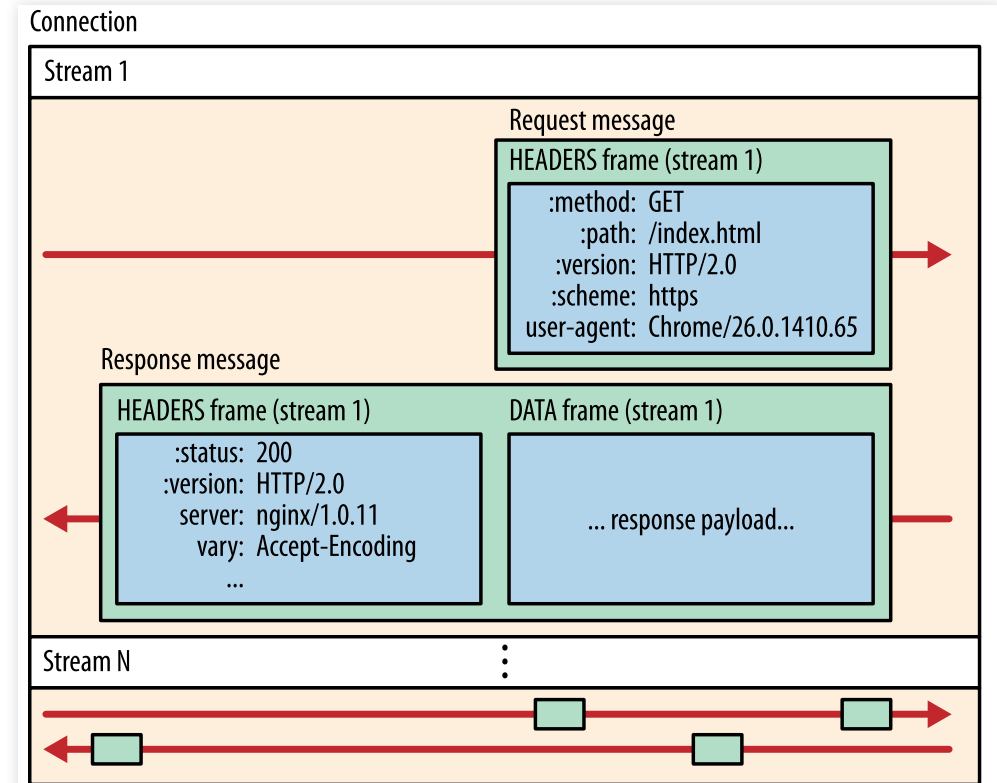
- HTTP/1.1: Pipelining
 - Idee: Client wartet nicht auf Antwort
 - TCP-Verbindung für mehrere Anfragen nutzbar
- Praktisch wenig genutzt/unterstützt



Quelle: <https://de.wikipedia.org/wiki/HTTP-Pipelining>

HTTP/2: Multiplexing

- Idee:
 - mehrere Streams
 - über eine TCP-Verbindung
- Antworten:
 - unbestimmte Reihenfolge
- macht TCP-Verbindungslimit irrelevant



Ursache 2: Große Datenmengen

- Moderne Websites: viele tendenziell große Elemente
 - (zeit.de: ca. 5 MByte für die Startseite)
 - hochauflösende Fotos
 - JavaScript-Bibliotheken
 - Schriftarten
- viele Daten dauern länger als wenige

Alles hängt zusammen

- **Anwender (A)**
 - Daten → Ladezeit
 - #Anfragen → Server-Last → Ladezeit
- **Betreiber (B)**
 - #Anfragen → CPU-Last (und RAM-Last)
 - Daten → Bandbreite
 - (Anfragen als Multiplikator)
- **Also: Drei Strategien**
 1. Kritischen Request-Pfad berücksichtigen
 2. Request-Zahlen minimieren
 3. Daten-Volumen minimieren

Kritischer Request-Pfad

Wichtiges zuerst

Lade-Prioritäten

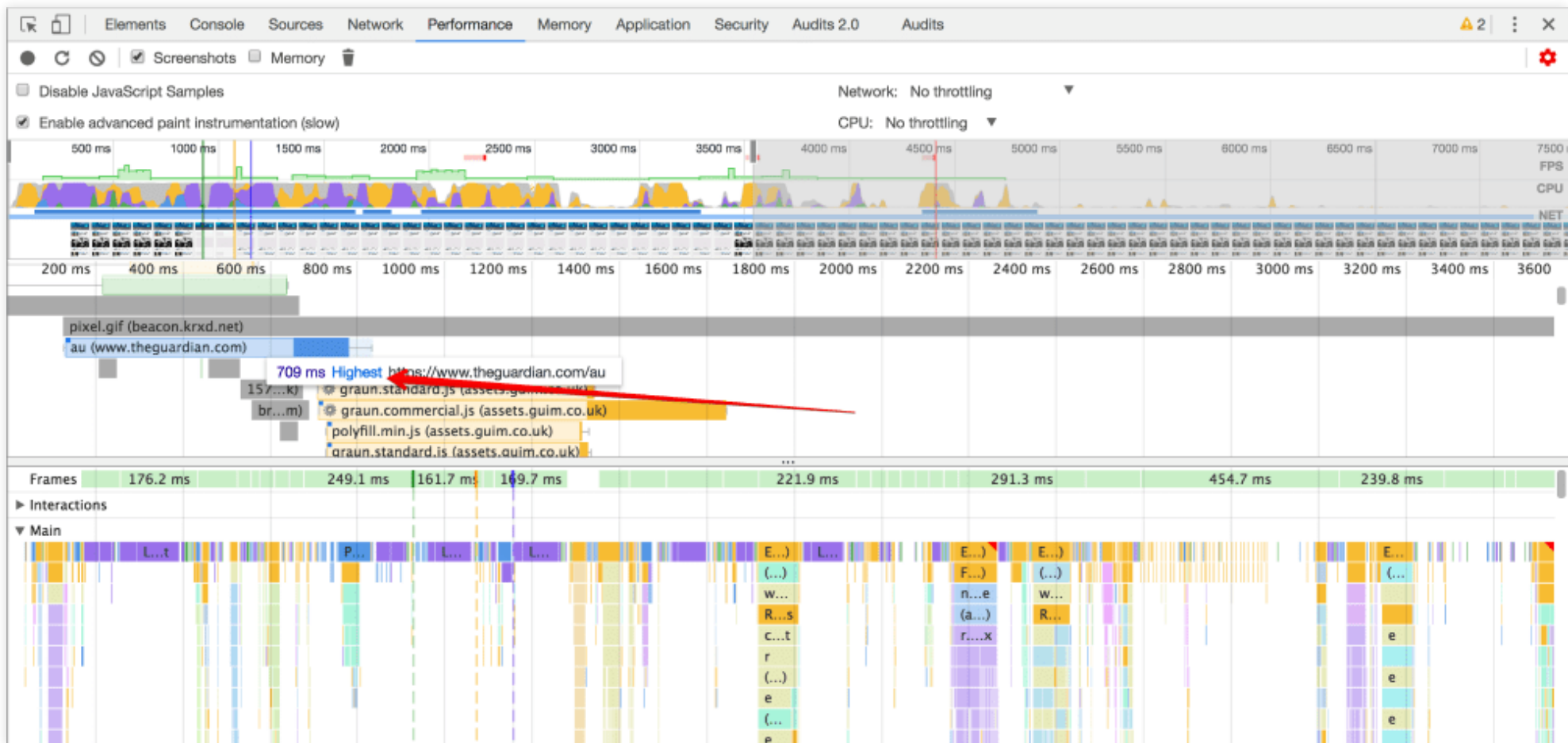
- Browser: komplizierte Regeln zur Lade-Reihenfolge
 - [Details zu Chrome](#), ungefähr so:
 - HTML: *höchste* Priorität
 - Style: *höchste* Priorität (`@import` nach blockierenden Skripten)
 - Bilder: *mittel/niedrig*, je nach Viewport (“above/below the fold”)
 - AJAX: *hoch*
 - Scripts: *hoch/mittel* via `script`, niedrig mit `async/defer`
 - Schriftarten: *hoch*, oft spät wg. `@import`
- kritische Ressourcen identifizieren und deklarieren

```
<link rel="preload" href="font.woff" as="font">  
<!-- You might not know it yet, but we're going to need this. -->
```

- [Preload](#), erklärt beim Mozilla Developer Network

Kritischen Pfad identifizieren

- Entwickler-Tools (F12): *Messen, messen, messen*



Quelle: <https://css-tricks.com/the-critical-request>

Request-Zahlen minimieren

Ansätze zur Request-Minimierung

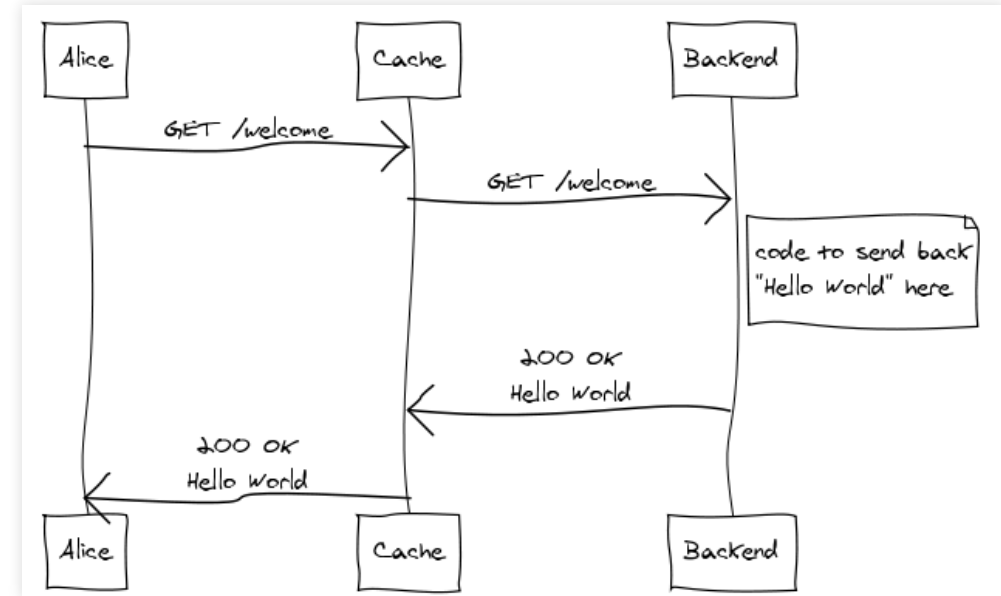
- Request-Vermeidung:
 - Lazy Loading
 - **HTTP-Caching**: Expiration & Validation
- Ressourcen bündeln
 - CSS und JavaScript: Bundle
 - Bilder: CSS-Sprites
- Verbesserung in HTTP/2
 - Server Push

Request-Vermeidung: Lazy Loading

- der schnellste Request:
 - einer, der man *nicht* macht
- Lazy-Loading mit JavaScript:
 - `img`-Elemente ohne `src`-Attribut (oder nur mit Vorschaubild)
 - erst setzen wenn `img`-Element in sichtbaren Bereich scrollt
 - Implementierungen in JavaScript:
 - <https://appelsiini.net/projects/lazyload/>
 - <http://luis-almeida.github.io/unveil/>
 - <http://jquery.eisbehr.de/lazy>

HTTP-Caches

- Idee:
 - dynamische Ressourcen nicht für jeden Request komplett neu
 - Ergebnis von Anfrage n speichern, für Anfrage $n+1$
- Cache: Knoten zwischen Front- und Backend
 - als Client-Komponente (im Browser)
 - eigener Server ([Varnish](#), [Squid](#))
 - Dienstleister ([Akamai](#))
 - Symfony-Komponente: [HTTP-Cache](#)

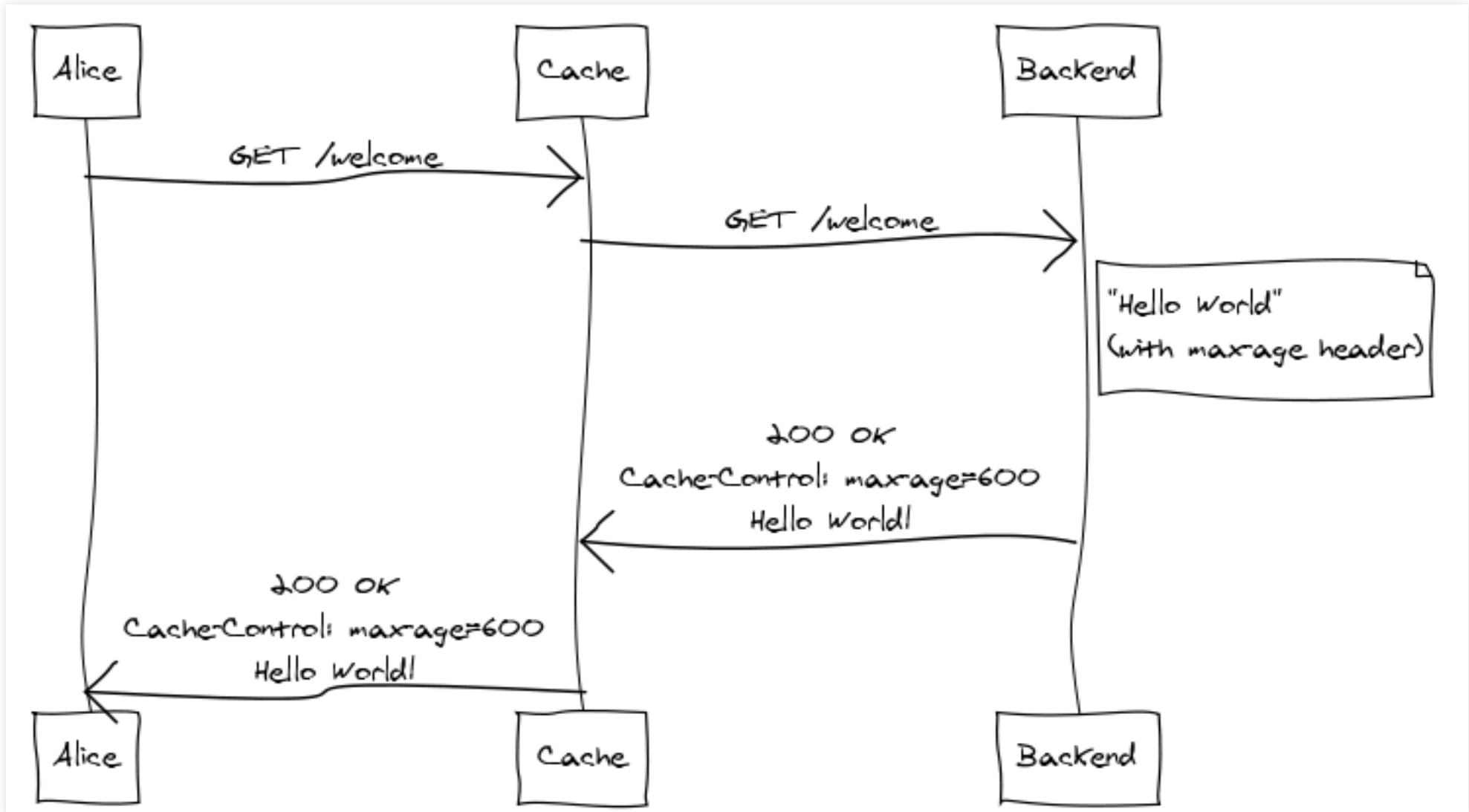


Quelle: <https://tomayko.com/blog/2008/things-caches-do>

HTTP-Caches: Zwei Modelle

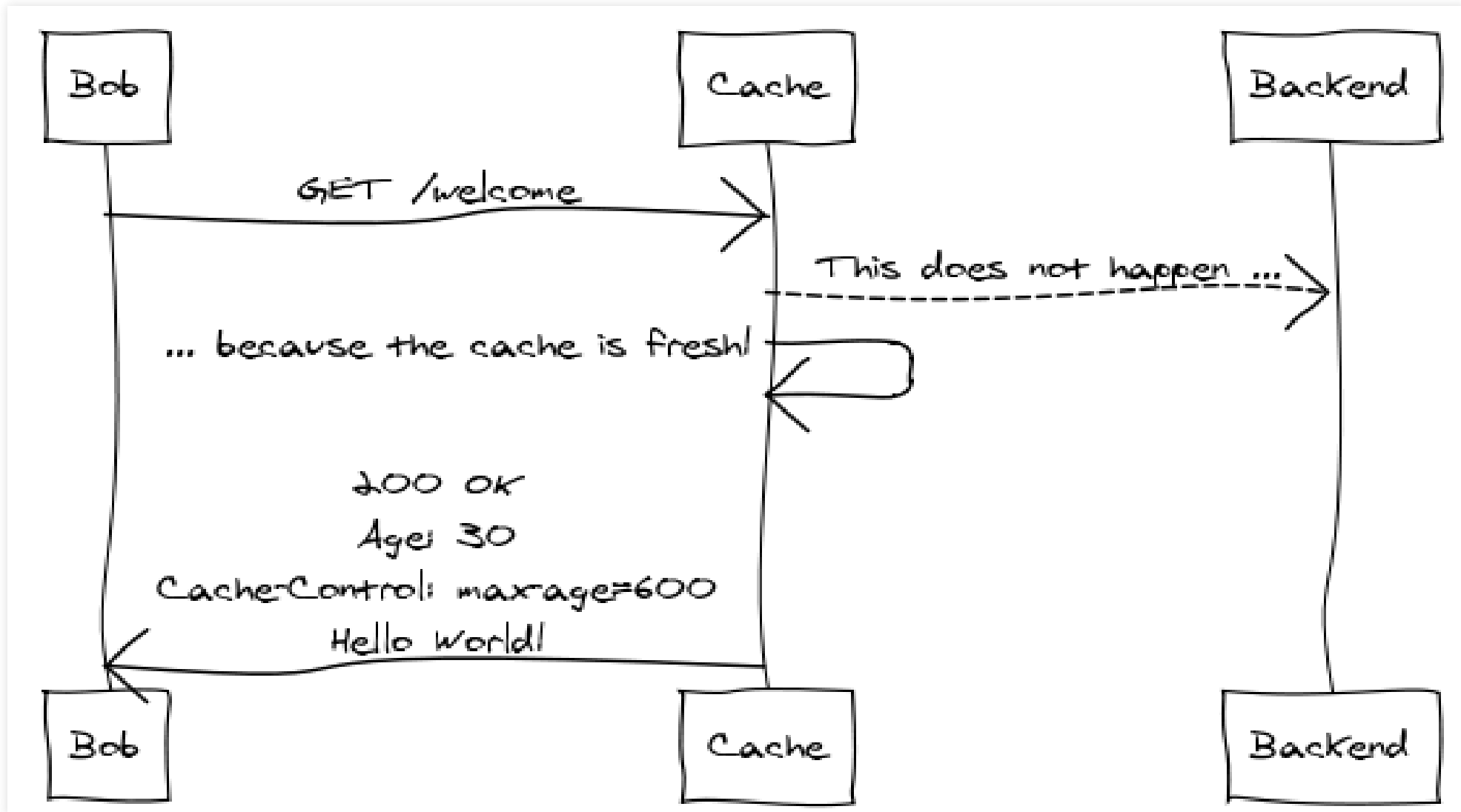
- Schwierig: Ist der Cache-Inhalt noch gültig?
- Zwei Modelle
 - **Expiration:**
 - Backend definiert Zeitspanne, wie lange Einträge “frisch” sind
 - **Validation:**
 - Backend bestimmt, ob neue Response nötig ist
- Informationen in beiden Modellen:
 - HTTP-Header in Requests und Responses (Details folgen)
 - Grundlage: [↗ RFC 7234 - Caching](#) (von 2014)

HTTP-Cache: Expiration (1/2)



Quelle: <https://tomayko.com/blog/2008/things-caches-do>

HTTP-Cache: Expiration (2/2)



Quelle: <https://tomayko.com/blog/2008/things-caches-do>

HTTP-Cache: Expiration

- In Symfony:

```
/**
 * @Route("/overview")
 * @Cache(smaxage=3600)
 */
public function showOverview() { /* aufwändiger Code */ }
```

- Ablauf

1. Anfrage:

- Cache ist leer, Backend wird angesprochen, Controller aufgerufen
- Cache speichert Response mit 1h Haltbarkeit, liefert an Client

2. Anfrage (innerhalb der Haltbarkeit):

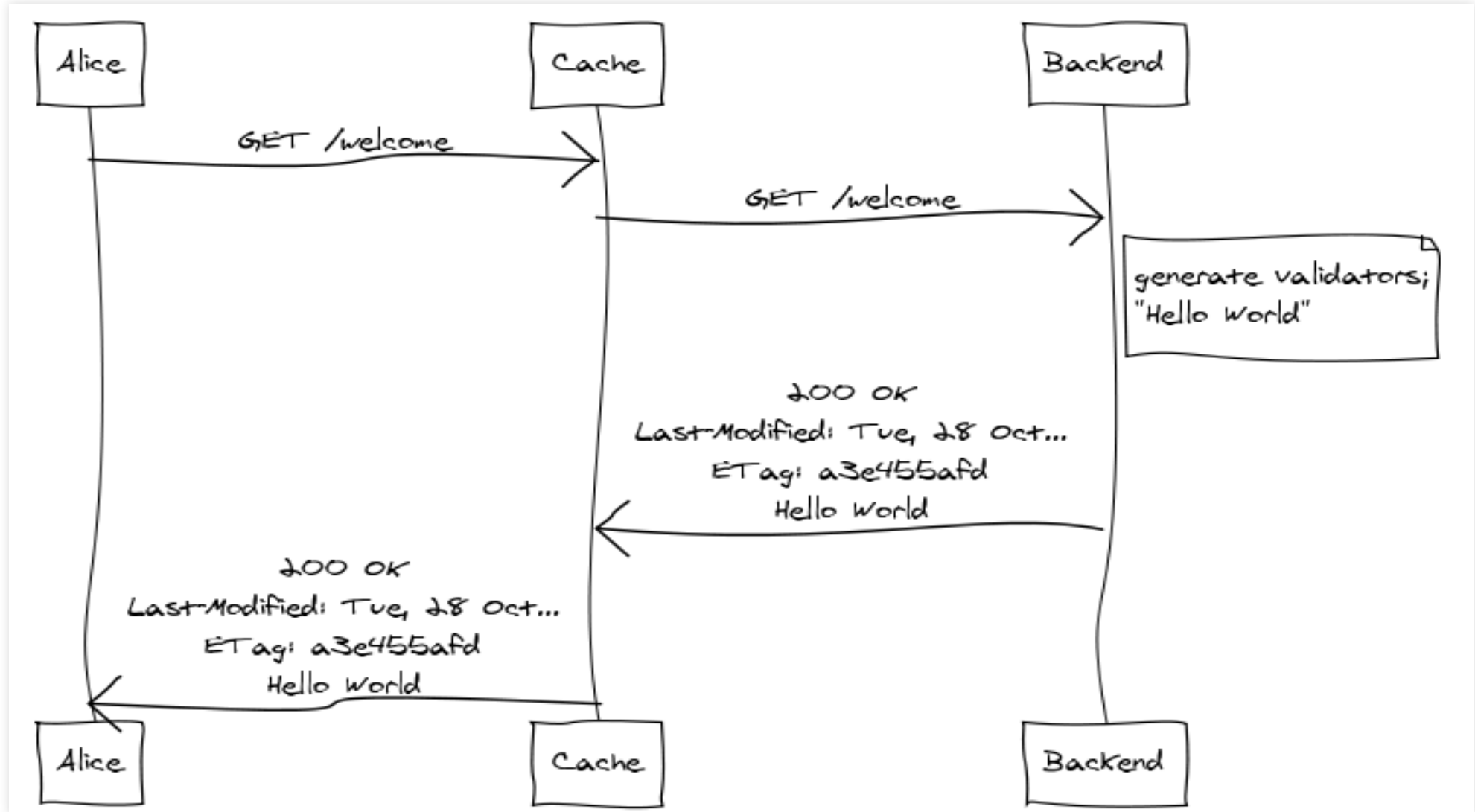
- Cache-Inhalt ist "fresh"
- Backend wird *nicht* aufgerufen, Cache liefert Response selbst aus

3. Anfrage (nach Haltbarkeit):

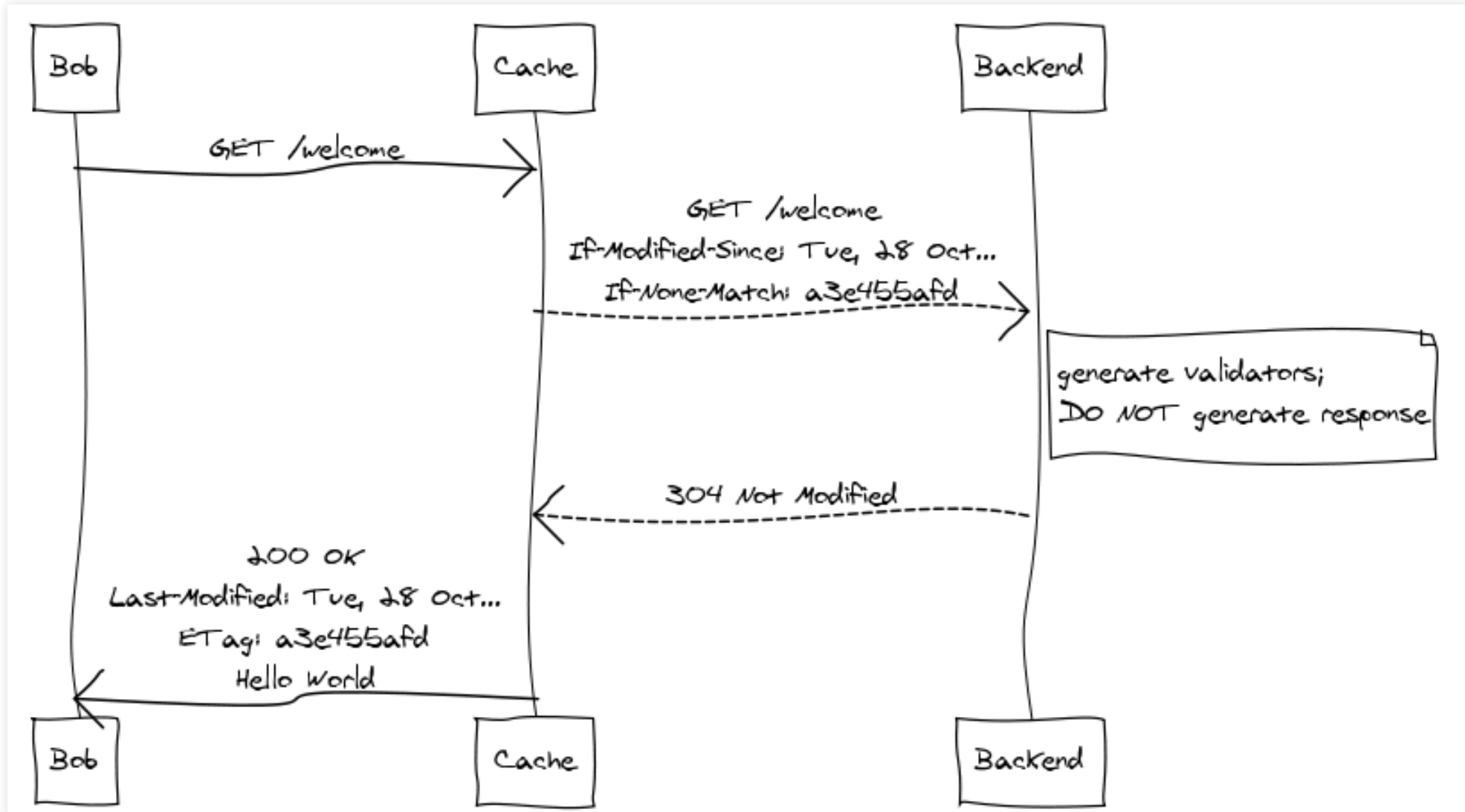
- Cache-Inhalt ist "stale", wie 1. Anfrage

Quelle: <https://symfony.com/.../annotations/cache.html#http-expiration-strategies>

HTTP-Cache: Validation (1/2)



HTTP-Cache: Validation (2/2)



HTTP-Cache: Validation

- In Symfony:

```
/**
 * @Route("/show/{id}")
 * @Cache(lastModified="post.getUpdatedAt()")
 */
public function showPost(Post $post) { /* aufwändiger Code */ }
```

- Ablauf

1. Anfrage:

- Cache ist leer, Backend wird angesprochen, Controller aufgerufen
- Cache speichert Response mit Datum, liefert an Client

2. Anfrage (ohne Änderung):

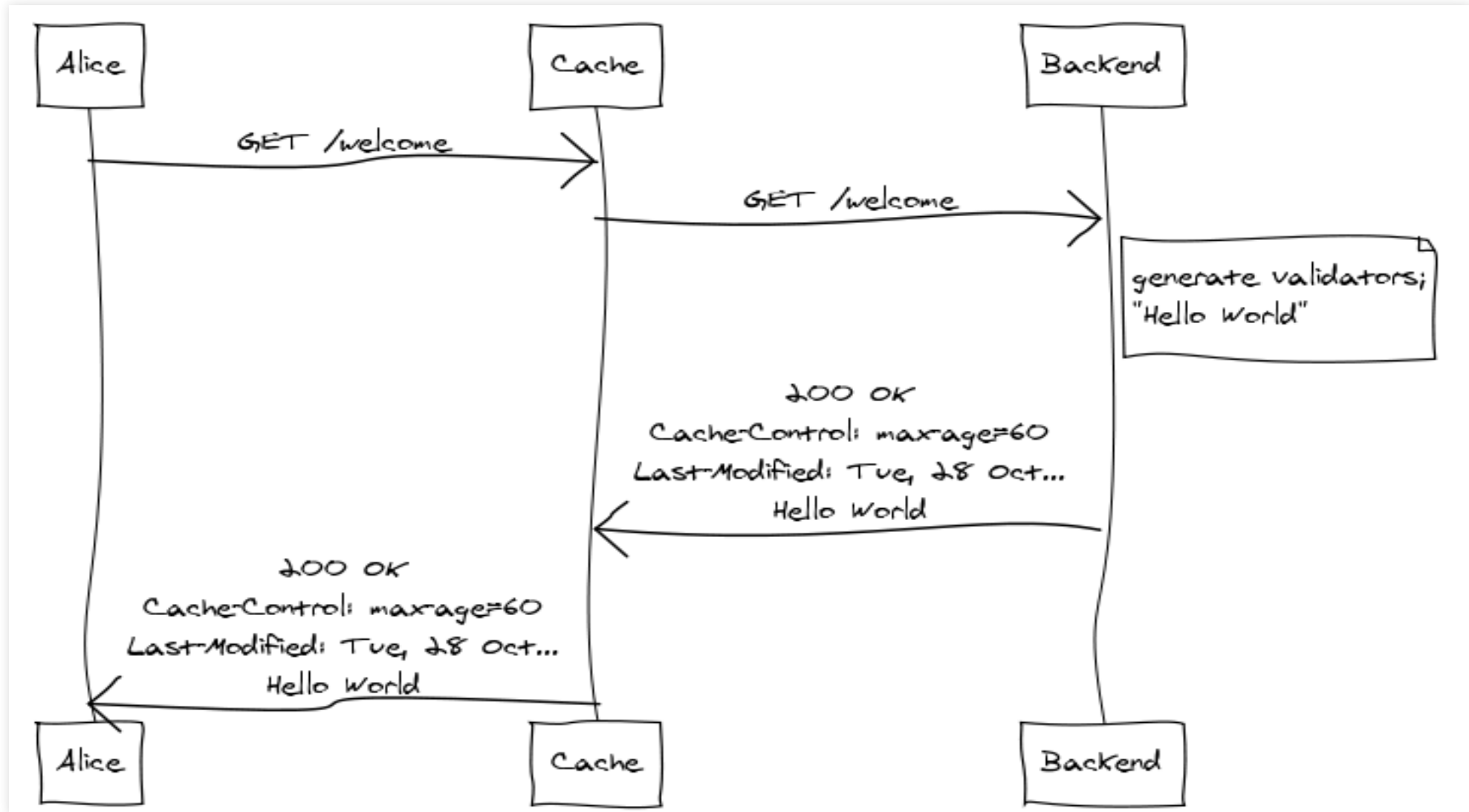
- Anfrage an Backend, Routing, Aufruf `post.getUpdatedAt()`
- Controller wird *nicht* ausgeführt, HTTP `304`, Cache liefert Response

3. Anfrage (nach Änderung):

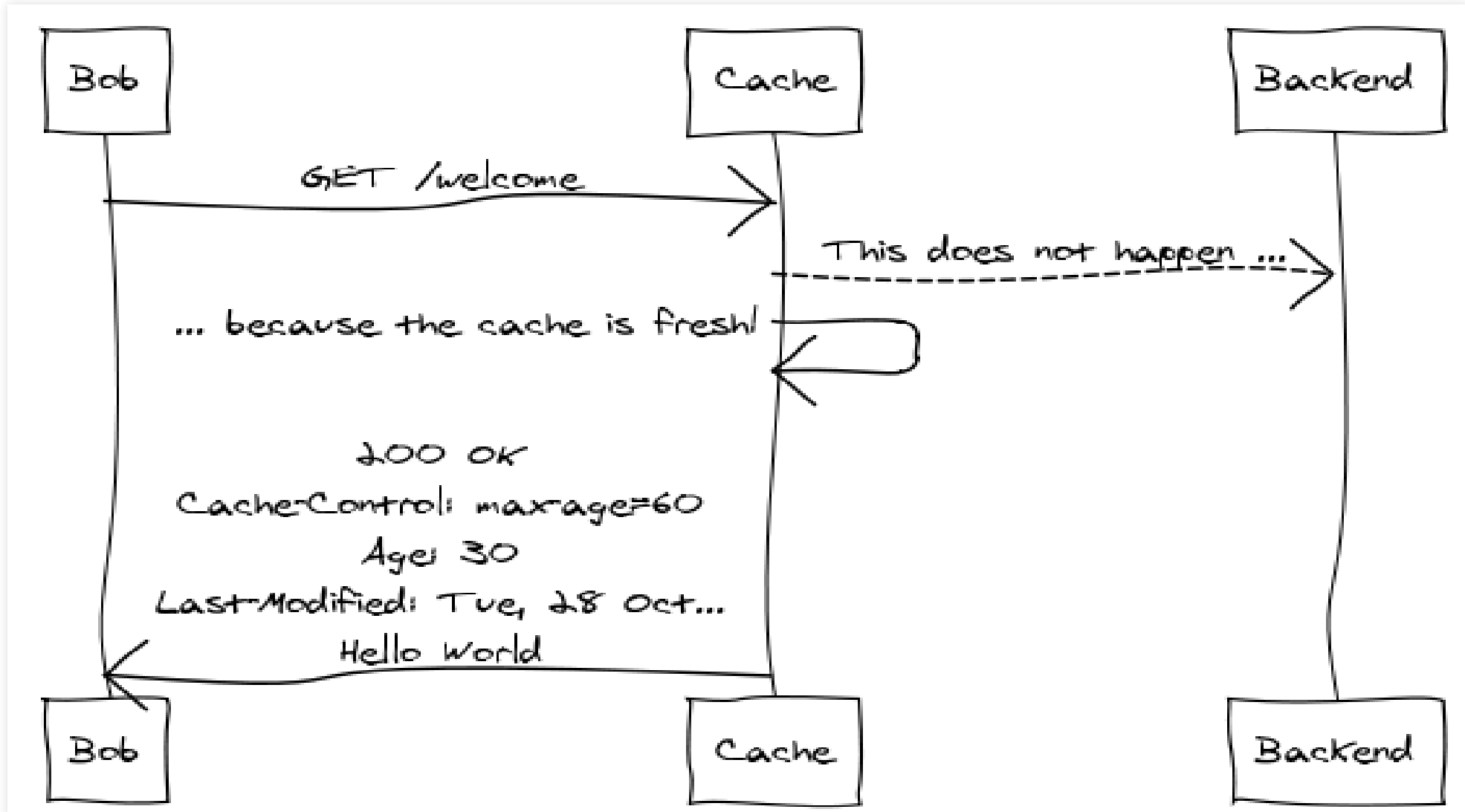
- Controller wird ausgeführt, Cache ungültig, Antwort vom Backend

Quelle: <https://symfony.com/.../annotations/cache.html#http-validation-strategies>

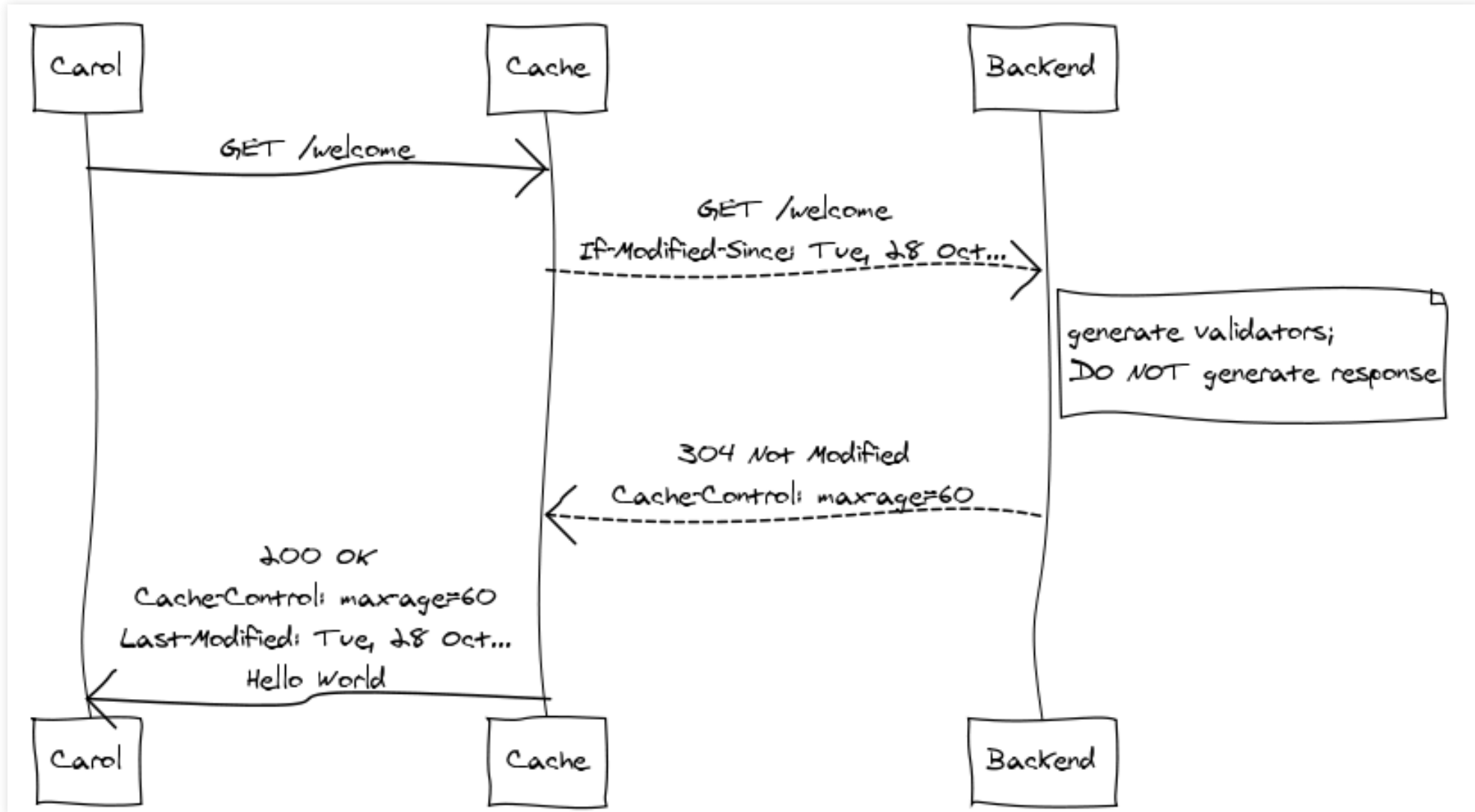
HTTP-Cache: Kombination (1/3)



HTTP-Cache: Kombination (2/3)



HTTP-Cache: Kombination (3/3)



HTTP-Cache: Kombination

- In Symfony:

```
/**
 * @Route("/show/{id}")
 * @Cache(smaxage=3600, lastModified="post.getUpdatedAt()")
 */
public function showPost(Post $post) { /* aufwändiger Code */ }
```

- Ablauf

1. Anfrage: Controller wird aufgerufen, Response → Cache
2. Anfrage (innerhalb von 1h):
 - Backend wird nicht aufgerufen
3. Anfrage (nach der 1h):
 - Backend wird aufgerufen und prüft Gültigkeit (`post.getUpdatedAt()`)
 - immer noch gültig: Controller nicht aufrufen, Cache erneuern
 - ungültig: Controller aufrufen, Cache erneuern

Text-Dateien bündeln

- Textbasierte Ressourcen bündeln: CSS und JavaScript

- Statt:

```
<link rel="stylesheet" href="css/base.css">  
<link rel="stylesheet" href="css/details.css">  
<!-- noch weitere Stylesheets -->
```

- Besser:

```
<link rel="stylesheet" href="css/bundle.css">  
<!-- beinhaltet alle Regeln -->
```

- (analog für JavaScript-Ressourcen)

- Entwicklungsprozess:

- Aus Entwickler-Sicht: *einzelne* Dateien
- Vor Deployment: Erstellen je eines CSS- und JavaScript-Bundles
 - programmatisch: [↗ Webpack](#)

Webpack in Symfony

- [Webpack](#) ist eine JavaScript-Bibliothek
- Integration in Symfony: [Webpack Encore](#)
 - Installation (Entwicklungsrechner)

```
$ composer require encore  
$ yarn install # Yarn ist wie Composer für JavaScript
```

- Konfiguration (im Symfony-Wurzelverzeichnis)

```
// webpack.config.js  
Encore  
    .setOutputPath('public/build/')  
    .addEntry('bundle', ['./assets/main.js', './assets/details.js']);
```

- Verwendung (Entwicklungsrechner)

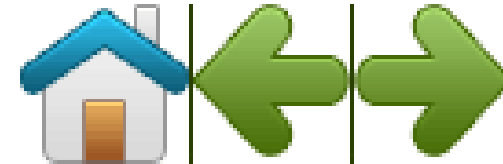
```
$ yarn run encore dev # erzeugt 'public/build/bundle.js'
```

- `bundle.js` beinhaltet `main.js` und `details.js`
- kann normal in HTML-Templates verwendet werden

Quelle: <https://symfony.com/doc/current/frontend/encore/simple-example.html>

Mehrere Bilder in CSS-Sprites

- Gleiche Idee für Grafiken:
 - nicht Einzelbilder, sondern eine Gesamtdatei
 - “Zuschneiden” per CSS

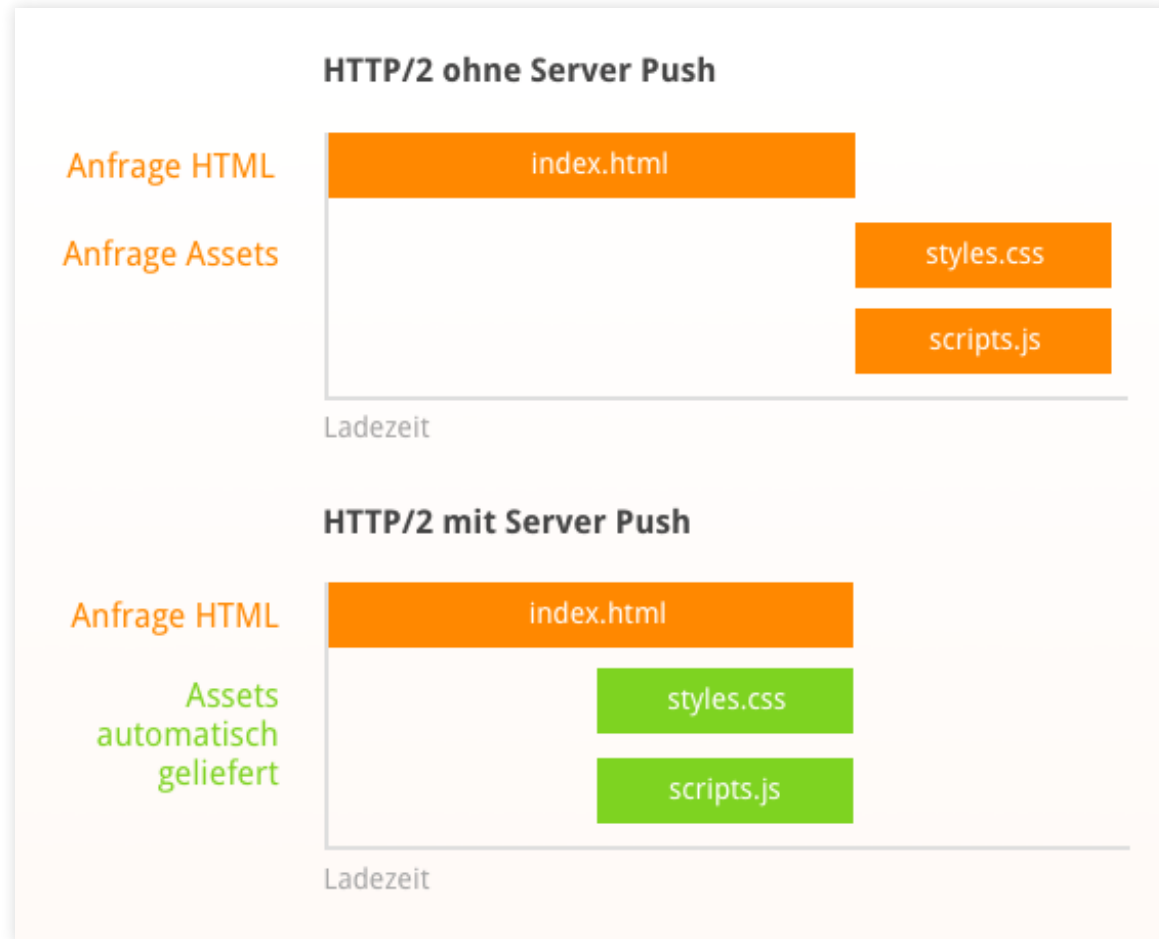


```
#home {
  width: 46px;
  background: url('navsprite.gif') 0 0;
}
#prev {
  width: 43px;
  background: url('navsprite.gif') -47px 0;
}
#next {
  width: 43px;
  background: url('navsprite.gif') -91px 0;
}
```

- Sprite- und CSS-Generatoren gibt es online
 - (verwandte Idee: [FontAwesome](#), 400+ Icons in Schriftart)

Quelle: https://www.w3schools.com/css/css_image_sprites.asp

HTTP/2: Server Push



Quelle: <https://www.cyon.ch/blog/HTTP2-Server-Push>

HTTP/2: Server Push

- Konfiguration in `.htaccess`

```
<FilesMatch "index.html">
  Header add Link "</css/styles.css>; rel=preload; as=style"
  Header add Link "</js/scripts.js>; rel=preload; as=script"
</FilesMatch>
```

- Konfiguration in PHP:

```
<?php
header("Link: </css/styles.css>; rel=preload; as=style, "
      . "</js/scripts.js>; rel=preload; as=script");
```

- Beides sorgt dafür, dass der (HTTP/2)-Webserver automatisch `styles.css` und `scripts.js` ausliefert

Datenvolumen verringern

Ansätze zur Daten-Reduktion

- **Vorbereitende Maßnahmen**
 - (Reduktion erfolgt vor den Anwender-Requests)
 - Verlustfreie Komprimierung von “Texten”
 - Kompression von Medien, optimierte Datenformate
- **Kontextabhängige Maßnahmen**
 - (Reduktion erfolgt basierend auf Anwender-Profil)
 - Komprimierung der übertragenen Daten
 - Kontext-abhängige Bilder

Komprimierung von Text-Dateien

- **Minify:** “Verlustfreie” Komprimierung
 - Löschen von unnötigen Whitespaces
 - Löschen von Kommentaren
- Beispiel mit Symfonys *Webpack Encore*:





```
$ yarn run encore dev # erzeugt lesbare 'public/build/bundle.js'  
$ yarn run encore production # minified 'public/build/bundle.js'
```

- Trick um CSS-Dateien zu Bündeln und “Minifizieren”:
 - CSS-Dateien als “Abhängigkeiten” im JavaScript-Code

```
// assets/main.js  
require('style.css');
```

- `yarn run encore dev|production` → `public/build/bundle.js` & `.css`

Rechenbeispiel: Minified JS & CSS

- Diese Präsentation:
 -  CSS: 70 kBytes
 -  JavaScript: 458 kBytes
- Minified:
 -  CSS: 52 kBytes (= 73%)
 -  JS: 149 kBytes (= 33%)

Komprimierung von Medien

- richtiges Format für Qualität pro Dateigröße:
 - JPG: Fotos
 - PNG: Grafiken, Icons
 - SVG: Grafiken, technische Zeichnungen
- neue Formate, etwa [↗ WebP](#)
 - WebP Lossy: 25-34% kleiner als JPG
 - WebP Lossless: 26% kleiner als PNG
 - Unterstützung: Chrome und Opera, Firefox experimentell
- Anleitungen: [↗ Bild-Optimierung fürs Web](#)

Quelle: [↗ https://developers.google.com/speed/webp/](https://developers.google.com/speed/webp/)

Komprimierung der Übertragung

- Webserver kann Nutzdaten in Antwort komprimieren

```
HTTP/1.1 200 OK
Date: Fri, 15 Sep 2017 14:48:15 GMT
Content-Type: text/html; charset=UTF-8
Content-Encoding: gzip
[... binärer Inhalt, gezipptes HTML-Dokument ...]
```





- ... wenn der Client dies unterstützt












```
Host: www.zeit.de
Accept: text/html,application/xhtml+xml,application/xml;q=0.9
Accept-Language: de-DE
Accept-Encoding: gzip, deflate
```

- transparent z.B. mit Apaches `mod_deflate`:

```
# Direktive zur Komprimierung von Output
AddOutputFilterByType DEFLATE text/html
```

Rechenbeispiel: Kompression

- Minified:
 -  CSS: 52 kB (= 73%)
 -  JavaScript: 149 kB (= 33%)
- Komprimiert durch Apache `mod_deflate`:
 -  CSS: 11 kB (= 16%)
 -  JavaScript: 53 kB (= 12%)

- “Entwicklungsversion”, Einzeldateien:
 -  CSS: 70 kBytes
 -  JS: 458 kBytes
 -  HTTP-Overhead: 16 kB (11 separate Dateien)
- Minified:
 -  CSS: 52 kB (= 73%)
 -  JS: 149 kB (= 33%)
 -  HTTP-Overhead: 16 kB (11 separate Dateien)
- Gebündelt & Komprimiert:
 -  CSS: 11 kB (= 16%)
 -  JS: 53 kB (= 12%)
 -  HTTP-Overhead: 3 kB (2 Bündel-Dateien)
- **Gesamter HTTP-Traffic (= TCP-Payload):**
 -  544 kB
 -  67 kB

Kontextabhängige Bilder

- Mobilgeräte können mit größeren Bildern nicht viel anfangen
- Bilder in mehreren Auflösungen anbieten mit `srcset`

```

```

- bis 200px Viewport-Größe → `xs.png`; darüber `md.png`
- (>600px Bildschirm → 200px; sonst 50% des Bildschirms)

Nachbemerkungen

Weitere Möglichkeiten (Frontend)

- Es gibt noch viele weitere Möglichkeiten, z.B.
 - CDN: Content Delivery Networks
 - für geteilte Ressourcen (“globaler Cache”, z.B. für jQuery)
 - für eigene Ressourcen (Bilder, Stylesheets, JavaScript)
 - Code-Splitting:
 - Nicht eine `bundle.js`, sondern zugeschnitten pro Seite
 - Schriftarten zurechtstutzen:
 - Inkl. 20.000 asiatische Schriftzeichen, oder reicht lat. Alphabet?
- Toller Artikel, viele Checklisten: [↗ The State of the Web](#)

Weitere Möglichkeiten (Backend)

- Zwischenergebnisse speichern: [↗ Symfony Cache](#)

```
$topHeros = $cache->getItem('stats.top_heros');  
if (!$topHeros->isHit()) {  
    $topHeros->set(/* Ergebnis eines teuren SQL-Querys */);  
    $topHeros->expiresAfter(1800); // halbe Stunde frisch  
    $cache->save($topHeros);  
}  
return ['heros' => $topHeros->get()];
```

- Ort: [↗ Doctrine](#), [↗ Dateisystem](#), [↗ Memcached](#), [↗ Redis](#), ...
- allg. PHP-Performance:
 - aktuellere PHP-Version: neuere Engine, bessere Performance
 - **Byte Code Caches:** [↗ OPcache](#), [↗ APCu](#)
 - siehe auch: [↗ http://symfony.com/doc/current/performance.html](http://symfony.com/doc/current/performance.html)
 - [↗ HHVM](#) (von Facebook): PHP → C++

Quelle: [↗ http://symfony.com/doc/current/components/cache.html](http://symfony.com/doc/current/components/cache.html)

Allgemeines zu Optimierungen

- etwas vereinfachter Ablauf:
 1. Ziel und Budget der Optimierung definieren
 2. Metriken und Ziel-Werte definieren
 3. Mess-Szenarien definieren
 4. Baseline bestimmen
 5. Optimierung durchführen
 6. Messungen autom. durchführen (z.B. mit [↗ Lighthouse](#))
 7. Ziel erreicht oder Budget alle? → Nein: zu Schritt 5
- Realistischer:
 - Startpunkt: wirtschaftl. KPIs (*Key Performance Indicators*)
 - Runterbrechen auf Antwortzeiten, HTTP-Status-Codes
 - Kontinuierliches Monitoring

Zusammenfassung

- Perspektiven auf Optimierung
 - Technisch und betriebswirtschaftlich
- Zwei wichtige Optimierungsansätze:
 - **Anzahl der Requests verringern**
 - HTTP-Caching: Modelle *Expiration* und *Validation*
 - Bündeln von (Text-)Ressourcen
 - **Datenvolumen verringern**
 - Minify & Kompression
 - Requests spezifisch für Nutzerverhalten
- Pro Technik verstehen: Wo wirkt sie? Welcher Ansatz?
- Allgemeines Vorgehen bei Optimierungen

Danke!