Ostbayerische Technische Hochschule Amberg-Weiden

Fakultät Elektro- und Informationstechnik

Studiengang Medienproduktion und Medientechnik

Bachelorarbeit

von

Thomas Rehm

Titel (deutsch):

Optimierung der Website-Performance durch den Einsatz serverseitiger Komponenten und reaktionsfähigem Webdesign

Titel (englisch):

Website performance optimization using server side components and responsive web design

Ostbayerische Technische Hochschule Amberg-Weiden

Fakultät Elektro- und Informationstechnik

Studiengang Medienproduktion und Medientechnik

Bachelorarbeit

von

Thomas Rehm

Titel (deutsch):

Optimierung der Website-Performance durch den Einsatz serverseitiger Komponenten und reaktionsfähigem Webdesign

Titel (englisch):

Website performance optimization using server side components and responsive web design

Autor: Thomas Rehm

Erstprüfer: Prof. Dr. Dieter Meiller

Zweitprüfer: M. Eng. Fabian Baumgartner

Bearbeitungszeitraum: 14. November 2013 – 10. April 2014

Bestätigung gemäß § 12 APO

Name und Vorname des Studenten: Rehm, Thomas

Ich bestätige, dass ich die Bachelorarbeit mit dem Titel »Optimierung der Website-Performance durch den Einsatz serverseitiger Komponenten und reaktionsfähigem Webdesign« selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benützt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Amberg am 8. April 2014, Thomas Rehm

Zusammenfassung

Student (Name, Vorname):	Rehm, Thomas
Studiengang:	Medienproduktion und Medientechnik
Semester der Anmeldung:	7
Aufgabensteller/Erstprüfer:	Prof. Dr. Dieter Meiller
Zweitprüfer:	M. Eng. Fabian Baumgartner
Ausgabedatum:	14. November 2013
Abgabedatum:	8. April 2014
Semester der Abgabe:	8
Titel:	Optimierung der Website-Performance durch den Einsatz serverseitiger Komponenten und reaktionsfähigem Webdesign
Zusammenfassung:	Diese Bachelorarbeit beschäftigt sich mit der Optimierung von Webseiten-Performance und den Herausforderungen, denen Webentwickler gegenüberstehen. Anhand einer Beispiel-Webseite werden diverse Probleme betrachtet, die die Performance mindern. Um diese Probleme möglichst effektiv zu verbessern, werden konkrete Lösungsansätze und Techniken untersucht und umgesetzt. Die Optimierung wird für geräteunabhängige Webseitenaufrufe durchgeführt; im Verlauf der Bachelorarbeit aber speziell im Hinblick auf die Optimierung für mobile Aufrufe.
Schlüsselwörter:	Webseiten-Performance, Web-Performance-Optimierung, Responsive Webdesign, Serverseitige Optimierung

Abkürzungen

HTTP	Hypertext Transfer Protocol
HTML	Hypertext Markup Language
XHTML	Extensible Hypertext Markup Language
CSS	Cascading Style Sheets
JS	JavaScript
LTE	Long Term Evolution (Mobilfunkstandard)
UMTS	Universal Mobile Telecommunications System
3G	Dritte Generation des Mobilfunkstandards UMTS
GSM	Global System for Mobile Communications (Standard für volldigitale Mobilfunknetze)
URL	Universal Resource Locator
UA	User Agent
RWD	Responsive Web Design
LESS	Leaner CSS (Stylesheet-Sprache)
MB	Megabyte, das Millionenfache der Maßeinheit Byte
kB	Kilobyte, das Tausendfache der Maßeinheit Byte
DDR	Device Detection Repository
WURFL	Wireless Universal Resource File

Inhaltsverzeichnis

1 Einleitung

Diese Bachelorarbeit beschäftigt sich mit der Optimierung von Webseiten-Performance und den Schwierigkeiten, Herausforderungen und Chancen, denen Webentwickler gegenüberstehen. Anhand einer Beispiel-Webseite werden diverse Probleme betrachtet, die die Performance mindern. Um diese Probleme möglichst effektiv zu verbessern, werden konkrete Lösungsansätze und Techniken untersucht und umgesetzt. Der Einsatz der Techniken wird für geräteunabhängige Webseitenaufrufe durchgeführt; im Verlauf der Bachelorarbeit aber speziell im Hinblick auf die Optimierung für mobile Aufrufe.

1.1 Motivation

Seit der Einführung des erstens Apple iPhones ® im Jahr 2007 (Apple, 2007) hat sich der Markt der Mobiltelefone und Handheld-Devices weltweit verändert. Ein wahrer Smartphone-Boom entwickelte sich und dieser Markt ist mittlerweile einer der am rasantesten wachsenden Märkte. Durch die ansteigende Verbreitung von mobilen, internetfähigen Geräten, wie Smartphones oder Tablets, wächst die Anzahl der mobilen Internetkunden immer schneller (siehe Abbildung 1.1).

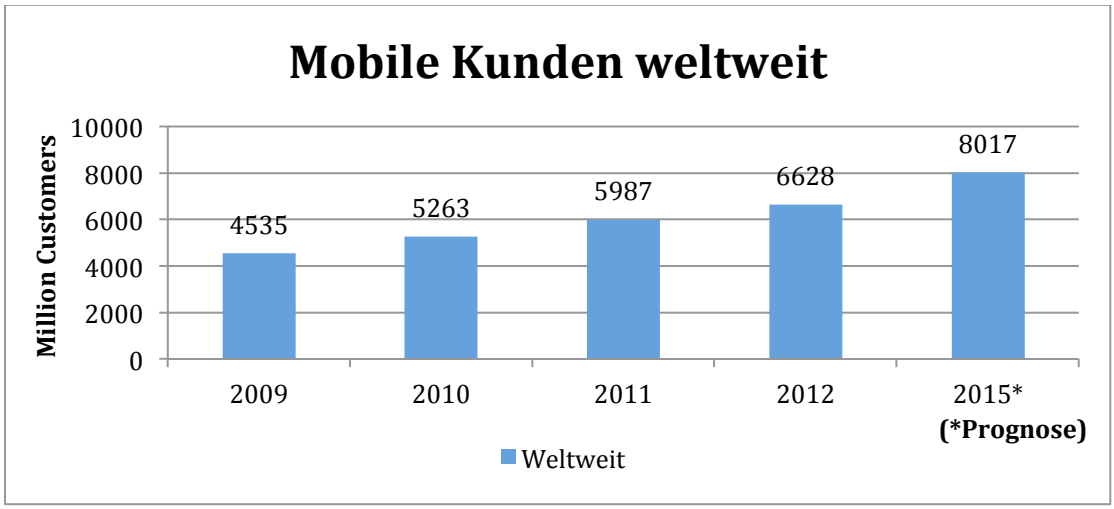

Abbildung 1.1: Mobile Kunden Weltweit, in Million | (DigiWorld by IDATE, 2012)

Mit der steigenden Verbreitung der Geräte und deren Nutzung nimmt auch der Zugriff auf das Internet von mobilen Geräten ständig zu. Die bisherigen Entwicklungen zeigen, dass der Globale Mobile Traffic, also der Datenverkehr bzw. der Zugriff auf Webseiten, zwischen 2006 und 2012 pro Jahr schneller wuchs (Wachstumsrate +146%/Jahr), als es der Globale Festnetz Traffic von 1997 bis 2003 tat (Wachstumsrate +127%/Jahr) (Statista, 2013).

Die Ergebnisse der »2012 Mobile Landscape: Western Europe«-Studie von iProspect (iProspect, 2012) zeigen deutlich: Der mobile Bildschirm bzw. das mobile Gerät entwickelt sich zum sogenannten »First Screen«, also dem im Alltag am häufigsten genutzten Bildschirm vor

Laptop, Desktop oder TV. Durch diese Verschiebung sind Webseiten im Nachteil, die nicht auf die Anforderungen mobiler Geräte eingehen und diese nicht unterstützen. Der Studie zufolge wechseln in einem solchen Fall 61% der Nutzer zu einer Konkurrenz-Webseite.

Eine kürzlich erschienene Prognose (siehe Abbildung 1.2) der Firma Cisco (Cisco Systems, 2014) unterstützt die Aussage, dass der mobile Webseitenaufruf steigt und zeigt, wie stark der mobile Traffic in den nächsten Jahren ansteigen könnte.

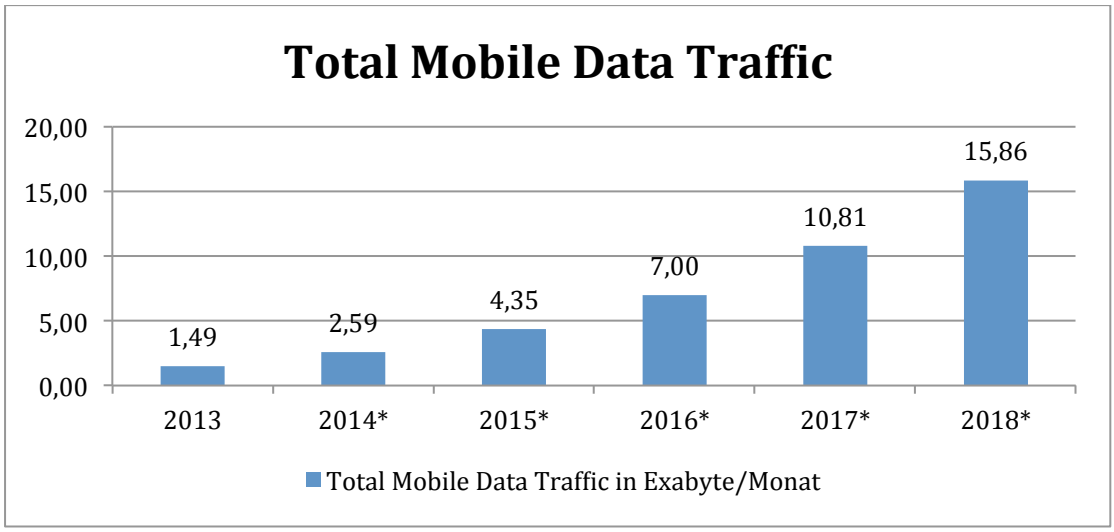

Abbildung 1.2: Total Mobile Data Traffic (* = Prognose) | (Cisco Systems, 2014)

Für Webentwickler bedeutet das, dass die Entwicklung für mobile Geräte immer wichtiger wird und nicht mehr die Entwicklung für den Desktop. Die Unterstützung mobiler Geräte ist nicht mehr nur optional, sondern sollte bei den meisten Projekten an erster Stelle stehen.

Allerdings stellt der mobile Zugriff auf Webseiten für Webentwickler eine ganz besondere Herausforderung dar: Per Kabelnetzwerk sind mittlerweile sehr hohe Übertragungsgeschwindigkeiten realisierbar. Dem gegenüber stehen Funknetzwerke mit viel höheren Ladezeiten und Latenzen. Latenz beschreibt die Zeit, die eine Nachricht für den Weg vom Sender bis zum Empfänger benötigt. Jedes zu ladende Skript und jedes Kilobyte fällt viel stärker ins Gewicht und beeinflusst das Erleben einer Webseite auf einem mobilen Gerät viel stärker als auf einem Gerät mit schneller Datenleitung. Trotz gutem User Interface Design und durchdachter Benutzerführung kann die User Experience einer Webseite schlecht sein, da nichts am Laden einer Webseite vorbeiführt. Je länger ein Nutzer auf eine Webseite zum Laden warten muss, desto geringer wird seine Aufmerksamkeit und sein Interesse an dieser Webseite. Jakob Nielsen zufolge beträgt diese Zeitspanne nur 1-10 Sekunden (Nielsen, 2010).

Neben der wichtigen User Experience und der hohen Erwartung der Nutzer, gibt es weitere Punkte, die durch die Optimierung der Performance einer Webseite profitieren: Die Ladezeit einer Webseite wird von Googles Such-Algorithmus einbezogen. So können Webseiten mit

einer optimierten Ladezeit in Googles Ranking aufsteigen. Dies geht aus einem Eintrag auf Googles Blog »Google Webmaster Central Blog« hervor (Singhal & Cutts, 2010).

Ebenso lässt sich mit einer optimierten Webseite Geld einsparen, da weniger Daten vom User geladen werden müssen und so weniger Traffic entsteht. Dabei sind Einsparungen aber sehr stark abhängig von der Popularität einer Webseite und den gebuchten Tarifen beim Web-Hoster.

Zur Herausforderung des mobilen Zugriffs kommt für die Designer und Entwickler das Problem der steigenden Vielfalt unterschiedlicher Bildschirmvarianten und Gerätearten. (Zillgens, 2013, S. 12f) Um diese Probleme möglichst einfach in den Griff zu bekommen, haben sich viele Entwickler und Designer an den Einsatz von Frameworks und Plugins gewöhnt (Zillgens, 2013, S. 321). So entstehen oft Webseiten, die zwar auf Geräten mit Breitband-Internet schnell, jedoch auf mobilen Geräten mit einer womöglich schwachen Funkverbindung nur langsam geladen werden können.

1.2 Problemstellung

Die vorrangehenden Abschnitte zeigen, dass die Web-Entwicklung für mobile Geräte hohe Anforderungen an die Entwickler stellt. Die Performance einer Webseite ist für die User Experience besonders auf mobilen Geräten wichtig und sollte von Anfang an bei einem Projekt richtig angegangen werden.

Hinter dem Titel »Optimierung der Website-Performance durch den Einsatz serverseitiger Komponenten und reaktionsfähigem Webdesign« verbirgt sich das Ziel, Lösungsansätze aufzuzeigen und zu untersuchen, die dazu dienen, die Ladezeiten von Webseiten im Allgemeinen, aber besonders im Bezug auf den mobilen Zugriff zu optimieren. Die Herausforderung Websites auf mobilen Geräten bedienbar zu machen, bildet einen Teil der Problemstellung: Ich werde eine Webseite erstellen, die sich mit Hilfe von reaktionsfähigem Webdesign den Bildschirmgrößen anpassen kann (siehe Abschnitt 3.2). Diese Webseite wird sich an dem Webdesign-Konzept von Ethan Marcotte orientieren:

> *Rather than tailoring disconnected designs to each of an ever-increasing number of web devices, we can treat them as facets of the same experience. We can design for an optimal viewing experience, but embed standards-based technologies into our designs to make them not only more flexible, but more adaptive to the media that renders them. In short, we need to practice responsive web design.*
>
> *(Marcotte, A List Apart - Responsive Webdesign, 2010)*

Der Hauptteil der Arbeit wird darin bestehen, die Beispiel-Webseite, die mit diversen Plugins und Frameworks arbeiten wird, zu betrachten. Für die beobachteten Probleme werden Lösungsansätze zum Optimieren untersucht und eingesetzt.

2 Problematik

In den folgenden Abschnitten werden diverse grundlegende Konzepte, Begriffe und Techniken erklärt. Diese sind für das Verständnis der darauf aufbauenden Techniken essentiell.

In der Einleitung wurden diverse Punkte angeführt, warum es wichtig ist, die Performance einer Webseite nicht zu vernachlässigen. Bisher wurde jedoch nicht geklärt, wie es überhaupt zu einer schlechten Performance einer Webseite kommt.

Die Performance einer Webseite definiert sich darüber, wie schnell eine Webseite und deren Komponenten geladen und auf einem Endgerät dargestellt werden. Eine Aussage darüber, wie schnell eine Webseite geladen wird, lässt sich treffen, indem man die benötigte Zeit misst. Diese Zeit nennt man »Page Load Time« (PLT) (Grigorik, HPBN, 2013, S. 166f). Hierzu kann man in diversen Browsern die Webentwickler Werkzeuge aufrufen und in einem Wasserfall-Diagramm (siehe Abbildung 2.1) darstellen lassen, welche Dateien der Browser lädt und wie viel Zeit dafür benötigt wird.

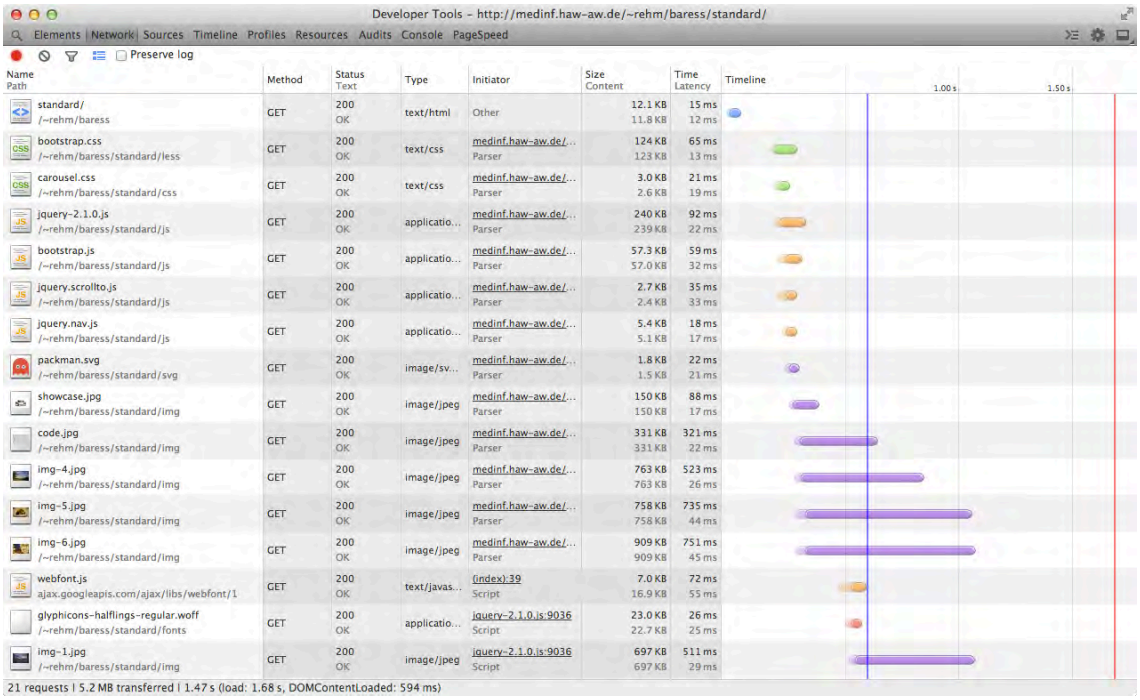

Abbildung 2.1: Screenshot Wasserfall Diagramm Google Chrome Developer Tools

Neben Verbindungstyp und Internetgeschwindigkeit ist die Ladezeit einer Webseite von zwei Hauptfaktoren abhängig: Größe und Anzahl der zu ladenden Dateien (Zillgens, 2013, S. 315f). Diese Datei-Anfragen, die via Hypertext-Übertragungsprotokoll (HTTP) übertragen werden sollen, nennt man »HTTP Request«. Hierzu zählen alle Dateien, die beim Aufrufen einer Webseite geladen werden. Normalerweise sind das HTML-Dokumente, CSS-Dateien, JavaScript-Dateien, Grafiken, Schriften und jegliche Mediendateien. Je mehr Requests von

einer Webseite gestellt werden, desto mehr Dateien müssen heruntergeladen werden und umso größer die Dateien sind, desto länger dauert der Download. Der Browser ist nicht in der Lage, alle Dateien auf einmal herunterzuladen, da er immer nur eine maximale Anzahl von Verbindungen zum selben Server gleichzeitig herstellen kann (Zillgens, 2013, S. 317). Bei aktuellen Browsern sind das zwischen vier und acht pro Server. Ein Download muss erst vollständig abgeschlossen sein, damit der Browser via HTTP wieder eine neue Verbindung öffnen kann, um die nächste Anfrage zu stellen. Folglich steigt mit der Anzahl der Requests und Größe der Dateien auch die Zeit zum Herunterladen aller Dateien. Damit der Browser etwas anzeigen kann, müssen die Dateien erst vollständig geladen, dann vom Browser analysiert und danach dargestellt werden. Dieser Vorgang, »parsing« genannt, dauert umso länger, je komplexer das Dokument ist (Zillgens, 2013, S. 317). Das Parsen der Dateien ist abhängig von der Leistung eines Geräts und kann unter Umständen viele Ressourcen benötigen, welche nicht bei allen Geräten gleichermaßen vorhanden sind. Probleme entstehen derzeit beim Parsen eigentlich nur noch bei schwächeren Mobilen Geräten. Bei aktuellen Smartphones, Tablets oder Laptops ist das nicht relevant, da diese in der Regel über genügend Leistung verfügen.

Die Verbindungsart macht sich allerdings stark bemerkbar. Nicht überall steht das aktuelle LTE-Netz zur Verfügung. In den meisten Regionen empfängt man ein 3G-Signal. Funknetze erreichen mittlerweile erstaunlich hohe Downloadgeschwindigkeiten, jedoch bleibt die Latenz von Funknetzwerken immer sehr viel höher als die von kabelgebundenen Netzwerken (Zillgens, 2013, S. 319-320).

Wenn also Performance-Probleme betrachtet werden, so stellt man fest, dass die Probleme nicht für alle auftretenden Fälle gleich schwerwiegend sind. Die Kombination aus einem schwachen mobilen Gerät mit schwacher mobiler Internetverbindung ergibt Performance-Probleme, wenn der Webentwickler nicht sinnvoll mit den Möglichkeiten gearbeitet hat, die ihm zur Verfügung stehen.

Wie schon in der Einleitung beschrieben, haben sich die Nutzer an schnelle Antwortzeiten einer Webseite gewöhnt und erwarten dies auch über eine mobile Internetverbindung. Darum ist es wichtig, sich Gedanken um die Performance einer Webseite zu machen, wenn man mobile Kunden weiterhin bedienen will.

3 Konzepte

Um mobile Internetnutzer, gerade an Smartphones oder Tablets, ernsthaft mit dem Inhalt einer Webseite versorgen zu können, haben sich in den letzten Jahren zwei wichtige Konzepte zur Bereitstellung von Webseiten etabliert.

3.1 Mobile Webseite

Erste Erfahrungen mit dem Aufruf von Webseiten über eine mobile Internetverbindung und einem Handy wurden Ender der 1990er Jahre gemacht, als die Entwicklung des GSM-Netzes weit genug vorangeschritten war. Seit der Einführung zur kommerziellen Nutzung der GSM-Weiterentwicklung »GPRS« wurde die Paketdatenübertragung über das GSM-Netz ermöglicht (GSM Association). Allerdings waren die Geschwindigkeiten sehr niedrig. Aus den technischen Gegebenheiten und dem Wunsch, Inhalte auf mobile Geräte mit einer annehmbaren Wartezeit zu bringen, entwickelte sich das Konzept für Mobile Webseiten: Parallel zur Desktop-Webseite wird eine zweite Variante entwickelt und betrieben. Traditionell handelt es sich bei diesen Webseiten um reduzierte Varianten der Desktop-Webseite.

Diese Variante der Unterstützung und Belieferung für mobile Browser kann sehr gut optimiert werden. Es wird nur Code geladen, der genau auf mobile Browser ausgelegt ist. Dadurch wird der geladene Code sehr klein und enthält nur wenige Skripte oder Bilddateien. Unter Umständen leidet die Optik und im schlimmsten Fall auch die Bedienbarkeit der mobilen Webseite darunter. Ein weiterer Nachteil ist die Weiterleitung auf eine andere URL als die Desktop-Webseite (z.B. `m.example.com` oder `www.example.com/mobi`). Eine Weiterleitung des Nutzers auf eine andere Domain bedeutet immer eine Verzögerung des eigentlichen Webseitenaufrufs, da das korrekte HTML-Dokument und die dazugehörenden Ressourcen noch nicht geladen werden (Souders, Yahoo! Developer Network, 2007).

Unterschiedliche URL können problematisch werden, wenn keine gute Weiterleitung eingerichtet ist. So kann beispielsweise ein mobiler Nutzer einen Link in einem sozialen Netzwerk posten. Ein anderer Nutzer ruft diesen Link am Desktop auf, wird aber nicht richtig weitergeleitet. So könnte er auf einer mobilen Webseite landen, die wiederum auf einem Desktop schlecht bedienbar ist und teilweise nur eine eingeschränkte Funktionalität bietet.

Bei manchen Webseiten wird der Nutzer automatisch zu der Variante der Webseite weitergeleitet, die seinem Gerät entspricht, wie zum Beispiel der Shop `www.alternate.de`. Andere Seiten bieten trotz mobiler Webseite keine automatische Weiterleitung an, wie beispielsweise das Nachrichtenportal `www.spiegel.de` (siehe Abbildung 3.1 und Abbildung 3.2).

Abbildung 3.1: Screenshot iPhone-Simulator spiegel.de-Startseite (Stand: 26. Februar 2014)

Abbildung 3.2: Screenshot iPhone-Simulator m.spiegel.de-Startseite (Stand: 26. Februar 2014)

Die automatischen Weiterleitungen lassen sich beispielsweise durch Auswerten des User Agent (UA) Strings realisieren (Google Developers, 2014). Dieser enthält Informationen über Browser und System (siehe Kapitel 4.3.1.1).

Vorteile des Konzepts:

- Sehr gute Optimierbarkeit für mobile Geräte, da ausschließlich für diese entwickelt
- Schnelle Ladezeiten, da Bilder und Skripte sehr stark komprimiert werden können
- Reduktion auf das Wesentliche einer Webseite (kann gleichzeitig zum Nachteil werden, da eventuell nicht alle Funktionen dem Nutzer der mobilen Webseite zur Verfügung gestellt werden)
  (Krenz-Kurowska & Kurowski, 2013, S. 7)

Nachteile des Konzepts:

- Hoher Entwicklungsaufwand, da parallel zur Desktop Webseite entwickelt und gepflegt werden muss
- evtl. doppelter Content bedeutet doppelte Pflege, gerade bei Mediadateien
- Unterschiedliche URLs können zum Problem bei der Verbreitung von Links werden

3.2 Responsive Webdesign

Bevor Ethan Marcotte in seinem Artikel (Marcotte, A List Apart - Responsive Webdesign, 2010) den Begriff »Responsive Webdesign« (RWD) prägte, wurden Webseiten normalerweise starr und für eine durchschnittliche Bildschirmbreite entwickelt. Die Designer konnten somit die Kontrolle über das Layout und Design einer Webseite behalten. Da die meisten Personen über einen Desktop-PC oder Laptop die Seiten aufriefen, bestand nicht unbedingt der Bedarf, möglichst viele unterschiedliche Bildschirmgrößen abzudecken. Es setzten sich Layout Raster durch, die sich an einer durchschnittlichen Bildschirmbreite von 1024 Pixeln orientieren, wie das beliebte 960gs-Grid-System[1] (Zillgens, 2013, S. 2).

Abbildung 3.3: Unterschiedliche Bildschirmgrößen (Beispielhaft, eigene Darstellung)

Mit der Einführung des iPhones änderte sich einiges auf dem Markt der Unterhaltungselektronik. Seitdem haben es Webentwickler mit einer großen Varietät an Bildschirmgrößen zu tun (siehe Abbildung 3.3). Betrachtet man die Produktpaletten der Hersteller von Smartphones und Tablets, wird einem bewusst, wie viele Unterschiedliche Bildschirmgrößen es mittlerweile gibt.

Um diese einfacher mit gut aussehenden Webseiteninhalten versorgen zu können, entwickelte Ethan Marcotte aus bestehenden Techniken das Konzept des Responsive Webdesign: Design und Layout einer Webseite sollten nicht mehr starr für eine feste, durchschnittliche Bildschirmbreite entwickelt werden, sondern »responsive«, also übersetzt »reaktionsfähig«, sein. Bei reaktionsfähigem Webdesign passt sich vor allem das Layout der Bildschirmgröße an: Bilder werden passend skaliert und am Raster ausgerichtete Elemente werden an bestimmten Breakpoints (dt. Anhaltepunkt o. Umbruchpunkt) per CSS3 Mediaqueries neu angeordnet. Die Breite von DIV-Elementen passt sich der Bildschirmbreite an und die Größe von Elementen,

[1] Vgl. http://960.gs/ (Stand: 24. März 2014)

wie zum Beispiel Buttons, kann ebenfalls durch die Breakpoints für unterschiedliche Bildschirmgrößen optimiert werden. Mediaqueries ist eine CSS3 Technik, die es ermöglicht, Bildschirmgrößen abzufragen und dazu ein entsprechendes CSS zu liefern (Zillgens, 2013, S. 37f). Die Entwicklung einer reaktionsfähigen Webseite bringt neben der Anpassbarkeit weitere Vorteile mit sich.

Vorteile des Konzepts:

- Es wird nur einmal Content benötigt, da es sich für alle Geräte um die gleiche Webseite handelt
- Auf allen Geräten erscheint unter der gleichen URL eine gut zu bedienende Webseite mit identischem Inhalt (wenn richtig umgesetzt)
- Da Responsive Webdesign zunächst einmal nur die Bildschirmbreite für die Reaktionsfähigkeit benötigt wird, ist das Konzept an sich geräteunabhängig und bedarf keiner Weiterleitung
  (Krenz-Kurowska & Kurowski, 2013, S. 10-14)

Das Konzept birgt jedoch gerade für die Entwicklung mit Ausrichtung auf mobile Geräte einige Nachteile, wenn es nicht durchdacht eingesetzt wird. Genauso wie das 960gs-Grid-System haben sich auch im Responsive-Bereich schnell einige Tools und Frameworks etabliert, die Designern und Entwicklern einige Arbeit abnehmen. Das wohl populärste Framework ist Bootstrap[2] von Twitter. Weitere Beispiele sind Foundation[3], Skeleton[4] oder Gumby[5]. Webdesign-Frameworks sind Sammlungen von Hilfsmitteln für die Gestaltung von Webseiten. Die meisten der genannten Beispiele sind nicht nur einfache Grid-Systeme, die das Umbrechen und neu anordnen von Elementen steuern, sondern beinhalten große Paletten an fertigen Elementen wie Buttons oder Icons. Der Einsatz eines Frameworks für ein Responsive-Web-Projekt ist verlockend und in sehr vielen Fällen auch durchaus sinnvoll, jedoch dann nicht, wenn es um umfassend optimierte Webseiten geht. Der CSS-Code, der geladen werden muss, ist in den meisten Fällen viel zu umfangreich und viele Funktionen werden überhaupt nicht verwendet.

Nachteile des Konzepts:

- Durch unbedachten Einsatz von Frameworks wird zu viel Code geladen
- Alte Browser unterstützen unter Umständen noch keine CSS3-Techniken, die aber von den meisten Frameworks benutzt werden
- Die meisten Webseiten, die Frameworks einsetzen, ähneln sich optisch stark
- Umsetzen von RWD erfordert viel Testen auf unterschiedlichen Geräten

[2] Vgl.: http://getbootstrap.com/ (Stand: 24. März 2014)
[3] Vgl.: http://foundation.zurb.com/ (Stand: 24. März 2014)
[4] Vgl.: http://www.getskeleton.com/ (Stand: 24. März 2014)
[5] Vgl.: http://gumbyframework.com/ (Stand: 24. März 2014)

3.3 RESS – Stand der Technik?

Man kann weder bei Responsive Webdesign noch bei Mobilen Webseiten vom »Stand der Technik« sprechen. Zu viele gravierende Probleme stehen dem Webentwickler bei beiden Konzepten im Weg. Beide Konzepte haben ihr Berechtigungsdasein, jedoch sind sie alleine nicht perfekt. Grundsätzlich gilt, Responsive Webdesign ist im Grunde nur ein Design- und Layout-Konzept, Mobile Webseiten gehen tiefer und bieten eine bessere Optimierbarkeit für die mobile Entwicklung.

Die Kombination aus mobiler Webseite und Responsive Webdesign könnte man zur Zeit als Stand der Technik bezeichnen. Genau dieses Konzept hat Luke Wroblewski in einem Artikel erklärt: »RESS: Responsive Web Design + Server-Side Components« lautet der Titel des Artikels und gleichzeitig auch des Konzepts und bedeutet auf deutsch in etwa »Reaktionsfähiges Webdesign kombiniert mit serverseitigen Komponenten« (Wroblewski, RESS: Responsive Design + Server Side Components, 2011). RESS verbindet die Vorteile von Responsive Webdesign mit den Vorteilen der Optimierbarkeit von mobilen Webseiten. Dadurch entsteht ein System, das gleichzeitig alle Bildschirmgrößen abdecken kann. Zusätzlich kann durch serverseitige Optimierung nur der Code oder die Komponenten auf ein Gerät geladen werden, die von diesem unterstützt werden. Durch Abfragen der Fähigkeiten eines Geräts kann eine bestmögliche User Experience ermöglicht werden (Krenz-Kurowska & Kurowski, 2013, S. 16). Die genaue Funktionsweise wird in Abschnitt 4.3 behandelt.

4 Praxis

Der praktische Teil der Bachelorarbeit beschäftigt sich mit dem Problem, wie man in einem Webprojekt die Performance einer Webseite effektiv verbessern kann. Hierzu habe ich eine Test-Webseite erstellt, um Techniken zur Optimierung vorstellen und analysieren zu können. Die Test-Webseite dient gleichzeitig als kleine Informations-Webseite zum Thema Webseiten Performance Optimierung. Der Arbeitstitel der Webseite lautet »BaRESS« und steht für »Bachelorarbeit Responsive Webdesign + server-side components«. Die Webseite ist in drei Stufen unterteilt: Die erste Stufe repräsentiert die Standard-Variante ohne Optimierung. Die zweite Stufe baut darauf auf und setzt allgemeine bewährte Optimierungs-Techniken ein. Die dritte Stufe baut wiederrum auf der optimierten Variante auf und beschäftigt sich mit der Frage, wie mittels serverseitiger Geräteerkennung für mobile Geräte noch mehr optimiert werden kann.

Die Webseite ist unter dem folgenden Link zu finden:

medinf.haw-aw.de/~rehm/baress/standard/

Hinweis: Da die Hochschule nicht garantieren kann, dass die Webseite für längere Zeit auf dem Server der Hochschule gehostet werden kann, stelle ich die Dateien sowie die Bachelorarbeit in einem freien GitHub Repository zur Verfügung:

github.com/thomasrehm/BaRESS

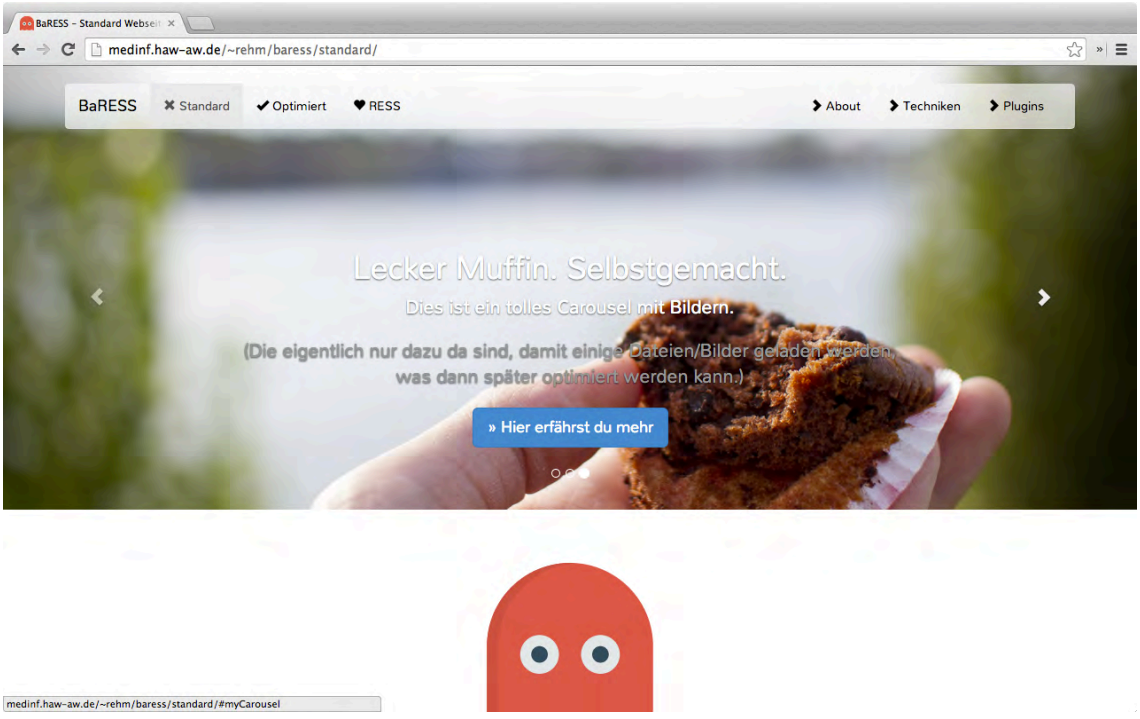

Abbildung 4.1: Screenshot BaRESS Standard

4.1 Standard BaRESS Webseite

Die Standard-Variante der BaRESS-Webseite soll eine durchschnittliche Webseite repräsentieren, die zur Design-Unterstützung für unterschiedliche Bildschirmgrößen nach dem RWD-Konzept aufgebaut ist. Um dies zu erreichen, bildet das Framework »Bootstrap« in der Version 3 von Twitter das Grundgerüst. Ich habe mich für dieses Framework entschieden, da es eines der populärsten und somit gut für eine aktuelle, durchschnittliche Webseite geeignet ist. Bootstrap ist mit der Stylesheet-Sprache »LESS«[6] entwickelt worden. LESS erweitert die Beschreibungssprache CSS um Variablen und die Fähigkeit, Regeln zu verschachteln. Am Ende des Entwicklungsprozesses werden die LESS-Dateien in CSS-Dateien kompiliert und in der Webseite als normales Stylesheet verlinkt. Der Vorteil von LESS ist, dass der Webentwickler deutlich weniger Code schreiben muss und durch Variablen sehr viel flexibler arbeiten kann.

Als Hauptschrift der Webseite verwende ich von Google Fonts die Schrift »Nunito« mit zwei Schriftschnitten (thin und bold).

Auf der Webseite habe ich Bilder eingebaut, damit Content zur Verfügung steht, um diesen später optimieren zu können. Die Bilder sind nicht optimal komprimiert, um dem Anspruch einer durchschnittlichen Webseite gerecht zu werden. Es werden Pixelgrafiken im JPG-Format sowie im Vektorgrafiken im SVG-Format eingebunden.

Außerdem wird die Webseite noch mit JavaScript-Plugins um kleinere Funktionen erweitert. JavaScript-Plugins einzubinden, ist im Webentwickleralltag üblich. Viele Plugins basieren auf der JavaScript-Bibliothek »jQuery«, die bei der BaRESS-Webseite in der Version 2.1.0 verwendet wird. Für die Funktion des weichen Scrollens bei Klick auf einen der Ankerlinks innerhalb der Navigation (siehe Abbildung 4.2, rechter Teil), werden die beiden JavaScripte »jquery.nav.js« und »jquery.scrollto.js« eingebunden.

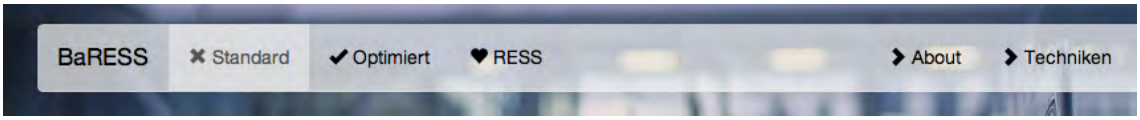

Abbildung 4.2: Screenshot Navigation

Alle verwendeten Ressourcen stehen unter freien Lizenzen zur Verfügung. Die Bilder sind von mir selbst erstellt worden.

4.1.1 Analyse

Die mit einigen Bildern und Inhalt gefüllte BaRESS-Standard-Webseite ist im Download nun einige MB groß und stellt eine bestimmte Anzahl an HTTP Requests. Mittels kostenloser online Tools lassen sich Webseiten testen und ein Performance Report generieren, wie beispielsweise

[6] Vgl.: http://www.lesscss.de/ (Stand: 24. März 2014)

das Website-Performance-Tool der Firma GTmetrix[7]. Dieser findet sich unter dem folgenden Link: http://gtmetrix.com/reports/medinf.haw-aw.de/nGOgSRnD

Die Analyse der BaRESS-Standard-Webseite (medinf.haw-aw.de/~rehm/baress/standard) ergab folgendes:

Page Load Time	Total Page Size	Total Number of Requests
3,62sec	5,24 MB	21

Tabelle 4.1: GTmetrix Analyse Ergebnis BaRESS Standard

Abbildung 4.3: Screenshot Wasserfall Diagramm BaRESS Standard

Dateityp	Anzahl/Gesamt	Größe/Gesamt
Documents	1 / 21	12,1 kB / 5,24 MB
Images	9 / 21	4,7 MB / 5,24 MB
Scripts	5 / 21	312 kB / 5,24 MB
Stylesheets	3 / 21	128 kB / 5,24 MB
Fonts	3 / 21	69,5 kB / 5,24 MB

Tabelle 4.2: Detaillierte Analyse Requests BaRESS Standard

Die detaillierte Auswertung der Requests (siehe Abbildung 4.3 und Tabelle 4.2) zeigt, dass der Hauptanteil an geladenen Daten in Bildern zu finden ist.

Im Vergleich mit aktuellen Werten (siehe Abbildung 4.4), die vom HTTP Archive[8] gemessen wurden, zeigt sich, dass die Webseite weit über dem aktuellen »Durchschnitts-Gewicht« von ca. 1,7 MB der analysierten Webseiten liegt. Am 3.1.2014 wurden vom HTTP Archive 290.835 URLs analysiert.

[7] Vgl.: http://gtmetrix.com/ (Stand: 24. März 2014)
[8] Vgl.: http://httparchive.org/interesting.php (Stand: 24. März 2014)

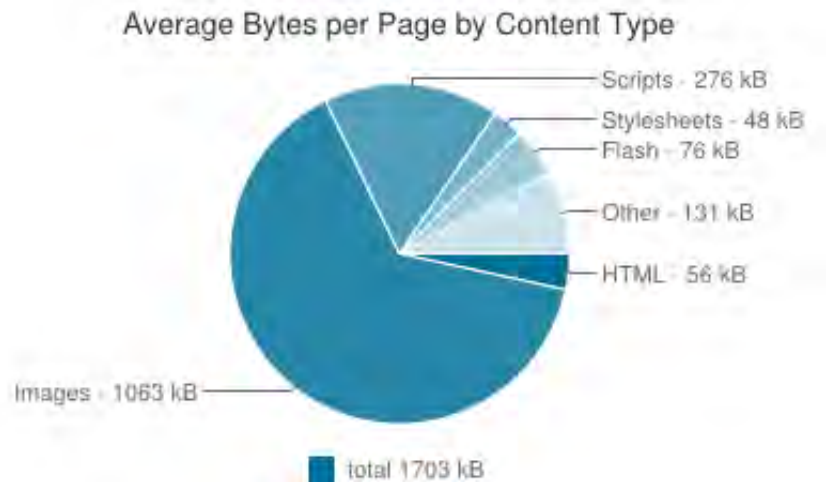

Abbildung 4.4: Werte vom 1.3.14, Quelle: http://httparchive.org/interesting.php

4.2 Optimierte BaRESS Webseite

Wie die Analyse der BaRESS-Standard-Webseite gezeigt hat, ist noch deutliches Verbesserungspotential in Sachen Performance vorhanden. Den größten Anteil des Gewichts machen die Bilder aus, danach kommen JavaScripte und CSS-Dateien. Die optimierte BaRESS-Webseite ist unter der URL medinf.haw-aw.de/~rehm/baress/optimiert/ zu finden.

Im Folgenden werden Lösungsvorschläge zur Optimierung untersucht und umgesetzt. Zunächst werden nur Techniken betrachtet, die geräteunabhängig die Performance der Webseite verbessern. Nicht nur für die Nutzung via mobiles Internet ist eine Optimierung sinnvoll. Natürlich kann auch bei Zugriffen über schnelle Internetverbindungen die User Experience durch schnellere Webseiten erhöht und Traffic reduziert werden.

Steve Souders, Buchautor zum Thema Web Performance und Head Performance Engineer bei Google, hat auf seinem Blog eine »Best-Practice«-Liste (Souders, Steve Souders, 2007) zusammengetragen. Diese Liste enthält bewährte Techniken aus der Praxis, an denen ich mich zur allgemeinen Performance Optimierung orientieren werde.

Der Konsens aus den meisten Büchern, Blogeinträgen oder Artikeln über die besten Methoden der Webseiten-Optimierung lautet: Dateigrößen verkleinern (siehe 4.2.1), HTTP-Requests verringern (siehe 4.2.2), Caching aktivieren (siehe 4.2.3) und sinnvoll CSS-/JS-Dateien im Dokument platzieren.

Hinweis: Die folgenden Optimierungstechniken werden im Rahmen dieser Bachelorarbeit für Webserver vom Typ »Apache« durchgeführt. Ich beschränke mich jedoch auf Apache-Webserver, da auch der Medieninformatik-Server von diesem Typ ist, auf dem ich die Techniken einsetzen kann. Es gibt viele weitere Webserver für die ähnliche Techniken zur Verfügung stehen. Die Ansätze sind bleiben jedoch gleich.

4.2.1 Dateigrößen verkleinern – Minifizierung

Im Grunde ist es möglich, bei den meisten Dateien noch etwas einzusparen. Bilder machen bei fast allen Webseiten den größten Anteil aus. Aus den Werten des HTTP Archive (siehe Abbildung 4.4) geht hervor, dass zurzeit Bilder etwa 62 % des gesamten Gewichts einer durchschnittlichen Webseite ausmachen. In unserem Fall machen Bilder ca. 89 % aus. Momentan werden 4,7 MB für Bilder benötigt. Dies lässt sich mit einigen Mitteln deutlich reduzieren. Jedes einzelne Bild ist zwar unter einem MB groß und damit für Bilddateien relativ klein, jedoch nicht für eine Webseite, die über das Internet am besten möglichst wenige Kilobyte laden sollte. Um die Bilder zu reduzieren, sollten diese zunächst beim Export aus einem Grafikprogramm wie »Photoshop« oder »Lightroom« von Adobe o.ä. mit einer hohen Komprimierungsrate exportiert werden. Dabei muss darauf geachtet werden, dass durch die Komprimierung nicht zu viel Bildinformation verloren geht oder das sogenannte Kompressionsartefakte[9] sichtbar werden. Wie viel pro Bild eingespart werden kann, ist abhängig von den Informationen im Bild und dem jeweiligen Dateiformat. Bilder im PNG- oder GIF-Format lassen sich beispielsweise mit dem kostenlosen Programm »ImageOptim«[10] gut optimieren. Durch Verkleinern der Auflösung und höherer Kompressionsraten wurden die Bilder von insgesamt 4,7 MB auf 2 MB verkleinert.

Da es an dieser Stelle um die allgemeine Webseiten-Optimierung geht, werden hier Techniken zur Bildoptimierung nicht weiter vertieft.

Bei Textdateien können ebenfalls einige Kilobyte eingespart werden. Die meisten Textdateien sind normalerweise ähnlich wie jedes beliebige Textdokument geschrieben: Leerzeichen, Leerräume, Einrückungen und Kommentare strukturieren einen Text oder Anweisungen so, dass sie von einem Menschen gut erfasst und interpretiert werden können (Wenz & Hauser, 2013, S. 368ff) Der Browser benötigt diese Elemente nicht, sondern überspringt sie beim Parsen einfach. Entfernt man alle überflüssigen Elemente, wird der Code kürzer und es müssen weniger Daten übertragen werden. Den Vorgang, alle überflüssigen Zeichen zu entfernen, nennt man »minification« (o.A. (Wikipedia), 2014). Besonders bei JavaScript und CSS ist das unter Umständen sehr effektiv.

Die Minifizierung des Codes kann wie folgt aussehen:

```
.fill {
```

[9] Vgl.: http://de.wikipedia.org/wiki/Kompressionsartefakt (Stand: 24. März 2014)
[10] Vgl.: http://imageoptim.com/ (Stand: 24. März 2014)

```
  width: 100%;
  height: 100%;
  background-position: center;
  background-size: cover;
}
```

wird zu:

```
.fill{width:100%;height:100%;background-
position:center;background-size:cover}
```

Die Einsparung beträgt bei diesem kurzen Beispiel 14 Zeichen.

Tabelle 4.3 zeigt die von der BaRESS Standard Webseite geladenen JS und CSS Files und die Einsparung durch die Minifizierung:

Datei	Größe unminified	Größe minified
bootstrap.js	58 kB	28 kB
jquery-2.1.0.js	245 kB	84 kB
jquery.nav.js	5 kB	2 kB
jquery.scrollto.js	2 kB	2 kB
bootstrap.css	102 kB	83 kB
carousel.css	3 kB	1 kB
Gesamtgröße	415 kB	200 kB

Tabelle 4.3: Minfication JavaSkript und CSS

Durch die Minifizierung des Codes lassen sich insgesamt 216 kB einsparen. Es müssen nur noch 48 % des eigentlichen Codes übertragen werden.

Die Minifizierung sollte nicht von Hand geschehen, sondern durch Tools, die es für genau diesen Zweck gibt. Hierfür gibt es einige kostenpflichtige Programme (Prepros[11], Smaller[12] etc.) oder diverse Online-Tools (refresh-sf.com[13], minify.avivo.si[14] etc.). Einige Entwicklungsumgebungen unterstützen diese Technik ebenfalls, wie beispielsweise PhpStorm von JetBrains[15].

Um die minifizierten Dateien nutzen zu können, müssen diese anstatt der nicht-minifizierten Dateien auf den Server geladen und im HTML- oder PHP-Dokument neu verknüpft werden. Um diesen Arbeitsschritt zu umgehen und die Minifizierung serverseitig ablaufen zu lassen, gibt es diverse kleine PHP Apps. Ich nutze bei der BaRESS Optimiert Webseite das Tool

[11] Vgl.: http://alphapixels.com/prepros/ (Stand: 24. März 2014)
[12] Vgl.: http://smallerapp.com/ (Stand: 24. März 2014)
[13] Vgl.: http://refresh-sf.com/yui/ (Stand: 24. März 2014)
[14] Vgl.: http://minify.avivo.si/ (Stand: 24. März 2014)
[15] Vgl.: http://www.jetbrains.com/phpstorm/ (Stand: 24. März 2014)

»SmartOptimizer«[16] von Ali Farhadi. Durch eine htaccess-Anpassung werden alle JS/CSS-Dateianfragen an die SmartOptimizer-PHP-Datei weitergeleitet, welches dann automatisch die Minifizierung durchführt. Gleichzeitig werden alle minifizierten Dateien gecached, also auf dem Server gespeichert, damit nicht bei jedem Seitenaufruf die Minifizierung neu durchgeführt werden muss, sondern nur dann, wenn sich etwas an den Originaldateien geändert hat. Die Anweisungen in der htaccess-Datei sehen wie folgt aus:

```
<IfModule mod_rewrite.c>
RewriteEngine On

RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule ^(.*\.(js|css))$ smartoptimizer/?$1

<IfModule mod_expires.c>
RewriteCond %{REQUEST_FILENAME} -f
RewriteRule ^(.*\.(js|css|html?|xml|txt))$ smartoptimizer/?$1
</IfModule>

<IfModule !mod_expires.c>
RewriteCond %{REQUEST_FILENAME} -f
RewriteRule
^(.*\.(gif|jpg|jpeg|png|swf|css|js|html?|xml|txt|ico))$
smartoptimizer/?$1
</IfModule>
</IfModule>
```

Mithilfe von Smartoptimizer lässt sich auch eine weitere effektive Technik schnell umsetzen. Hierbei handelt es sich ebenfalls um eine halb-automatische Technik, um HTTP Requests einzusparen (siehe 4.2.2).

4.2.2 HTTP-Requests verringern – Konkatenierung

Beim betrachten des Wasserfalldiagramms (Abbildung 4.3) sieht man, dass der Browser nicht alle Dateien gleichzeitig herunterlädt.

Das Problem ist hierbei, dass der Browser nicht alle Dateien via HTTP gleichzeitig herunterladen kann, sondern immer nur eine bestimmte maximale Anzahl (Wenz & Hauser, 2013, S. 386). Aktuelle Browser unterstützen mehr als zwei gleichzeitige Verbindungen pro Server, wie zum Beispiel Google Chrome ab Version 4+ mit bis zu sechs gleichzeitigen Verbindungen pro Server[17]. Dies führt dazu, dass sich Downloads gegenseitig blockieren. Der Browser muss also immer erst warten, bis die Verbindungen zu einem Server wieder frei sind, um die nächsten Dateien herunterzuladen. Die Größe einer Datei stellt normalerweise bei

[16] Vgl.: http://farhadi.ir/projects/smartoptimizer/ (Stand: 24. März 2014)
[17] Vom W3-Konsortium wurden in den HTTP/1.1-Empfehlungen maximal 2 offene Verbindungen pro Server empfohlen um die HTTP Antwort Zeiten zu verbessern und dafür zu sorgen, dass es zu keiner Überlastung kommt. Vgl.: http://www.w3.org/Protocols/rfc2616/rfc2616-sec8.html#sec8.1.4 (Stand: 24. März 2014)

schnellen Internetverbindungen kaum noch ein Problem dar. Viel gravierender ist die Anzahl der HTTP Requests. Diese fallen bei einer mobilen Internetverbindung viel stärker ins Gewicht, da die Latenz höher ist.

Die Beschränkung simultaner Downloads bezieht sich auf die Verbindung zu einem Server, also von ein und derselben Domain, zum Beispiel `medinf.haw-aw.de`. Durch Verteilen der Dateien auf unterschiedliche Subdomains kann also die Anzahl gleichzeitiger Downloads erhöht werden, zum Beispiel auf `medinf1.haw-aw.de`, `medinf2.haw-aw.de` und `medinf3.haw-aw.de`. Diese Technik nennt sich »Domain Sharding« (Souders, stevesouders.com, 2009).

Besteht die Möglichkeit nicht, Dateien auf verschiedene Subdomains zu verteilen, müssen also HTTP Requests eingespart werden, um das Laden einer Webseite zu beschleunigen. Da meistens nicht einfach Elemente entfernt werden können, gibt es einen Weg um trotzdem HTTP Requests einzusparen: Konkatenation, also die Verkettung von Zeichen bzw. Zeichenketten. (Smashing Magazine, 2012, S. 194) Da eine Webseite normalerweise nicht nur eine Anfrage pro Dateityp stellt, ist hier in der Regel Optimierungspotential vorhanden.

Wie Tabelle 4.2 zeigt, werden bei der BaRESS Standard Webseite 5 JavaScript- und 3 CSS-Dateien geladen, wovon jeweils eine Datei für die Webtypografie von einem anderen Server geladen wird. Somit bleiben 4 JavaScript und 2 CSS Dateien. Durch Konkatenation dieser lassen sich also 4 HTTP Requests einsparen. Die Verkettung der Dateien könnte per Kopieren und Einfügen realisiert werden oder durch Nutzen des bereits eingesetzten Tools »SmartOptimizer«. Da die JS/CSS-Dateien sowieso schon die SmartOptimizer-Minifier durchlaufen, kann man durch kommagetrenntes Eintragen der Dateien im `<link>`-Tag die Dateien verketten. Der Aufruf ändert sich dann wie folgt:

```
<link type="text/css" rel="stylesheet"
href="css/bootstrap.css" />

<link type="text/css" rel="stylesheet"
href="css/carousel.css" />
```

Wird zu:

```
<link type="text/css" rel="stylesheet"
href="css/bootstrap.css,carousel.css" />
```

Für JavaScript-Dateien erfolgt die Änderung analog.

Das Ergebnis ist eine einzelne Datei. Beim Einsatz verketteter Dateien muss auf die Reihenfolge geachtet werden, gerade bei JavaScripts, die aufeinander aufbauen, wie beispielsweise jQuery-Plugins, die auf der jQuery-Bibliothek basieren. Sind die Dateien nicht in der korrekten Reihenfolge, kann dies zu Fehlern führen.

Um noch mehr HTTP Requests einzusparen, betrachten wir, welche weiteren Dateitypen in hoher Zahl vorhanden sind. Das sind normalerweise, genauso wie auf der BaRESS Webseite, Bilder. Bei Bildern, gerade bei wiederkehrenden User Interface Elementen, ist der Einsatz von CSS Image Sprites (siehe Abbildung 4.5) sinnvoll. Das sind Bilddateien, in der viele kleine Bilder nebeneinander gesammelt sind. (Smashing Magazine, 2012, S. 206)

Abbildung 4.5: CSS Sprite amazon.de Stand: 11.3.14

Per CSS wird ein Bildausschnitt ausgewählt, der sichtbar ist. Bei anderen Bildern, beispielsweise einer Bildergalerie, funktioniert diese Technik auch, sie ist jedoch sehr umständlich zu pflegen. Da die User Interface Elemente der BaRESS Webseite nicht durch Bilder realisiert sind, sondern durch CSS Regeln, erwähne ich diese Technik hier nur und setze diese nicht ein. Per CSS und CSS3 die Elemente zu gestalten, ist ebenfalls eine gute Möglichkeit, HTTP Requests einzusparen. Es kann jedoch dann durchaus sein, dass manche Geräte mit schlechterer CSS Unterstützung keine CSS Level 3 Features anzeigen, wie beispielsweise Verläufe, Multiple Backgrounds oder abgerundete Ecken. Werden diese Techniken eingesetzt, sollten diese in möglichst vielen unterschiedlichen Browsern getestet werden und unter Umständen Fallbacks anbieten, also Lösungen, auf die der Browser zurückgreifen kann, sofern er bestimmte Techniken nicht unterstützt.

Eine weitere Technik, Icons zu optimieren und gleichzeitig alle Bildschirmgrößen und hochauflösende Displays besser zu unterstützen, ist der Einsatz von Icon Fonts. Das sind Webtypografien, die nicht mit Rastergrafiken, sondern mit Vektorgrafiken arbeiten und somit beliebig skalierbar und farblich veränderbar sind (siehe Abbildung 4.6). Icon Fonts können alle möglichen Formen darstellen und fast beliebig viele Zeichen enthalten. Von Bootstrap wird standardmäßig die Icon Font »Glyphicons«[18] eingesetzt. Es lassen sich jedoch auch andere Schriften einbinden oder mittels »Adobe Illustrator« und einem zusätzlichen Programm, wie »Inkscape«[19], eigene Schriften erstellen (Pickering, 2012).

Abbildung 4.6: Screenshot Glyphicons Beispiel Skalierung und Farbe

4.2.3 Weniger übertragen – GZIP-Komprimierung

Um die bisher schon minifizierten und verketteten Dateien noch schneller zu übertragen, bietet sich der Einsatz der GZIP Komprimierung an. GZIP Komprimierung basiert auf dem »Deflate«-Algorithmus, einem Verfahren zur verlustfreien Datenkompression. Der Deflate-Algorithmus sucht zunächst nach redundaten Zeichenfolgen und ersetz diese. Danach folgt eine Entropiekodierung nach Huffman[20].

Eine GZIP-Komprimierung wird nur durchgeführt, wenn der Browser dem Server in der GET-Anfrage mitteilt, dass er GZIP Komprimierung unterstützt und fähig ist, die Daten zu entpacken. Ist serverseitig die GZIP-Komprimierung der Daten aktiviert, werden diese bei einer GET-Anfrage eines Clients zunächst auf dem Server gesucht, komprimiert und dann zum Client-Browser gesendet. Dieser entpackt die Daten automatisch und stellt sie dar (Smashing Magazine, 2012, S. 189f).

[18] Vgl.: http://getbootstrap.com/components/#glyphicons-glyphs (Stand: 24. März 2014)
[19] Vgl.: http://www.inkscape.org/de/ (Stand: 24. März 2014)
[20] Mehr Informationen: http://de.wikipedia.org/wiki/Deflate (Stand: 24. März 2014)

Aktivieren kann man die GZIP-Komprimierung, indem in der htaccess-Datei, den zu komprimierenden Dateien, ein Output Filter zugewiesen wird. So wird jede der eingetragenen Dateien zunächst komprimiert und dann übertragen. Die Änderung der htaccess-Datei kann wie folgt aussehen:

```
<IfModule mod_deflate.c>
AddOutputFilterByType DEFLATE text/html text/plain text/xml
text/css text/javascript application/javascript
</IfModule>
```

Ob die GZIP-Komprimierung auf einem Server aktiviert ist, kann man mit dem Online-Tool gidnetwork.com/tools/gzip-test.php testen. Im Falle der BaRESS Standard Webseite ist GZIP nicht aktiviert. Bei der optimierten Webseite ist sie durch die htaccess-Anpassung aktiv.

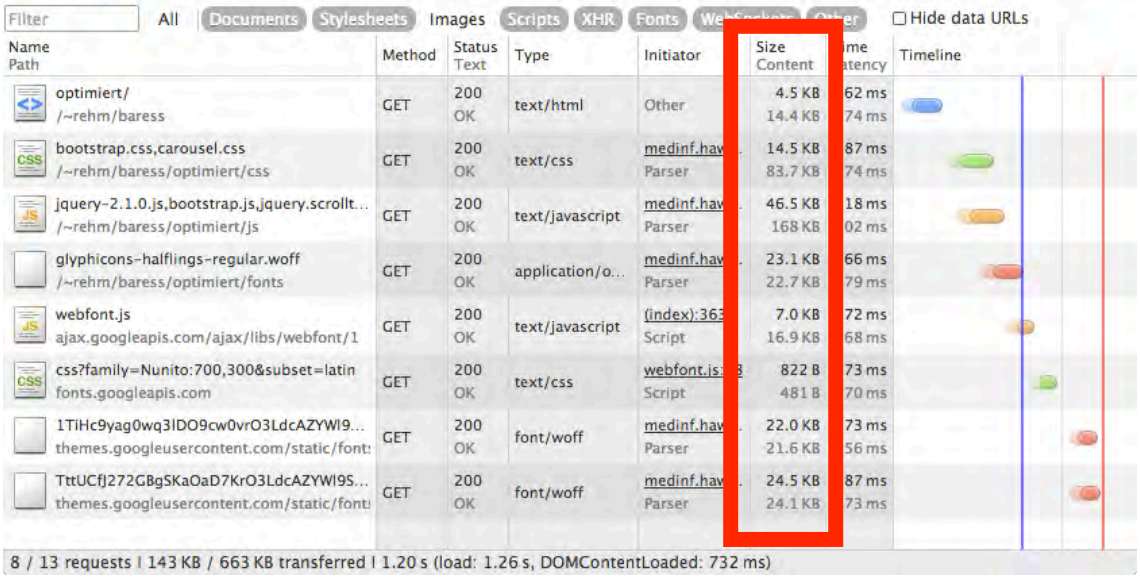

Abbildung 4.7: Screenshot Developer Tools BaRESS Optimiert

Betrachtet man die Netzwerkaktivität mithilfe der Entwicklerwerkzeuge eines Browsers, so sieht man genau, wie groß die übertragenen Dateien sind. In Abbildung 4.7 ist die Spalte »Size« rot markiert. Pro Datei werden zwei Werte ausgegeben; der Obere steht jeweils für die heruntergeladene Dateigröße und der Untere für die Dateigröße nach Entpacken durch den Browser. Bei Analysieren der Requests sieht man, dass nur die ersten vier GET-Anfragen auch von unserem eigenen Server beantwortet werden. Nur bei diesen Dateien kann die Komprimierung gesteuert werden. 88,6 kB werden gezipped vom Server geladen. 288,8 kB sind die Dateien nach dem Entpacken groß. Also werden im Fall der BaRESS Optimiert Webseite durch Einsetzen der GZIP-Kompression fast 70% der bei der Übertragung komprimierter Dateien eingespart.

4.2.4 Schnellerer erster Webseitenaufruf – Lazy Loading

Der erste Webseitenaufruf ist entscheidend für die User Experience, gerade der Ausschnitt, der vom User als erstes gesehen wird. Im Normalfall ist dies der oberste Abschnitt der Webseite. Die Elemente, die direkt sichtbar sind, wenn die Webseite aufgerufen wird, sind unerlässlich. Jeder weitere Inhalt, der erst später zu sehen sein wird, ist nicht ganz so wichtig. Nur durch Scrollen sichtbar werdende Elemente, beispielsweise Bilder, können erst später geladen werden. Diese Technik wird »lazy loading« genannt (Zillgens, 2013, S. 345). Im Falle der BaRESS Webseite sind die drei Bilder, die sich im Carousel drehen, auf jeden Fall immer beim Seitenaufruf zuerst sichtbar, alle weiteren Bilder nicht unbedingt. Mithilfe des kleinen JavaScript-Plugins »unveil.js«[21] ist es einfach, Bilder per lazy loading nachzuladen. Dazu muss die JavaScript-Datei »unveil.js« geladen und das img-Tag wie folgt abgeändert werden:

```
<img src="img/bsp_img.jpg">
```

Wird zu:

```
<img src="img/loader.gif" data-src="img/bsp_img.jpg">
```

Die Seite lädt nun beim Aufruf zunächst ein kleines animiertes GIF, um dem User zu signalisieren, dass noch etwas passiert. Sobald der User scrollt und das gewünschte Bild einen bestimmten Abstand zum sichtbaren Bereich erreicht hat, wird per JavaScript das Bild, das im data-src-Attribut enthalten ist, nachgeladen und gegen das loader.gif ausgetauscht.

Durch den Einsatz einer lazy-loading-Technik können bei der BaRESS Webseite die 5 HTTP Requests und ca. 1,7 MB beim ersten Laden der Webseite eingespart werden.

4.2.5 Caching & Expires

Die bisherigen Optimierungen beziehen sich hauptsächlich darauf, den ersten Webseitenaufruf zu optimieren. Ist die Entwicklung einer Webseite abgeschlossen, werden sich die meisten Dateien kaum noch ändern. Um bei einem erneuten Aufrufen der Webseite nicht alles neu zu laden, sollten die Dateien clientseitig abgelegt werden. Damit der Client-Browser Dateien im Cache ablegt, muss im HTTP Header Feld »Expires« (dt. ablaufen) eine Ablaufzeit eingetragen werden. Diese Einstellung wird in der htaccess-Datei vorgenommen. Für jeden Dateityp lässt sich eine individuelle Verfallszeit eintragen. Somit kann genau bestimmt werden, nach welcher Zeit der Browser erst wieder überprüfen soll, ob sich eine Datei geändert hat oder nicht. Genauso kann man eintragen, dass bestimmte Dateitypen nie im Cache gespeichert werden sollen, beispielsweise dynamische Dateien oder HTML Dateien, deren Inhalte regelmäßig aktualisiert werden (Zillgens, 2013, S. 343).

[21] Vgl.: http://luis-almeida.github.io/unveil/ (Stand: 24. März 2014)

Bei der BaRESS Webseite habe ich die Einstellungen wie folgt vorgenommen:

```
<IfModule mod_expires.c>
ExpiresActive On
ExpiresDefault "access plus 30 days"
<FilesMatch "\.(jpg|jpeg|png|gif|swf|ico|svg)$">
ExpiresActive On
ExpiresDefault "access plus 30 days"
Header set Cache-Control "access plus 30 days, public"
</FilesMatch>
<FilesMatch "\.(php|cgi|pl|htm?l)$">
ExpiresActive OffHeader set Cache-Control "private, no-cache,
no-store, proxy-revalidate, no-transform"
Header set Pragma "no-cache"
</FilesMatch>
</IfModule>
```

Zunächst wird die Expires-Funktion für alles eingeschaltet und die Default-Ablaufzeit auf 30 Tage ab Abrufdatum eingestellt, für alle Bilder oder Grafiken ebenfalls. PHP, HTML und weitere Inhalte sollen nicht gecached werden, um stets die aktuellsten Dateien vom Server geschickt zu bekommen.

Die Information, ob eine Datei gecached werden soll oder nicht, wird vom Server im Response Header jeder Datei eingetragen und kann in den Webdeveloper Tools ausgelesen werden (siehe Abbildung 4.8). Bei erneutem Webseitenaufruf schickt der Browser im Request-Header die Informationen zu der Datei mit, die im Cache abgelegt sind. Der Server prüft dann, ob sich die angefragte Datei geändert hat. In diesem Beispiel antwortet der Server, dass sich die Datei nicht verändert hat (oberer roter Kasten). Im unteren Teil sieht man den Response-Header, den der Server mit jeder Datei mitgibt. In ihm befindet sich die Information, ob eine Datei gecached werden soll oder nicht (unterer roter Kasten). So müssen bei erneuten Webseitenaufrufen viel weniger Daten übertragen werden, da alles was gecached wurde, aus dem Cache gelesen werden kann. Es kommt zu weniger Traffic und die Webseite lädt schneller.

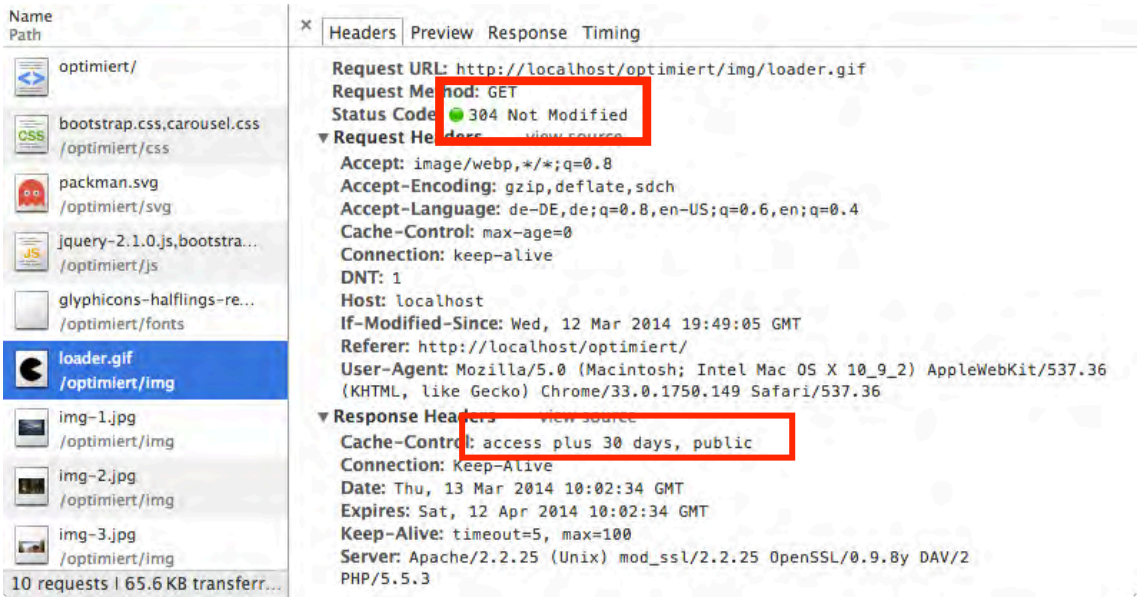

Abbildung 4.8: Screenshot Beispiel HTTP Header Webdeveloper Tools

Werden vor Aktivieren des Browsercaching bei Aufruf bzw. beim erstmaligen Aufruf der BaRESS Optimiert Webseite noch ca. 655 kB (ohne die später per Lazy Loading geladenen Bilder) vom Server übertragen, so werden bei aktiviertem Browsercache bei jedem weiteren Aufruf der Webseite nur noch 4,5 kB übertragen (siehe Abbildung 4.9).

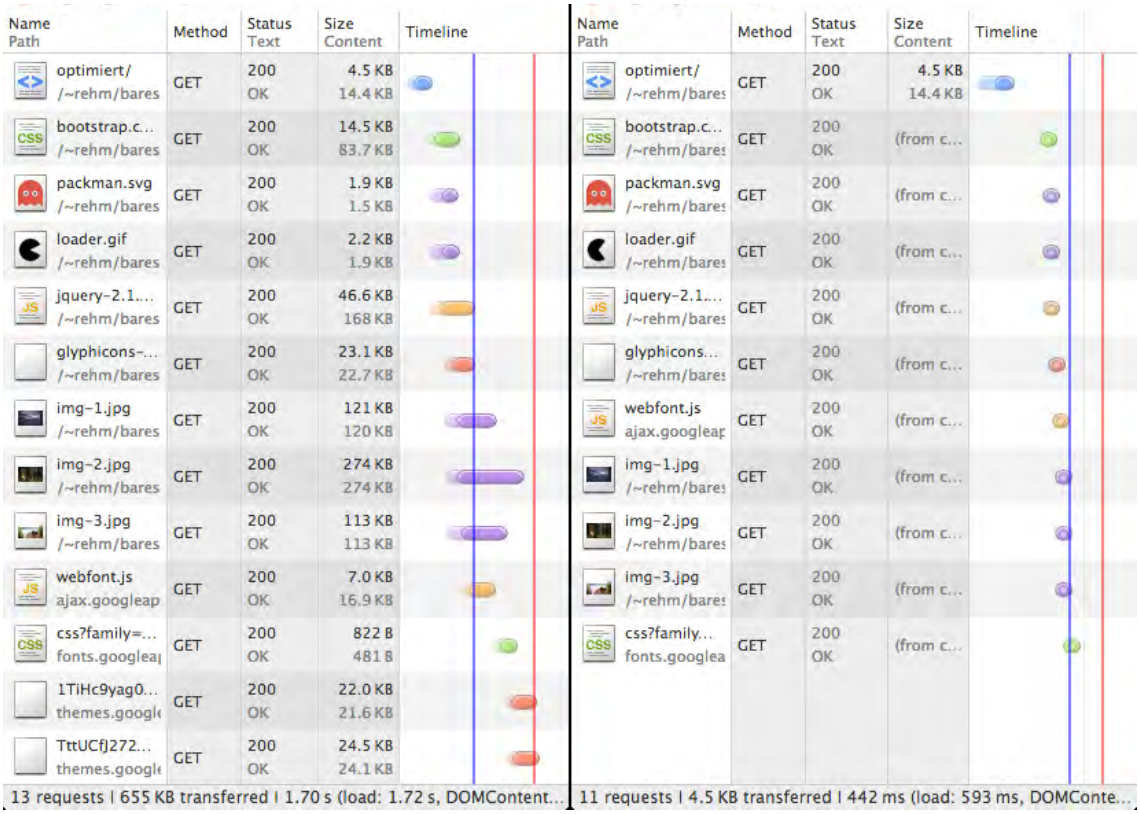

Abbildung 4.9: Screenshot Vergleich no-cache (l.S.) <> cache (r.S.)

4.2.6 CSS und JS richtig anordnen

Damit der Browser eine Webseite darstellen kann, sind CSS- und JS-Dateien wichtig, da diese das Aussehen der Webseite beschreiben. Während des Ladens von CSS und JS verhält sich der Browser anders als beim Laden von beispielsweise Bildern, da diese nur dargestellt und nicht interpretiert werden müssen.

Bevor eine CSS-Datei nicht fertig geladen und interpretiert ist, wird der Browser nichts anzeigen und der User sieht nur einen weißen Bildschirm. Um diese Zeit möglichst kurz zu halten, ist es wichtig, die CSS-Datei so früh wie möglich zu laden. Die Platzierung der CSS-Verlinkung ist im Head-Bereich des HTML-Dokuments richtig, wie auch schon seit vielen Jahren gelehrt wird.

Allerdings empfehlen viele Hersteller von JS-Plugins ebenfalls, diese im Head des HTML-Dokuments zu referenzieren. Um jedoch alle Inhalte einer Webseite darzustellen, würde der Browser solange warten, bis alle im Head geladenen JavaScripte komplett heruntergeladen und ausgeführt sind. Während dieser Zeit sind alle weiteren Downloads blockiert. Dies liegt daran, dass per JavaScript das HTML und CSS der Webseite verändert werden kann. Der Browser muss erst genau wissen, was das JavaScript verändert, damit keine Elemente falsch dargestellt werden. Um das Blockieren zu verhindern, sollten JavaScript im HTML so weit unten wie möglich platziert werden, damit der Browser zunächst die Webseite aufbauen kann, während die JavaScripts in Ruhe laden können. (Smashing Magazine, 2012, S. 199f)

4.2.7 Fazit

Die Optimierungstechniken der bisherigen Abschnitte erfordern nur leichte Veränderungen des bestehenden Codes der BaRESS Standard Webseite. Die Verbesserungen, die dadurch erreicht werden, sind jedoch hoch. Der Vergleich der beiden Varianten mithilfe des GTmetrix Tools zeigt, dass bei der optimierten Variante 8 HTTP Requests eingespart werden konnten, wovon später per LazyLoad noch 5 Bilder nachgeladen werden. Außerdem wurde die Downloadgröße der Webseite beim ersten Aufrufen von 5,2 MB auf 668 kB reduziert. Mit nachgeladenen Bildern beträgt die Gesamtgröße der Webseite trotzdem nur 2,1 MB.

Hier sind die GTmetrix-Tests zu finden:

Standard-Variante: http://gtmetrix.com/reports/medinf.haw-aw.de/e72o1Fmm

Optimierte-Variante: http://gtmetrix.com/reports/medinf.haw-aw.de/SfoKYEyM

4.3 RESS BaRESS Webseite

Die bisher durchgeführten Veränderungen und Erweiterungen optimieren den Code und viele Komponenten sehr gut für einen normalen Webseitenaufruf von einem Gerät mit guter Leistung und guter Internetverbindung. Wird jedoch von einem älteren Smartphone mit einer langsamen mobilen Internetverbindungen die Webseite aufgerufen, so können die JavaScripts, CSS-Regeln und Fonts das Gerät an seine Leistungsgrenzen bringen oder unter Umständen manche Inhalte gar nicht darstellen.

Ein Beispiel: Ein User besucht die BaRESS Webseite mit einem Handy mit Browser, einem sogenannten Feature Phone. Da das Handy eigentlich auf Telefonie und die »normalen« Funktionen eines Mobiltelefons ausgelegt ist, fehlt es an Leistung, um die CSS- und JS-Datei in angemessener Zeit zu parsen. Das Gerät muss sehr viele CSS-Regeln parsen, die es gar nicht benötigt und somit unnötig viel Zeit dafür aufbringen. Zusätzlich lädt es alle Bilder in einer Größe, die es gar nicht unterstützen kann, da diese für einen Desktop-/Laptop-Bildschirm ausgelegt sind.

Ein weiteres Beispiel: Ein User mit einem älteren Smartphone, ca. 3 Jahre alt mit veraltetem Android 2.2.2, besucht die BaRESS Webseite (Standard oder Optimiert). Ihm wird die Pacman-SVG-Datei nicht angezeigt, da das Smartphone dieses Format noch nicht unterstützt. (siehe Abbildung 4.10)

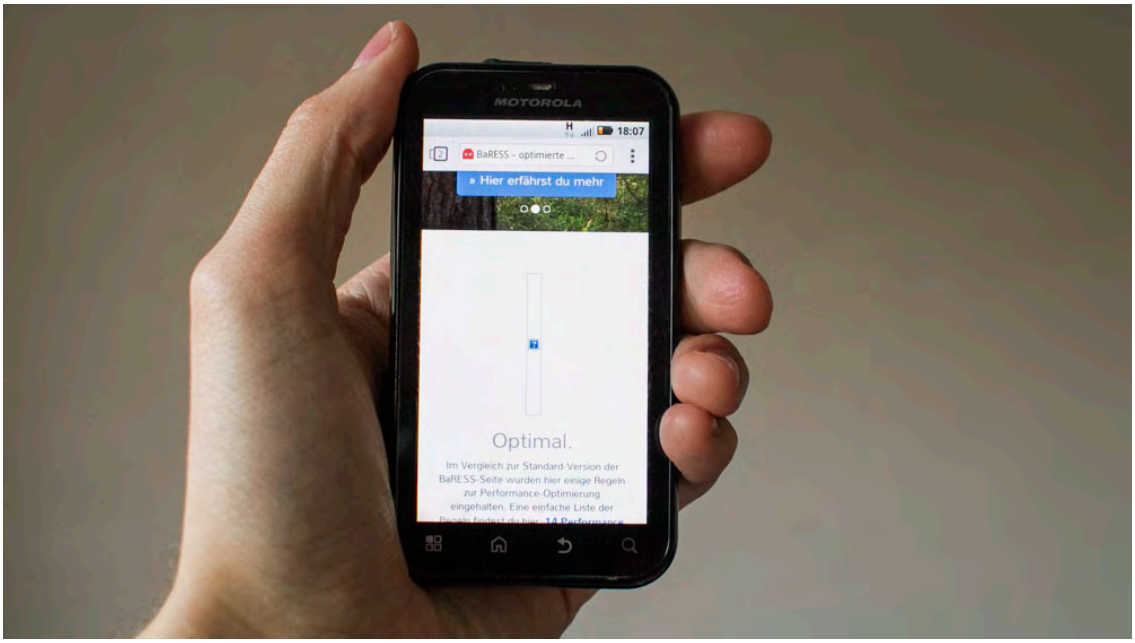

Abbildung 4.10: Beispiel altes Android-Smartphone ohne SVG-Unterstützung

Die Beispiele machen deutlich, welche Probleme durch den Einsatz von Responsive Webdesign und neuen Technologien entstehen können. Manche Geräte können somit nicht optimal beliefert werden, auch wenn das eigentlich der Grundgedanke des Responsive Webdesign ist, zumindest alle Bildschirmgrößen abzudecken. Ist nicht bekannt, mit was für einem Gerät eine Webseite

besucht wird, kann man nie sicher sein, dass dieses alle Funktionen der Webseite komplett unterstützt. Eine bessere Belieferung der unterschiedlichen Geräte ist nur mit dem normalen Responsive Webdesign Ansatz, das Layout an die Bildschirmgrößen mittels CSS Media Queries anzupassen, nicht möglich.

An diesem Punkt setzt das RESS-Konzept an. Wie in Abschnitt 3.3 schon vorgestellt, kombiniert RESS die Vorteile einer mobilen Webseite und Responsive Webdesign. RESS ist also ein System, das mittels reaktionsfähigem Layout sich an die Bildschirmgrößen anpassen kann und durch serverseitiges Erkennen des Geräts und dessen Fähigkeiten, eine hohe Optimierbarkeit bietet. Wie das in der Praxis umgesetzt werden kann, behandeln die folgenden Abschnitte.

4.3.1 Device Detection

Um ein Gerät zu identifizieren, also eine sogenannte »Device Detection« durchzuführen, gibt es zwei Methoden: »Browser Detection« und »Feature Detection«. Durch die Device Detection lassen sich Geräte klassifizieren und der Geräteklasse oder Funktionen eines Geräts entsprechend Code liefern. Beide Methoden haben ihre Stärken und Schwächen und sind alleine nicht wirklich verlässlich einzusetzen. (Smashing Magazine, 2012, S. 214f)

4.3.1.1 Browser Detection – WURFL

Die erste Methode nennt sich »Browser Detection« und bezeichnet den Vorgang, den Browser zu bestimmen, von dem eine Webseiten-Anfrage gestellt wird. Das ist zunächst einmal nichts Neues und wurde schon lange eingesetzt, um Browser zu erkennen. Vor einigen Jahren war die Entwicklung der Browser noch nicht so weit fortgeschritten und die Browser unterschieden sich stark in ihren Funktionen. Es gab die Praxis, durch Browser Detection den Browser zu erkennen und falls dieser bestimmte Funktionen noch nicht unterstützte, diese einfach weg zu lassen. Radikaler war die Praxis, dem User eine Meldung auszugeben, dass sein Browser nicht unterstützt werde und dass er sich bitte einen der unterstützten Browser installieren solle. Dadurch hat der Ruf der Browser Detection stark gelitten. Mittlerweile kann diese Technik jedoch sinnvoll zur Einstufung genutzt werden, ob eine Mobile- oder eine Desktop-Variante der Webseite ausgeliefert werden soll.

Die Browser Detection ist durch Auswerten des »User Agent« Strings (UA) möglich, auch »User Agent Sniffing« genannt (Andersen, 2008). Bei jeder Server-Anfrage werden Browser-Informationen im HTTP Header übertragen. In der Abbildung 4.8 ist ein HTTP Header abgebildet. Im Request Header Abschnitt findet man an letzter Stelle den UA. In diesem Fall werden vom Browser die folgenden Informationen übertragen:

```
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_2)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/33.0.1750.149
Safari/537.36
```

Der UA stellt Informationen über den Browser und das Betriebssystem des Geräts zur Verfügung. Aus den Informationen kann die Geräteart abgeleitet werden, in diesem Fall ein Macintosh-Computer. Interessant ist dabei die Möglichkeit mithilfe der Information zu bestimmen, ob es sich um ein mobiles Gerät handelt oder nicht. Darüber lässt sich beispielsweise eine Weiterleitung zu einer mobilen Webseite einrichten. Um mithilfe des UA weitere Informationen über ein Gerät zu erhalten, muss ein Abgleich des UA mit einer Bibliothek vorgenommen werden, einer sogenannten »Device Detection Repository« (DDR). Hier sind Informationen über viele Eigenschaften eines Geräts hinterlegt, insofern dieses der DDR bekannt ist und eingepflegt wurde. Eine der größten kommerziellen DDRs ist »WURFL«. Eigentlich handelte es sich um ein OpenSource-Projekt, das im Jahr 2001[22] ins Leben gerufen wurde. Mittlerweile wird WURFL von der Firma ScentiaMobile[23] betrieben und ist mit unterschiedlichen Lizenzplänen verfügbar. WURFL ist ein Akronym für »Wireless Universal Resource File«. Die DDR ermöglicht es, viele unterschiedliche Funktionen aus dem Repository abzufragen. (Kadlec, 2013, S. 213ff)

Die Browser Detection wird im RESS-Konzept allerdings nicht eingesetzt, um die Anfrage auf eine mobile Webseite umzuleiten, sondern um kritische Funktionen einer Webseite an eine Abfrage zu binden. Kritische Funktionen sind solche, die unter Umständen nur von neuesten Geräten und Browsern unterstützt werden und durch deren Ausfall eventuell die User Experience stark leiden kann. Am Anfang des Kapitels habe ich in einem Beispiel von einem User geschrieben, dem SVG-Grafiken auf einem älteren Smartphone nicht angezeigt werden. Dies wäre eine solche Problematik und lässt sich mittels UA und WURFL gut lösen.

Um WURFL in der RESS-Variante der BaRESS Webseite zu implementieren, habe ich mich für die WURFL-Cloud-Lösung entschieden. Diese ist mit bis zu 5 unterschiedlichen Capability-Tests und 5000 Abfragen pro Monat kostenlos. Zur Nutzung ist eine Registrierung bei ScentiaMobile notwendig, um einen API-Key zu erhalten.

Nach der Registrierung kann man online 5 aus 527 Capability-Tests auswählen (siehe Abbildung 4.11), die dem API-Key zugewiesen werden.

Unter scientiamobile.com/wurflCapability findet sich eine detaillierte Liste aller Tests. Ich habe mich unter anderem für den Test entschieden, ob das Gerät SVG-Grafiken anzeigen kann (siehe Abbildung 4.11)

[22] Vgl.: http://wurfl.sourceforge.net/faq.php (Stand: 25. März 2014)
[23] Vgl.: http://scientiamobile.com/cloud (Stand: 24. März 2014)

Abbildung 4.11: Screenshot scentiamobile.com Free WURFL-Cloud; ausgewählte Capabilites

Die zu testende Variable heißt `svgt_1_1` und gibt einen booleschen-Wert zurück, also `TRUE` oder `FALSE`. Um die WURFL-Cloud nutzen zu können, werden die Dateien zur Registrierungsabfrage der Cloud-Lizenz auf den Server geladen und der API-Key im Projekt eingetragen. Mit einer Abfrage des Tests können jetzt die DeviceCapabilitys, also die Fähigkeiten eines Geräts, abgefragt werden.

```
if ($client->getDeviceCapability('capability') {do this;}
else {do that;}
```

Im Folgenden wird getestet, ob das Gerät SVG-Grafiken unterstützt. Ist das Ergebnis positiv, so wird der Variable `$svgtpng` der String `svg` zugewiesen, andernfalls `png`. Dadurch wird das Dateisuffix der Grafik ausgetauscht.

```
if ($client->getDeviceCapability('svgt_1_1'))
{$svgtpng = "svg";} else {$svgtpng = "png";}
```

```
<img src="<?= $svgtpng ?>/packman.<?= $svgtpng ?>" width="auto"
height="200px">
```

Der Einsatz der WURFL-Cloud ist gut umzusetzen und durch die vielen unterschiedlichen Testmöglichkeiten ist das Tool sehr vielseitig einsetzbar. Eine sehr interessante Testmöglichkeit der WURFL-Cloud ist die Abfrage des XHTML-Levels eines Geräts. Die Test-Variable `xhtml_support_level` liefert beispielsweise Informationen darüber, wie gut die CSS-Unterstützung eines Geräts ist. Dadurch ließen sich beispielsweise für ältere Geräte einfache HTML-Varianten einer Webseite ausliefern.

Einen großen Nachteil der DeviceDetection mittels User Agent Sniffing gibt es allerdings: Da in den meisten Browsern die Möglichkeit vorhanden ist den UA umzustellen, ist die Geräteerkennung mittels UA nicht sehr zuverlässig. Nicht nur die Entwicklertools von diversen Desktopbrowsern unterstützen diese Funktion, sondern auch mobile Browser (siehe Einstellungen diverser Browser). Damit wird UA Sniffing noch weniger verlässlich.

4.3.1.2 Feature Detection – Detector

»Feature Detection« ist die zweite Methode, um die Eigenschaften eines Geräts zu bestimmen. Clientseitig wird ein JavaScript mit diversen Tests ausgeführt und so Informationen über das Gerät bereitgestellt. Dadurch lassen sich Informationen abfragen, die im UA nicht übertragen werden, beispielsweise die Bildschirmgröße oder welche CSS Funktionen der Browser unterstützt. Da diese Information aber nur auf Clientseite zur Verfügung steht, lässt sich durch die Abfrage keine Code-Optimierung auf der Serverseite durchführen, um zum Beispiel weniger Daten übertragen zu müssen. Wird jedoch auf Serverseite eine Datenbank angelegt und die Funktionen, die durch das JavaScript getestet wurden, mittels Cookie an den Server übertragen, so lässt sich auch vor Übertragen des Codes zum Client eine serverseitige Anpassung durchführen (Kadlec, 2013, S. 204f).

»Detector«[24] ist ein Tool von Dave Olsen, das genau diesen Ansatz umsetzt. Es steht unter einer OpenSource Lizenz zur Verfügung und ist eine Kombination diverser OpenSource Projekte. Es basiert auf den beiden Projekten »UA-Parser«[25] und »modernizr-server«[26] und ist eine Lösung zur Device Detection. Die Ergebnisse aus den Feature Tests mittels modernizr-server werden mit dem User Agent String verknüpft in einer Datenbank auf dem Server abgespeichert. So hat das System den großen Vorteil, dass es sich selbst aktualisiert und erweitert, je nachdem, welche Geräte registriert werden und ob die modernizr-Tests ausgeführt werden können. (Kadlec, 2013, S. 205f)

Um Detector zu nutzen, lädt man den aktuellsten Release (Version 0.9.5 zum Zeitpunkt der Bachelorarbeit) von Olsens GitHub Repository[27] und kopiert die Bibliothek auf den Server. Mittels `require_once "Detector/Detector.php";` wird Detector im Projekt eingebunden. Detector übergibt automatisch alle Information, die aus den Tests hervorgehen, an das Array `$ua`. Aus dem Array kann dann ausgelesen werden, ob ein Feature `TRUE` oder `FALSE` ist. Dazu benötigt man nur noch den spezifischen Feature Namen. Eine Liste aller Detector-Tests findet sich auf der Projektwebseite detector.dmolsen.com.

```
if ($ua->featureName) {do this;} else {do that;};
```

Durch die Tests und Informationen, die aus dem UA ausgelesen und durch die modernizr-Tests abgefragt werden, ordnet Detector das getestete Gerät einer Geräte-Familie zu: Mobile, Tablet oder Computer. Beispielsweise könnte man mithilfe der Geräte-Familie diverse Funktionen austauschen oder bestimmte Code-Teile anpassen.

Um eine Webseite für mobile Geräte so zu optimieren, dass auf diesen weniger HTTP Requests gestellt werden und die Seite somit langsamer machen, könnte durch die Abfrage der Geräte-

[24] Vgl.: http://detector.dmolsen.com (Stand: 24. März 2014)
[25] Vgl.: https://github.com/tobie/ua-parser (Stand: 24. März 2014)
[26] Vgl.: https://github.com/jamesgpearce/modernizr-server (Stand: 24. März 2014)
[27] Vgl.: https://github.com/dmolsen/Detector (Stand: 24. März 2014)

Familie ein Google-Maps-iFrame durch eine statische Google-Maps-Karte, die zu Google-Maps verlinkt, ersetzt werden. So würden mit nur einem HTTP Request die Karteninformationen bereitgestellt werden und durch Klick auf die Karte hätte der Nutzer die volle Google-Maps Funktionalität zur Verfügung. Der folgende Code-Schnipsel würde genau das durchführen.

```
$maps_iframe = '<iframe ... ></iframe>';zum
$maps_static = '<a ... ><img ... /></a>';
if ($ua->isMobile) {echo $maps_static;}
else {echo $maps_iframe};
```

Um zu verdeutlichen, wie viel durch solche Lösungen eingespart werden könnte, habe ich die kleine Testwebseite medinf.haw-aw.de/~rehm/baress/ress/ress_sample.php erstellt. Mittels Detector wird das Gerät erkannt und kann auf Wunsch umgeschaltet werden. So werden bei Aufruf durch einen Desktopbrowser ca. 113 HTTP Requests gestellt, bei Aufruf durch ein mobiles Gerät sind es lediglich ca. 54 HTTP Requests. Diese Verbesserung wird erreicht, indem bestimmte Funktionen ausgetauscht werden, wie oben beschrieben. Zum einen wird der Google-Maps-iFrame durch eine statische Google-Maps-Karte ausgetauscht, zum anderen wird im Facebook-Plugin der Data-Stream deaktiviert und somit nicht mehr der komplette Facebook-Feed der verlinkten Seite geladen. Als weitere Optimierung könnte zum Beispiel auf einem einfachen Handy nur noch ein normaler Link zur Facebook-Seite geladen werden.

Größter Nachteil der Feature Detection und damit auch von Detector: Ist JavaScript im Browser deaktiviert, lässt sich keine Feature Detection mithilfe eines JavaScripts durchführen. Somit ist die Methode ebenfalls nicht vor Ausfällen geschützt. In diesem Fall greift Detector auf die aufgebaute Bibliothek zurück und ruft die Informationen ab, die mit dem UA verknüpft und auf dem Server abgelegt sind.

Detector bietet bei weitem nicht den Funktionsumfang und den Komfort, den eine kommerzielle WURFL-Lizenz bietet (Kadlec, 2013, S. 226). Dafür ist es kostenlos verfügbar und somit für kleine Projekte durchaus interessant.

4.3.2 AdaptiveImages

Ein viel diskutiertes Problem der Entwicklung für mobile Geräte mit Responsive Webdesign ist die Frage danach, wie Bilder dem Gerät entsprechend geladen werden können. Durch die vielfältigen Möglichkeiten, die RESS bietet, ist es durchaus möglich, ein System zu schaffen, das die Fähigkeiten eines Geräts erkennt und somit die optimalen Bilder lädt. Dadurch wäre es auch möglich, für unterschiedliche Bildschirmauflösungen die Bilder auf einen wichtigen Bildausschnitt zu beschneiden (siehe Abbildung 4.12). Auf mobilen Geräten besteht die Gefahr, dass durch die Skalierung der Bilder eventuell der eigentliche Inhalt eines Bildes in den Hintergrund gerät.

Abbildung 4.12: Beispiel alternativer Bildausschnitt (eigene Darstellung)

Die Umsetzung eines solchen Ansatzes wäre mit den vorgestellten Techniken auch nicht allzu aufwendig, viel aufwendiger wäre die Pflege der Bilder. Diese müssten in unterschiedlichen Auflösungen, Formaten und Zuschnitten bereitgestellt und organisiert werden. Auch bei Verzicht auf die Optimierung des Bildausschnitts, wäre die Pflege für ein solches selbst geschriebenes System aufwendig.

Viel effektiver ist der Einsatz eines Tools, das die Generierung der Bilder für unterschiedliche Auflösungen übernimmt. »AdaptiveImages«[28] (AI) von Matt Wilcox ist ein PHP-Skript, das genau diese Funktionalität bereitstellt (Krenz-Kurowska & Kurowski, 2013, S. 70ff). Durch eine Änderung der htaccess-Datei werden alle Bildanfragen an das PHP-Skript verwiesen, außer Bildanfragen aus explizit ausgenommenen Ordnern (mittels `RewriteCond %{REQUEST_URI} !assets`).

```
<IfModule mod_rewrite.c>

Options +FollowSymlinks
RewriteEngine On
RewriteRule \.(?:jpe?g|gif|png)$ adaptive-images.php

</IfModule>
```

Mithilfe eines kurzen JavaScript-Befehls wird die Bildschirmhöhe und –breite des Geräts ausgelesen und in einem Cookie an das AI-Skript übergeben.

[28] Vgl.: http://adaptive-images.com (Stand: 24. März 2014)

```
<script>
document.cookie = 'resolution=' +
Math.max(screen.width,screen.height) + '; path=/';
</script>
```

Im AI-Skript sind mehrere Breakpoints für Bildschirmauflösungen festgelegt, anhand derer die Bilder generiert und die Geräte unterteilt werden. Dadurch muss nicht für jede individuelle Bildschirmauflösung ein neues Bild generiert werden.

Das AI-Skript wertet nun die Bildanfrage aus und sucht das Bild aus dem Originalverzeichnis. Dann wird entsprechend der übertragenen Bildschirmauflösung das neue Bild generiert und in einem Cache-Verzeichnis auf dem Server abgelegt. Dadurch muss die Generierung nicht bei jeder Bildanfrage mit der gleichen Bildschirmauflösung neu durchgeführt werden. Das Bild wird in einem solchen Fall aus dem Cache geladen, insofern sich dieses nicht verändert hat. AI bietet die Möglichkeit, unterschiedliche Bildschirmgrößen für Bildgenerierung sowie die Bildqualität der komprimierten Bilder einzustellen und ebenso dem HTTP Header mitzuteilen, dass und wie lange der Browser die Bilder im Cache speichern und abrufen soll (siehe Kapitel 4.2.5 Caching & Expires).

Durch den Einsatz von AI können abhängig von der Bildschirmgröße Bilder generiert und geladen werden, ohne dass dies zu wiederkehrendem Mehraufwand für Entwickler oder Redakteure einer Webseite führt. Dadurch kann die Webseite schneller geladen, Traffic eingespart und mobile Datenverträge geschont werden. Bei der RESS-Variante der BaRESS Webseite wird AdaptiveImages eingesetzt. Dadurch werden die Bilder noch weiter komprimiert und auf dem Gerät werden weniger Daten heruntergeladen.

Um genau zu untersuchen, was von einem mobilen Gerät geladen wird, ist der Aufwand etwas größer als das Öffnen der Web Developer Tools eines Browsers. Diese Funktion ist in mobilen Browsern nicht gegeben und so muss ein kleiner Umweg gegangen werden. Eine einfache Methode ist das Testen der Webseite in einem online Webperformance-Testing-Tool, wie »Mobitest«[29] von Akamai. Der kostenlose Service bietet an, eine Webseite auf einem simulierten Smartphone oder Tablet aufzurufen und die Zeit und Downloadgröße der Webseite auf dem Gerät zu messen. Für einfache Tests ist dieses Tool unter Umständen ausreichend, gerade für Tests, die die Ladezeit einer Webseite betreffen.

Ein besserer Weg, um den Ladevorgang genauer entschlüsseln zu können, bieten die Web Developer Tools moderner Browser mit dem Feature »Remote Debugging«. Um beispielsweise einen solchen Test in Apples Browser »Safari« durchzuführen, benötigt man Safari sowie den kostenlosen iOS-Simulator, der in den Xcode Developer Tools von Apple enthalten ist. Xcode

[29] Vgl.: http://mobitest.akamai.com/m/index.cgi (Stand: 24. März 2014)

ist im Apple App Store frei verfügbar. Ruft man im Simulator eine Webseite auf, kann diese mit den Developer Tools von Safari untersucht werden (siehe Abbildung 4.13).

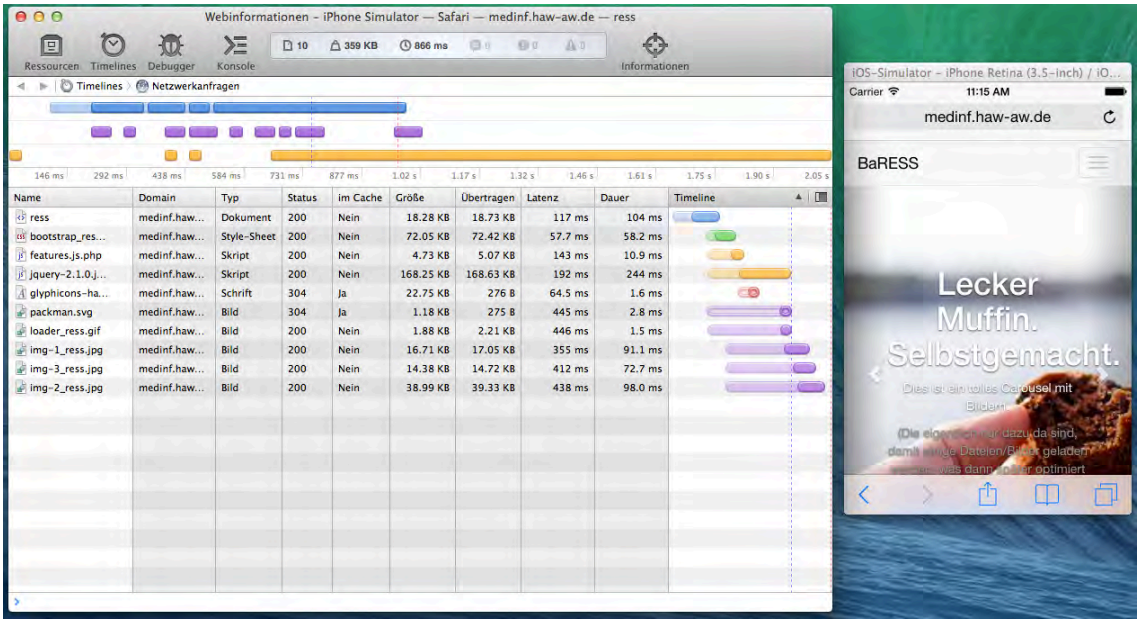

Abbildung 4.13: Screenshot iOS-Simulator/Safari WebDevTools BaRESS – RESS

Vergleicht man mithilfe dieses Tools die beiden Varianten RESS (siehe Abbildung 4.13) und Optimiert (siehe Abbildung 4.14), stellt man einen signifikanten Unterschied fest. Wie erwartet lädt die RESS Variante mittels AdaptiveImages kleinere Bilder vom Server als die Variante ohne AI. Das Bild »img-1.jpg« beispielsweise wird bei der optimierten Variante in der Auflösung 1400 x 933px und der Größe 120,49KB geladen. Bei der RESS Variante wird das Bild in der Auflösung 480 x 320px und der Größe 16,71KB geladen. Als iOS-Simulator Test-iPhone diente ein Retina iPhone mit 3,5-Zoll-Display.

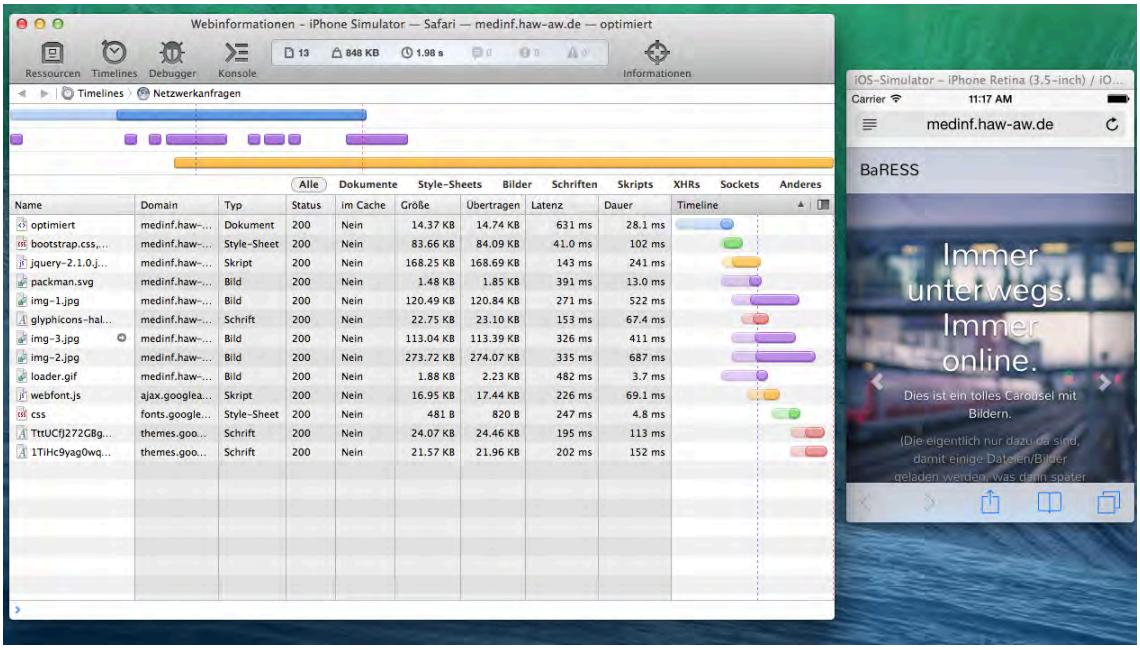

Abbildung 4.14: Screenshot iOS-Simulator/Safari WebDevTools BaRESS – Optimiert

Um AdaptiveImages einsetzen zu können, müssen die benötigten Rechte auf dem Server vergeben werden, damit das Skript die Bilder generieren und cachen kann. Ist das nicht möglich, ist der Einsatz von AdaptiveImages auch nicht möglich.

4.3.3 Chancen durch RESS

Für Webentwickler und Webdesigner bietet der Einsatz eines RESS-Systems eine Fülle an neuen Möglichkeiten die User Experience auf unterschiedlichen Geräten zu verbessern. Dabei geht es keinesfalls ausschließlich um Performance Optimierung durch RESS. Viel spannender könnte RESS zur Nutzung geräteabhängiger Features werden. Hier sind so gut wie keine Grenzen gesetzt. Beispielsweise der Einsatz von Funktionen, die auf ortsabhängige Daten zurückgreifen, könnte bei einer Webseite zu verbesserter Interaktivität führen. Für Kontaktinformationen auf einer Webseite bietet sich ebenso an, direkt mit den Telefonfunktionen eines Smartphones zu arbeiten. Wenn in Zukunft der Zugriff auf weitere Gerätefeatures aus dem Browser heraus möglich werden würde, könnten Webseiten eine unvorstellbare Breite an Funktionen entwickeln, ähnlich den Möglichkeiten, die bisher nur durch auf dem Gerät installierte Apps möglich sind.

Allerdings ist RESS nicht nur eine Mobile vs. Desktop Lösung, sondern bietet für Desktop-Browser die gleiche Optimierbarkeit. Bei Desktop-Browsern gibt es immer noch signifikante Unterschiede und die Unterstützung veralteter Browser wird auch weiterhin ein großes Thema bleiben. Es sind bei weitem nicht alle Nutzer technikaffin und daran interessiert, stets auf dem aktuellsten Stand der Entwicklung zu sein. Durch serverseitige Feature und Browser Detection ist es möglich, zukunftsfreundliche Webseiten zu erschaffen, bei denen aber auch veraltete Browser unterstützt werden können (Smashing Magazine, 2012, S. 220).

4.3.4 Fazit

Nach Abschluss der Entwicklung der RESS-Variante der BaRESS Webseite und nach Umsetzen der vorgestellten Techniken, lädt die RESS-Variante der Webseite nur noch einen Bruchteil der Daten der Standard-Variante. Die folgende Tabelle gibt einen Überblick über die Datenmenge und Request-Anzahl der drei unterschiedlichen Varianten beim ersten Aufruf durch ein Gerät:

BaRESS Variante	Standard	Optimiert	RESS Laptop	RESS iPad	RESS iPhone
Anzahl Requests	21	13	14	10	10
Gesamtgröße	5 MB	863 kB	642 kB	547 kB	359 kB

Tabelle 4.4: Vergleich BaRESS Varianten (Quellen Siehe Abbildung 6.1 bis Abbildung 6.5)

Das RESS-Konzept ist ein sehr spannender Ansatz zur Optimierung von Webseiten für mobile Geräte. Allerdings ist die Idee noch relativ jung und befindet sich in den Anfangsstadien der Entwicklung. Bei einem Einsatz der Techniken muss dem Webentwickler bewusst sein, dass er

nicht ohne Weiteres auf bestehende Techniken zurückgreifen kann, sondern eventuell selbst Entwicklungsarbeit leisten muss. Außerdem bringt die hohe Optimierbarkeit nicht nur positive Aspekte mit sich. Ein RESS-System fordert viel Pflege, da es zum einen auf aktuelle Technologien setzt, die sich wahrscheinlich noch stark weiterentwickeln werden, zum anderen immer die neusten Browser und Geräte unterstützen sollte. Da beides sehr schnelllebig ist, muss ein solches System unter Umständen oft aktualisiert werden.

RESS ist ein Konzept, das genauso wie RWD keine Wunderwaffe in der Entwicklung für mobile Geräte ist. Zwar bietet RESS eine große Bandbreite an neuen Techniken zur Optimierung und das System kann sehr vielfältig eingesetzt werden, jedoch kann der Testaufwand eines solchen Systems enorm groß werden, je nachdem, wie viele geräteabhängige Funktionen umgesetzt werden sollen. Der Einsatz eines RESS-Systems ist also eine große Chance für Projekte, jedoch sollte vor der Umsetzung eine Kosten-Nutzen-Analyse erfolgen.

5 Evaluation

Performance ist ein wichtiges und herausforderndes Thema in der Webentwicklung. Um eine möglichst große Bandbreite unterschiedlicher Geräte mit der gleichen URL zu versorgen und auf allen Geräten gleichzeitig eine bestmögliche User Experience zu ermöglichen, muss eine Webseite sehr gut geplant sein.

Für diese Bachelorarbeit war es besonders herausfordernd, im weiten Feld der Performance-Optimierung eine Beschränkung auf Themen zu realisieren, die mir aktuell am sinnvollsten erschienen. Gerade die Schnelllebigkeit sowie der OpenSource-Charakter der Webentwicklung machte es schwierig, sich auf bestimmte Themen zu beschränken. Mittlerweile hätte es aktuellere Ansätze gegeben, die allerdings noch nicht ganz fehlerfrei einsetzbar sind. Hat man einen Artikel in einem Magazin zu einem aktuellen Thema zu Ende gelesen, stößt man direkt auf den nächsten Blog eines Entwicklers, der einen neuen innovativen Ansatz vorstellt und untersucht. Die Bereitschaft vieler Entwickler, ihr Wissen, ihre Erfahrungen und am besten ihren Code online der Community zur Verfügung zu stellen, war bei der Suche nach Techniken sehr wertvoll. Gleichzeitig war es aber auch sehr herausfordernd für mich, einen Schlussstrich zu ziehen.

Die vorgestellten Techniken auf dem Medieninformatik-Server der Hochschule ausprobieren zu können, war für diese Bachelorarbeit sehr wichtig. Das testweise Ausprobieren der Techniken an BaRESS-Webseite hat mir gezeigt, dass manche der Techniken ohne großen Aufwand umgesetzt werden können und andere einen hohen Zeitaufwand fordern. Sind auf dem Apache-Server alle benötigten Module installiert und aktiviert, sollte der Einsatz der meisten Optimierungstechniken aus Kapitel 4.2 ohne größere Probleme möglich sein. Diese bewährten Techniken der Webseitenoptimierung sollten am besten immer verwendet werden. Komprimierung, Konkatenierung, Minifizierung und Caching sind die Techniken mit dem geringsten Aufwand bei gleichzeitig hoher Einsparung von Daten und Requests.

Ob es sich für ein Projekt lohnt, ein RESS-System einzusetzen, ist schwer zu sagen. Dafür sind Inhalt und Zielgruppen des Projekts entscheidend. Genauso kommt es stark auf den Entwickler und seine Fähigkeiten an. Für FrontEnd-Developer oder Webdesigner, die normalerweise nicht viel mit serverseitigen Skriptsprachen arbeiten, könnte es durchaus sehr herausfordernd sein, sich in die Funktionsweise eines solchen Systems einzuarbeiten. Für einen Entwickler hingegen, der viel serverseitig entwickelt, ist es dafür umso leichter. Allgemein bietet ein RESS-System die höchste Optimierbarkeit, allerdings auch den höchsten Aufwand.

5.1 Fazit

Ein wichtiger Aspekt wurde mir auch erst während dem Arbeiten an der Bacheloarbeit bewusst: Soll eine Webseite oder App wirklich performant sein, sollte am besten alles selbst gecodet werden oder man sollte nur extern programmierte Skripte und Frameworks einsetzen, die nicht mehr als die benötigte Funktion enthalten oder auf diese gekürzt sind. Frameworks und Pluginlösungen, wie Bootstrap oder jQuery, sind großartige Hilfen für die schnelle Prototypenentwicklung oder für kleinere Projekte, bei denen hohe Performance keine große Rolle spielt. Für alles andere sind diese Hilfen zu überladen. Viele Funktionen werden gar nicht benötigt, aber trotzdem geladen. Soll eine Webseite also so schnell wie möglich zu laden und zu parsen sein, sollte diese am besten von Grund auf von Hand gecodet werden. Nur so hat der Webentwickler die volle Kontrolle darüber, was geladen wird und kann diese Vorgänge viel besser optimieren.

Ich finde das RESS-Konzept sehr spannend. Die Einsatzmöglichkeiten sind fast unbegrenzt und ich bin gespannt, was in den nächsten Jahren in diesem Bereich weiterhin passieren wird. Ob sich das Konzept allerdings auf Dauer durchsetzt, bleibt offen. Für kleinere Projekte lohnt es sich oft nicht den Aufwand zu betreiben, den ein RESS-System fordert. Bei großen Projekten ist der Einsatz jedoch interessant. Die Entwicklung einer Extension für Content-Management-Systeme könnte einen großen Vortrieb des Ansatzes bringen. Welche Funktionen und Techniken durch RESS noch möglich sind, konnte im zeitlichen Rahmen dieser Bachelorarbeit nicht allumfassend erforscht werden. Es sind weitere Arbeiten nötig, um erfassen zu können, was noch alles möglich wäre und was sich durch RESS noch alles verändern könnte.

5.2 Ausblick

Es hat sich gezeigt, dass die Performance im Web mit die wichtigste Rolle für die User Experience darstellt. Mit den Techniken der vorangehenden Kapitel wird versucht, auf Basis der bestehenden Übertragungsprotokolle mit bestimmten Workarounds Verbesserungen zu erzielen. Allerdings stellen diese Workarounds keine dauerhaften Lösungen dar, da die Kernproblematiken nicht behandelt werden, sondern nur darum herum gearbeitet wird. Was das Internet benötigt, ist die Überarbeitung der bisherigen Standards, Techniken und Protokolle, die Probleme hervorrufen.

Eines der aktuellen großen Performanceprobleme ist nicht die Übertragung von Daten selbst, sondern die Art und Weise, wie diese auf den Weg vom Server zum Client gebracht werden. Der bestehende HTTP/1.0 Standard fordert für die Übertragung von Daten eine bestimmte Abarbeitung der Requests. Bevor ein neuer Request bearbeitet werden kann, muss der gesamte vorangehende Request abgeschlossen sein. Für jeden Request muss eine neue Verbindung geöffnet werden. Das bedeutet, dass mit jeder Verbindungsherstellung viele Header-Bytes übertragen werden müssen, was den Traffic erhöht und den Download der Dateien verzögert.

Mit dem HTTP/1.1 Standard wurde die Möglichkeit geschaffen, gleichzeitig Requests anzunehmen. Der Server kann die Requests jedoch nur in dessen Reihenfolge abarbeiten. Außerdem wurde durch die HTTP Header Anweisung »Connection: keep-alive« eingeführt. Dadurch muss nicht für jede Anfrage eine neue Verbindung geöffnet werden, sondern ein einmaliger Verbindungsaufbau zum Server reicht aus und die Verbindung wird offen gehalten. Der Server kann trotzdem nur alle Anfragen in deren Reihenfolge abarbeiten. Das kann unter Umständen dazu führen, dass der Download einer nachfolgenden Datei blockiert wird. Wenn der Server beispielsweise erst eine dynamische Datei generieren muss, aber andere statische Inhalte schon fertig zur Verfügung stehen, werden diese trotzdem blockiert.

HTTP/2.0 stellt die Überarbeitung durch die HTTPbis-Gruppe, einer Working-Group der Internet Engineering Steering Group (IESG), dar. Die genaue Spezifikation soll im November 2014 der IESG als »Proposed Standard« vorgelegt werden (HTTPbis, 2013).

HTTP/2.0 soll HTTP/1.1 nicht ersetzen, sondern es stellt eine Alternative dar und ist abwärtskompatibel. Hauptziel des HTTP/2.0 ist allgemein die Reduzierung der Wartezeit beim Laden von Webseiten durch:

- Reduzierung der Latenz durch gleichzeitiges Senden kompletter Anfragen/Antworten
- Reduzierung des Protokoll-Overhead (Datenüberhang) durch effiziente Kompression der HTTP Header Felder
- Unterstützung der Priorisierung von Dateien
- Unterstützung von Server-Push-Funktionen ohne Refresh des Users (Grigorik, HPBN, 2013, S. 207)

HTTP/2.0 benötigt keine Anpassung des eigentlichen Codes einer Webseite oder deren Inhalt. Lediglich die bisher genutzten Workarounds, wie z.B. Domain Sharding, Image Sprites und Konkatenierung, wären überflüssig und könnten wieder entfernt werden. Domain Sharding würde sogar zu mehr ausbremsenden Overhead und neuen Verbindungen führen.

Der Bedarf für den neuen HTTP/2.0 Standard ist groß. Firmen, wie Google oder Microsoft, arbeiten sehr engagiert an dem Projekt mit und forschen parallel an Vorschlägen für den HTTP/2.0 Standard. Das Projekt »SPDY«[30] (gesprochen speedy) von Google wurde in Version 2 von der HTTPbis-Gruppe adaptiert und als Ausgangspunkt für weitere Entwicklungen von HTTP/2.0 verwendet (Grigorik, HPBN, 2013, S. 209).

Microsoft verfolgt mit dem Projekt »HTTP Speed+Mobility« (S+M) einen ähnlichen Ansatz wie SPDY. Der größte Unterschied der beiden Entwicklungen: Bei SPDY wird jede Verbindung verschlüsselt, was die Ressourcen eines Geräts beansprucht und somit Akku und Leistung verbraucht. Das wirkt sich gerade bei mobilen Geräten negativ aus. S+M verschlüsselt

[30] Vgl.: http://www.chromium.org/spdy (Stand: 25. März 2014)

nicht jede Verbindung und verbraucht so weniger Akku und Leistung. Außerdem spricht sich das Projekt gegen die Server-Push Funktion gerade bei mobilen Geräten aus, da dies unter Umständen zu einem unerwarteten Download von Daten führen könnte und so mobile Datenverträge unnötig belastet würden.

Mit der Entwicklung und Implementierung weiterer Features in das neue HTTP/2.0 könnten sich auch weitere Probleme, wie das Liefern von Bilder, optimieren lassen. Ilya Grigorik hat in seinem Blog eine Technik vorgeschlagen, die er »HTTP Client Hints« nennt und die er als Vorschlag der HTTPbis-Gruppe vorgelegt hat (Grigorik, IETF Internet-Drafts, 2013). HTTP Client Hints sieht vor, dass im HTTP Client Request Header ein neues Feld eingeführt werden soll. In diesem Feld teilt der Client dem Server mit, welche Pixelbreite, –höhe und –dichte sein Bildschirm hat. In anderen Worten: Der Client soll im HTTP Request dem Browser mitteilen, welche Inhalte er gerne hätte. So könnte der Browser die passenden Bilder liefern, ähnlich wie es mit AdaptiveImages (siehe Abschnitt 4.3.2) gelöst wurde, nur ohne einen zusätzlichen Cookie übertragen zu müssen. Der Ansatz könnte die nicht ganz zuverlässigen Methoden UA Sniffing und Feature Test in diesem Bereich ablösen. Eine solche Lösung würde andere Responsive Image Workarounds zur Lieferung geräteabhängiger Bilder überflüssig machen.

Die Entwicklung von HTTP/2.0 ist noch lange nicht abgeschlossen und verspricht spannende Ansätze, um die Ladezeit von Webseiten in Zukunft deutlich zu verkürzen und die Kommunikation zwischen Client und Server zu optimieren.

6 Anhang

6.1 Wegweiser zur DVD

Auf der beiligenden DVD befinden sich die folgenden Dateien:

- **baress.zip:** Gezippte BaRESS-Webseite
- **Bachelorarbeit-ThomasRehm.pdf:** Diese Bachelorarbeit

6.2 Lokale BaRESS-Installation

Um die BaRESS-Webseite lokal zu betreiben, wird vorausgesetzt, dass bereits ein lauffähiges System, bestehend aus Apache mit PHP, installiert ist.

Folgende Schritte sind zum lokalen Betrieb notwendig:

- Entpacken Sie das ZIP-Archiv **baress.zip**
- Kopieren Sie den enthaltenen Ordner **baress** in das Document Root Verzeichnis Ihres lokalen Apache-Servers
- Um das RESS-System nutzen zu können, registrieren Sie sich für die kostenlose WURFL-Cloud (https://scientiamobile.com/cloud/signup/free) und tragen Sie Ihren API-Key in der Datei `/baress/ress/_inc/wurfltest.php` in die Variable `$config->api_key` ein
- Rufen Sie im Browser das Verzeichnis zum lokalen Server und der BaRESS-Installation auf, beispielsweise `localhost/baress/standard/`

Alternativ zur DVD steht der gesamte DVD-Inhalt im folgenden GitHub Repository zur Verfügung:

github.com/thomasrehm/BaRESS

6.3 Abbildungen

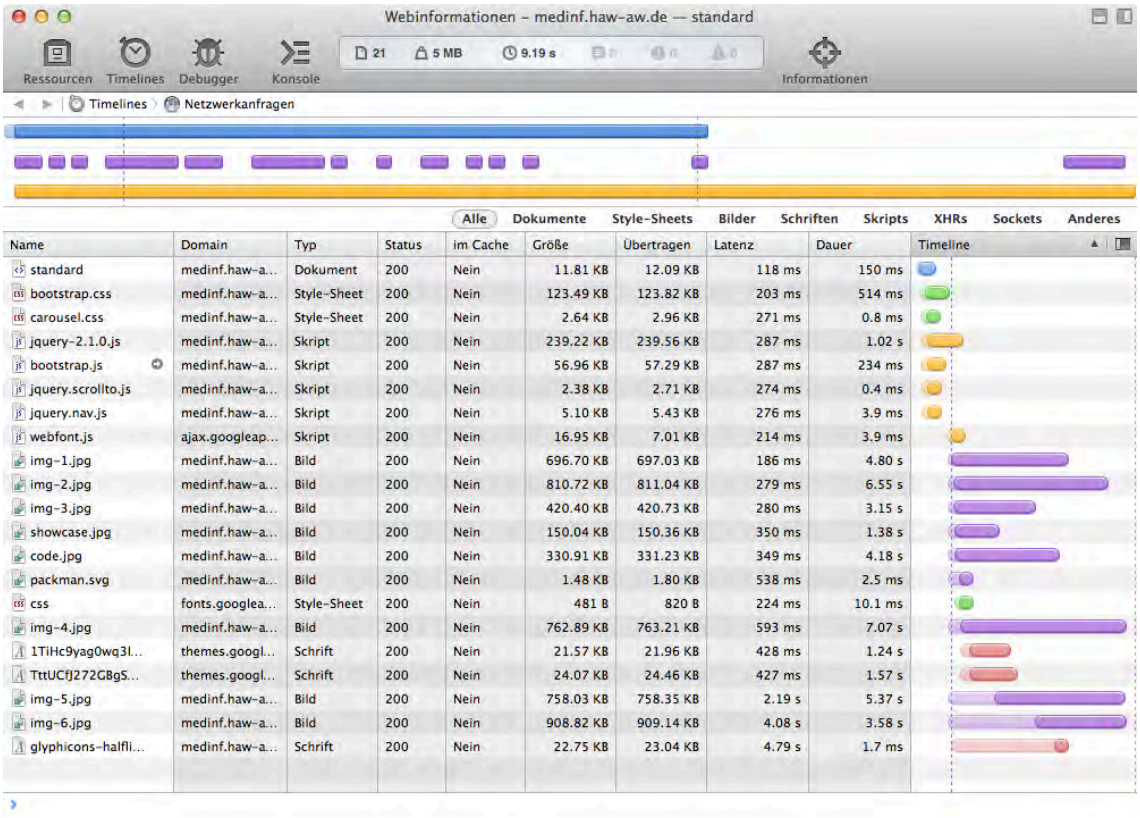

Abbildung 6.1: Screenshot WebDevTools Safari BaRESS Standard

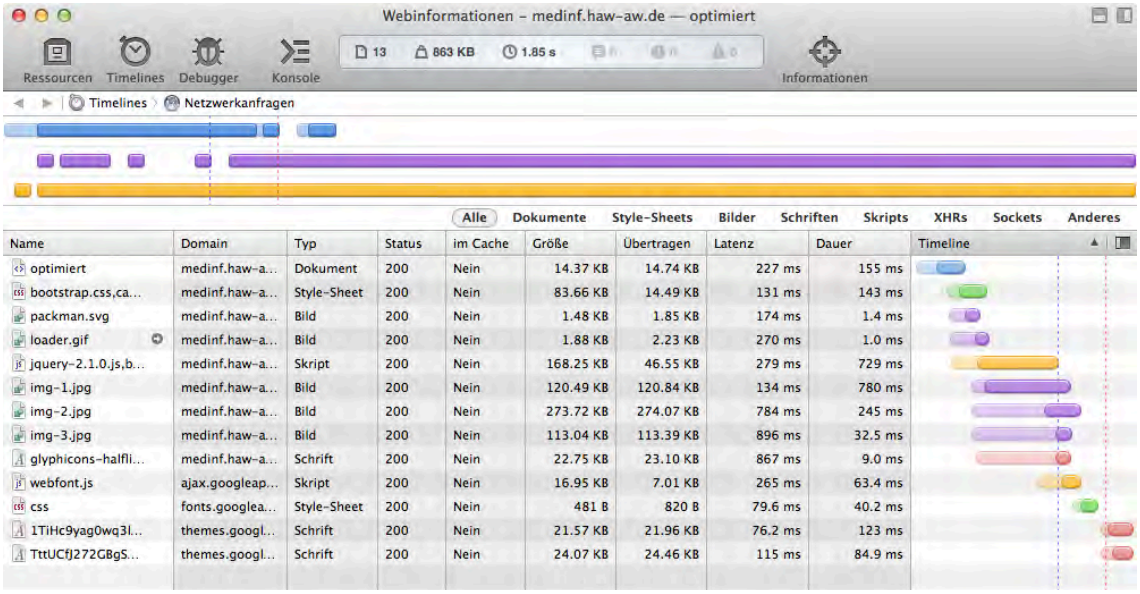

Abbildung 6.2: Screenshot WebDevTools Safari BaRESS Optimiert

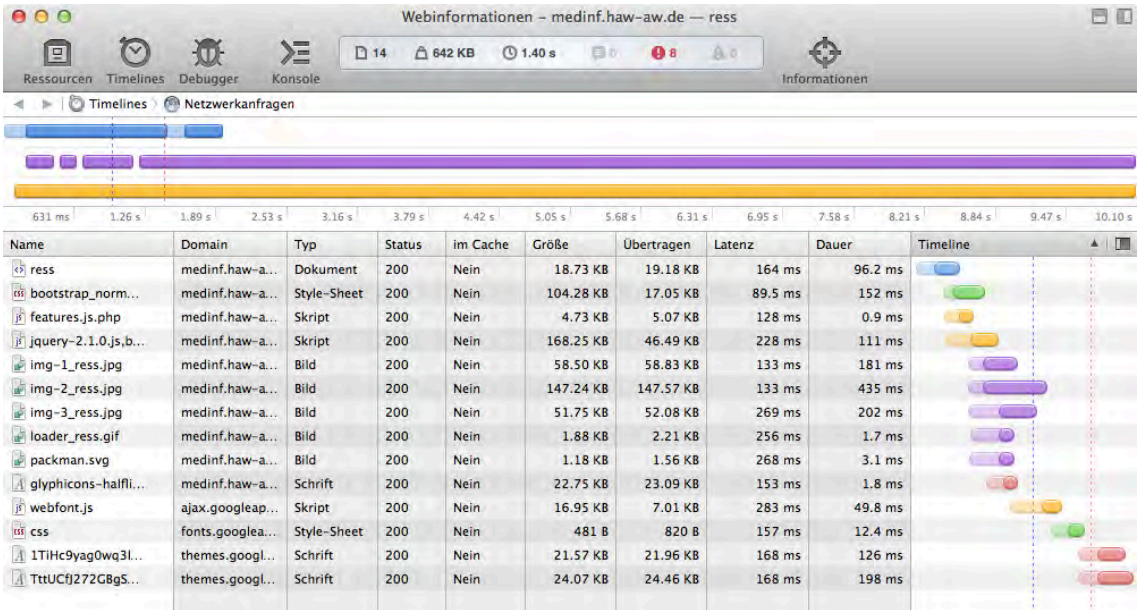

Abbildung 6.3: Screenshot WebDevTools Safari BaRESS RESS

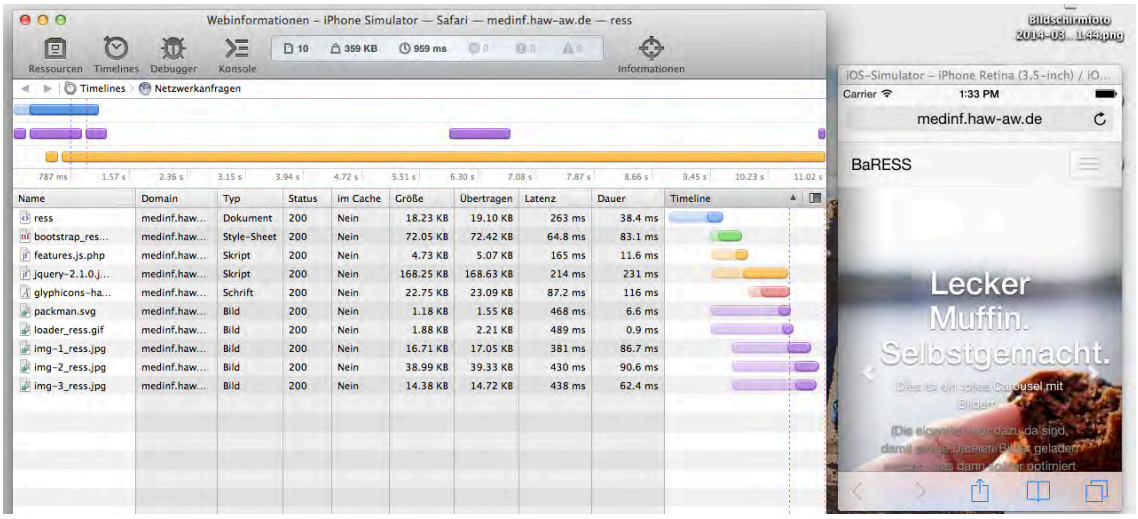

Abbildung 6.4: Screenshot WebDevTools Safari/iPhoneSimulator BaRESS RESS

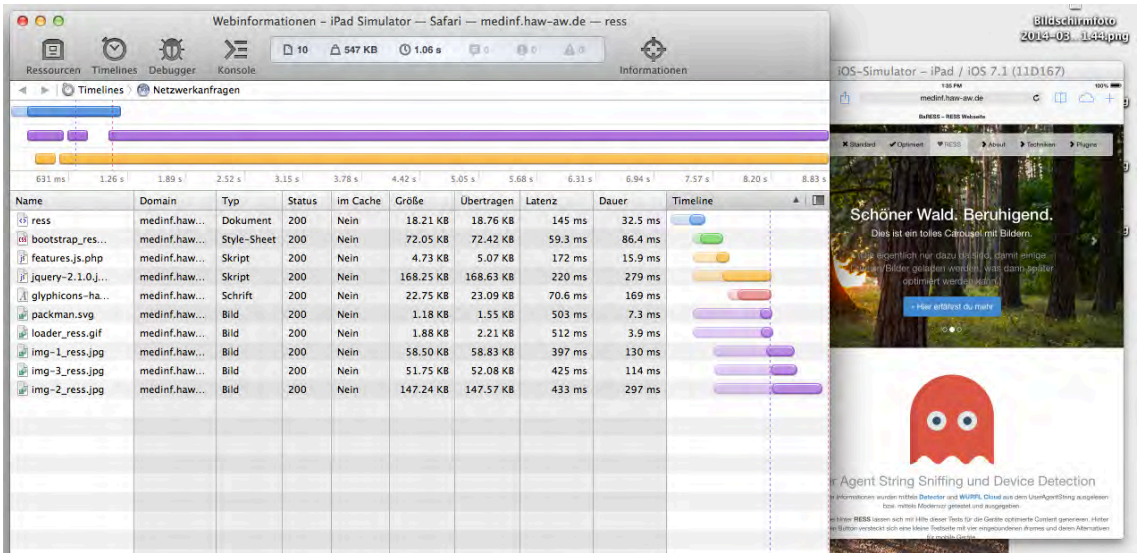

Abbildung 6.5: Screenshot WebDevTools Safari/iPadSimulator BaRESS RESS

6.4 Abbildungsverzeichnis

Soweit nicht anders angegeben, sind alle verwendeten Grafiken vom Autor selbst erstellt.

7 Literaturverzeichnis

Andersen, A. (8. September 2008). *webaim.org*. (webaim.org, Herausgeber) Abgerufen am 28. März 2014 von webaim.org - History of the browser user-agent string: http://webaim.org/blog/user-agent-string-history/

Apple. (9. Januar 2007). *Apple Presse Archiv*. Abgerufen am 20. November 2013 von http://www.apple.com/de/pr/library/2007/01/09Apple-Reinvents-the-Phone-with-iPhone.html

Cisco Systems. (5. Februar 2014). *Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2013–2018.* Abgerufen am 24. Februar 2014 von Cisco Systems: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white_paper_c11-520862.html

DigiWorld by IDATE. (2012). *DigiWorld Yearbook 2012.* (D. Publishing, Hrsg.) Montpellier: IDATE.

Google Developers. (11. Februar 2014). *Webmasters & Google Developers*. Abgerufen am 28. März 2014 von Redirects and User-Agent Detection - Webmasters & Google Developers: https://developers.google.com/webmasters/smartphone-sites/redirects?hl=en

Grigorik, I. (2013). *High Performance Browser Networking.* (I. O'Reilly Media, Hrsg.) Sebastopol: O'Reilly Media, Inc.

Grigorik, I. (6. Dezember 2013). *IETF Internet-Drafts.* Abgerufen am 25. März 2014 von IETF Internet-Drafts - HTTP Client Hints: http://tools.ietf.org/html/draft-grigorik-http-client-hints-01

Grigorik, I. (19. Juli 2012). *igvita.com | Ilya Grigorik*. Abgerufen am 25. März 2014 von igvita.com - Latency: The New Web Performance Bottleneck: http://www.igvita.com/2012/07/19/latency-the-new-web-performance-bottleneck/

GSM Association. (kein Datum). *GSM Association - About US*. Abgerufen am 28. März 2014 von GSM Association - About US - History: http://www.gsma.com/aboutus/history

HTTPbis. (21. Mai 2013). *Datatracker IETF*. Abgerufen am 25. März 2014 von Hypertext Transfer Protocol Bis (httpbis) - Charter: https://datatracker.ietf.org/wg/httpbis/charter/

iProspect. (29. Oktober 2012). *2012 Mobile Landscape: Western Europe.* Abgerufen am 28. Februar 2014 von iProspect GmbH: http://www.iprospect.se/wp-content/uploads/The-Mobile-Landscape.-Western-Europe.pdf

Kadlec, T. (2013). *Implementing Responsive Design - Building sites for an anywhere, everywhere web.* Berkeley, CA: New Riders.

Krenz-Kurowska, J., & Kurowski, J. (2013). *RESS Essentials.* Birmingham: Packt Publishing.

Marcotte, E. (25. Mai 2010). *A List Apart - Responsive Webdesign*. Abgerufen am 21. November 2013 von A List Apart: http://alistapart.com/article/responsive-web-design

Marcotte, E. (2011). *Responsive Web Design.* New York: A Book Apart.

Nielsen, J. (21. Juni 2010). *Website Response Times - Nielsen Norman Group*. Abgerufen am 28. Februar 2014 von Nielsen Norman Group: http://www.nngroup.com/articles/website-response-times/

o.A. (Wikipedia). (7. März 2014). *Wikipedia*. Abgerufen am 28. März 2014 von Minification (programming) - Wikipedia, the free encyclopedia: http://en.wikipedia.org/wiki/Minification_(programming)

o.A. (-. - -). *Wikipedia*. Abgerufen am 25. März 2014 von Wikipedia - HTTP 2.0: http://en.wikipedia.org/wiki/Http_2.0

Pickering, H. (24. Januar 2012). *Webdesignerdepot*. Abgerufen am 28. März 2014 von Webdesignerdepot - How to make your own icon webfont: http://www.webdesignerdepot.com/2012/01/how-to-make-your-own-icon-webfont/

Singhal, A., & Cutts, M. (9. April 2010). *Using site speed in web search ranking - Google Webmaster Central Blog*. (Google, Herausgeber) Abgerufen am 28. Februar 2014 von Google Webmaster Central Blog: http://googlewebmastercentral.blogspot.de/2010/04/using-site-speed-in-web-search-ranking.html

Smashing Magazine. (2012). *The Mobile Book.* (P.-P. Koch, S. Rieger, T. Walton, B. Frost, D. Olsen, D. Kardys, et al., Hrsg.) Freiburg: Smashing Media GmbH.

Souders, S. (18. September 2007). *Steve Souders*. Abgerufen am 24. März 2014 von High Performance Web Sites - 14 Rules for Faster-Loading Web Sites: http://stevesouders.com/hpws/rules.php

Souders, S. (12. Mai 2009). *stevesouders.com*. Abgerufen am 28. März 2014 von stevesouders.com - Sharding Dominant Domains | High Performance Web Sites: http://www.stevesouders.com/blog/2009/05/12/sharding-dominant-domains/

Souders, S. (23. Juli 2007). *Yahoo! Developer Network*. Abgerufen am 1. April 2014 von YDN Blog - High Performance Web Sites: Rule 11 Avoid Redirects: http://developer.yahoo.com/blogs/ydn/high-performance-sites-rule-11-avoid-redirects-7209.html

Statista. (26. März 2013). *Mobiler Traffic wächst mit beispielloser Geschwindigkeit - Statista*. (M. Brandt, Herausgeber) Abgerufen am 25. Februar 2014 von Statista: http://de.statista.com/infografik/1010/wachstum-mobiler-internet-traffic/

Walker Sands. (5. November 2013). *Walker Sands Mobile Traffic Report Q3 2013*. Abgerufen am 25. Februar 2014 von Walker Sands digital: http://www.walkersandsdigital.com/Walker-Sands-Mobile-Traffic-Report-Q3-2013

Wenz, C., & Hauser, T. (2013). *Websites optimieren.* München: Addison-Wesley Verlag.

Wroblewski, L. (2011). *Mobile First.* New York: A Book Apart.

Wroblewski, L. (12. September 2011). *RESS: Responsive Design + Server Side Components*. Abgerufen am 21. November 2013 von Luke Wroblewski: http://www.lukew.com/ff/entry.asp?1392

Zillgens, C. (2013). *Responsive Webdesign - Reaktionsfähige Websites gestalten und umsetzen.* München: Carl Hanser Verlag.