Ausgabe 5.13 August/September www.phpmagazin.de S&S

PHPmagazin

PHPmagazin

Deutschland 9,80 € Österreich 10,80 € | Schweiz 19,20 sFr
Niederlande 11,25 € | Luxemburg 11,25 €

PHP • JavaScript • Open Web Technologies

RabbitMQ
Geheimnisse des asynchronen Nachrichtenversands

Single-Page-Apps
Die Konzepte dahinter

PHP-Security
Die Evolution seit PHP 4.x

Legacy-Anwendungen
Sicherer Umgang mit Altlasten

©istockphoto.com/iatsun

Chrome Developer Tools
Werkzeugkasten für WWW-Arbeiter

Magento Testsuites
Unit Testing wie die Profis

4 195649 609809 05

Webentwicklung mit den Chrome Developer Tools

Tools for Workers

Seit September 2008 bereichert der von Google entwickelte Webbrowser Chrome die Browserlandschaft. Seit Februar 2013 ist es der am meisten genutzte Browser weltweit [1]. Wieder mal eine Erfolgsgeschichte aus „sunny California“ – und das zu Recht. Denn nicht nur ist er einer der schnellsten Browser, er bietet dem Entwickler mit den Chrome Developer Tools auch einen hervorragend ausgestatteten Werkzeugkasten, der kaum Wünsche offen lässt. Einen Teil der Werkzeuge werde ich in diesem Artikel vorstellen. Also Blaumann anziehen und Helm auf …

von Andreas Wenk

Google ist **die** Internetcompany. Wahrscheinlich gibt es keine andere Firma, die innerhalb von vierzehn Jahren mit, für und durch das Internet so erfolgreich geworden ist. Angefangen mit einem langweiligen *input field* als Suchmaschine für das Internet, hinter dem sich allerdings ein innovativer Algorithmus verbarg, ist Google heute ein Gigant. Und bei aller Kritik aus meiner Sicht immer noch „not evil“. Weitaus wichtiger ist aber die Tatsache, dass Google uns innovative Produkte beschert, die zumindest im Bereich Software fast durch die Bank Open Source sind. Dass für Services dann eine Gebühr bezahlt werden muss, ist dabei, meiner Meinung nach, völlig legitim. Aus der Schmiede purzeln Google-Produkte wie Google Analytics, Google App Engine, G+, Google Chrome OS, neue Sprachen wie GO und Dart und eben auch der Webbrowser Google Chrome. Aber ist der nun so viel besser als andere Browser? Ja und nein. Nein, weil Mozilla Firefox und Opera ebenso hervorragende Produkte am Markt platziert haben. Ja, weil Chrome einige Features bietet, die aus meiner Sicht einfacher zu bedienen sind und somit die Nutzung deutlich verbessern: z. B. die Handhabung von Extensions, die Konfiguration oder eben die Developer Tools. Letztere werden frei Haus mitgeliefert.

Nicht zu vergessen ist auch das Etablieren eines neuartigen Releasezyklus. Release early and often. Versionsnummern spielen dabei nur noch eine untergeordnete Rolle. Wissen Sie aus dem Stegreif, welche Version Ihres Firefox- oder Chrome-Browsers Sie gerade nutzen?

Developer-Tools-Tabs

Bevor ich im weiteren Verlauf des Artikels einige Tools genauer beschreiben werde, soll hier ein Überblick über die einzelnen Tabs der Developer Tools gegeben werden. Aber vorab die Frage: Wie öffnet man denn die Developer Tools? Der einfachste Weg ist der Shortcut. Je nach Betriebssystem ist dieser unterschiedlich. Auf einem Mac OS X ist dies ALT+CMD+J. Alternativ kann das Menü VIEW | DEVELOPER | DEVELOPER TOOLS genutzt werden (**Abb. 1**) oder das Aufklappmenü (**Abb. 2**) rechts neben der Adresszeile (drei waagerechte Striche): TOOLS | DEVELOPER TOOLS.

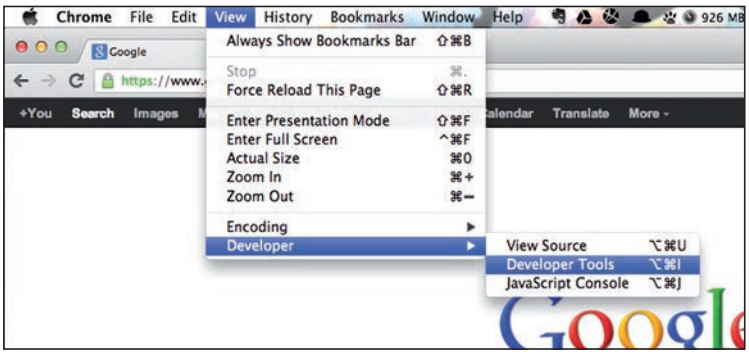

Abb. 1: „View"-Menü

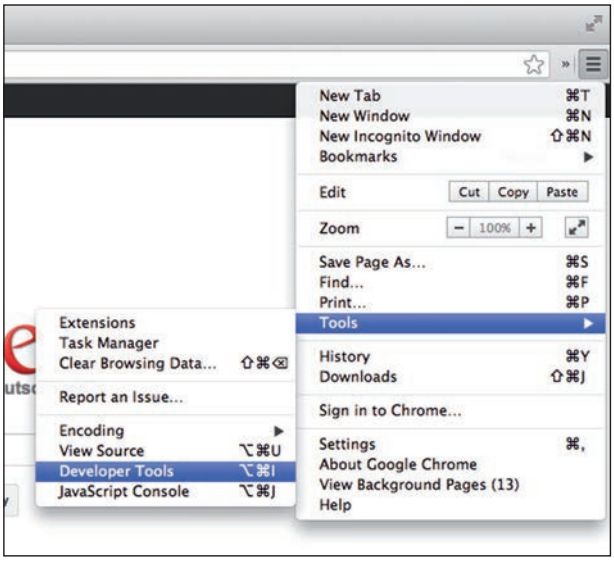

Abb. 2: Aufklappmenü

Die einzelnen Tabs im Überblick

Elements

Elements beinhaltet im linken Bereich die Repräsentation des DOM und auf der rechten Seite den Bereich mit allen CSS-Angaben zur aktuell geöffneten Webseite. In beiden Bereichen können Änderungen vorgenommen werden, die sich dann auch sofort auf die Darstellung der Webseite im Browserfenster auswirken.

Resources

Hier sind alle genutzten Ressourcen zu finden. Dazu gehören zum einen alle auf der Webseite verwendeten Dateien und Bilder, HTML5-Datenbanken, Cookies, Sessions und ebenso der Cache.

Network

Dieser Tab ermöglicht das Laden der einzelnen Ressourcen und Requests zu externen Ressourcen zu inspizieren. Dabei kann im linken Bereich *Name/Path* eine Ressource ausgewählt und auf der rechten Seite Informationen zu Headers, eine Preview und die Response des Requests eingesehen werden.

Sources

Alle auf der Webseite genutzten und geladenen JavaScript- und CSS-Ressourcen sind hier zugänglich. Hier ist der richtige Ort, um CSS- und JavaScript-Dateien zu editieren (ja, richtig!) und um den JavaScript Debugger zu nutzen. Auf der linken Seite wird eine Übersicht aller Ressourcen angezeigt, in der Mitte wiederum über Tabs angeordnet die Inhalte der einzelnen Ressourcen und auf der rechten Seite werden die Debuggertools angezeigt.

Timeline

Die Timeline bietet die Möglichkeit, das Laden der einzelnen Ressourcen und Elemente der Website in ihrem zeitlichen Ablauf zu recorden und danach zu untersuchen. Dabei wird zwischen Events – Aktionen wie Mausklick, Frames –, rendern der Anzeige in Frames per Second (FPS) und dem aktuellen Speicherverbrauch unterschieden.

Profiles

Der Tab Profiles dient ebenso wie der Tab Timeline der Untersuchung einiger Gegebenheiten während des Ladens der Webseite. Es können drei Profile erstellt werden: JavaScript CPU Profile zeigt die Nutzung der CPU beim Ausführen von JavaScript-Code. CSS Selector Profile erstellt eine Auswertung bzgl. der Performance der CSS-Selektoren. Und der Heap Snapshot zeigt die Verteilung des genutzten Memorys in Bezug auf JavaScript-Objekte und deren korrespondierende DOM-Elemente.

Audits

Auch dieser Tab reiht sich in die Profiling-Tools ein. Hier werden Sie nach Ausführung eines Audits auf Missstände bzgl. „Best Practices" im Umgang mit CSS, Bildern, Ladeverhalten von Ressourcen usw. aufmerksam gemacht. YSlow und PageSpeed lassen grüßen.

Console

Wenn Sie die Developer Tools noch nicht in der Tiefe genutzt haben, die Konsole haben Sie mit Sicherheit schon genutzt. Die Ergebnisse von JavaScript-*printf*-Debug-Statements, Fehler beim Laden oder Rendern der Web-

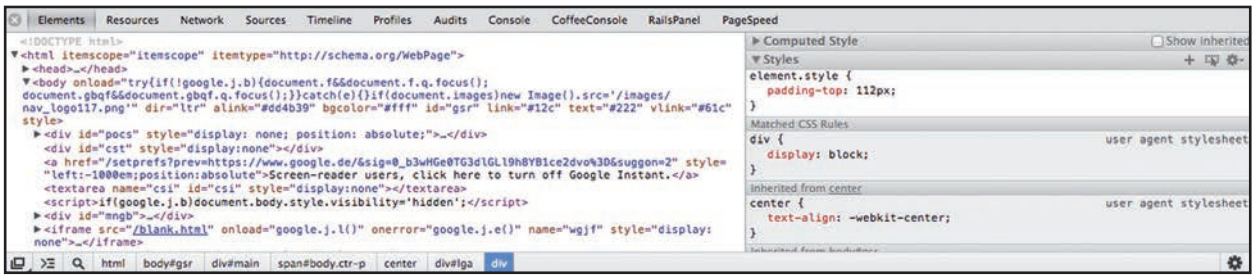

Abb. 3: Developer Tools

seite und viele weitere Ausgaben sind hier zu finden. Aber noch weiter: Sie können hier auch direkt JavaScript-Code ausführen. Alle Tabs sehen Sie auch in **Abbildung 3**.

Eine Website während der Entwicklung verändern

In den folgenden Abschnitten werde ich nun tiefer in zwei Bereiche der Developer Tools einsteigen. Zum einen in den Bereich Elements, um den DOM und die CSS-Style-Angaben der aktuell aufgerufenen Website on the fly zu verändern. Danach werde ich dann zeigen, wie im Bereich Sources Änderungen aufgezeichnet und wieder im Projekt gespeichert werden können und wie der JavaScript Debugger genutzt wird.

Beim Bauen einer Webseite kommt zwangsläufig die Umsetzung des Designs auf den Entwickler zu. Sprich die Design- oder die Creation-Abteilung hat ein Layout erstellt, das nun in HTML und CSS gegossen werden muss. In modernen Webapplikationen wird dabei ohne Zweifel eine Trennung zwischen Applikationscode und Frontend vorgenommen, wobei es meistens ein oder mehrere Grundlayoutdateien gibt, in die dann dynamisch die Views der einzelnen Bereiche der Website integriert werden. Sicherlich wird zuerst im Editor der Wahl die Struktur der Seiten angelegt und das HTML und CSS umgesetzt. Beim Feintuning kommt es dann allerdings auf jedes Pixel an. Ganz zu schweigen vom Integrieren von CSS3-Eigenschaften, wie Transitions, Schatten, runde Ecken („wenn's schee macht") und weiteren coolen Eigenschaften.

Um die Arbeitsschritte zu erleichtern, sind die Developer Tools das Werkzeug der Wahl. Zum ausprobieren können Sie jede beliebige Website in Goggle Chrome aufrufen und die Developer Tools einblenden. Hier im Artikel folgen Sie den Beispielen am besten, wenn Sie die Beispielwebsite vom Git Repository unter [2] herunterladen. Sie brauchen keinen Webserver, sondern müssen den Browser nur mit dem Parameter, wie im Kasten „Beispielwebseite lokal aufrufen" beschrieben, starten.

Nachdem Sie die Beispielwebseite und darin die Developer Tools geöffnet haben, klicken Sie auf den Tab Elements. Im zweigeteilten Bereich des Tabs sehen Sie links das HTML-Markup und können es nun *inspecten*. Fahren Sie mit der Maus über die einzelnen Tags im Markup und verfolgen Sie im Browserfenster den gerade gewählten Bereich – dargestellt durch einen farblich wechseln-

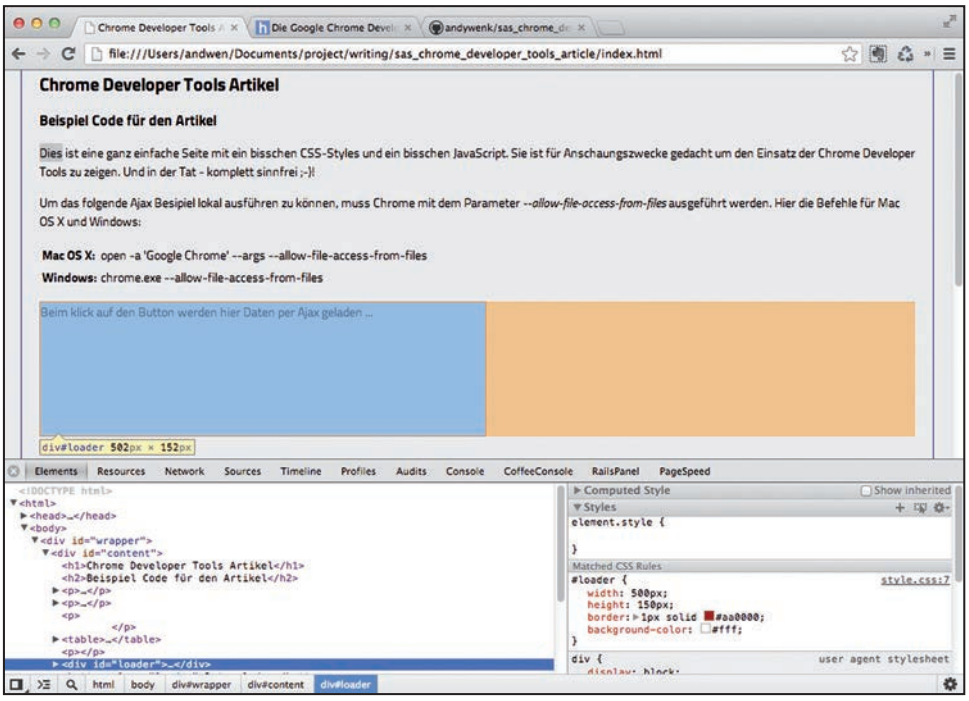

Abb. 4: Der Inspector bei der Arbeit

Beispielwebseite lokal aufrufen

Wenn Sie die Beispielwebseite des unter [2] angegebenen Git Repositorys nutzen möchten, müssen Sie dies zuerst klonen:

```
git clone git://github.com/andywenk/sas_chrome_developer_tools_article.git
```

Wechseln Sie danach in das Verzeichnis *sas_chrome_developer_tools_article* und starten Sie Google Chrome folgendermaßen:

Betriebssystem	Startbefehl
Mac OS X	open -a 'Google Chrome' –args –allow-file-access-from-files
Windows	chrome.exe –allow-file-access-from-files

Dies ist notwendig, da sonst aufgrund der Same Origin Policy das AJAX-Beispiel nicht funktionieren würde.

den Hintergrund. Dabei ist Hellblau der Bereich der Inner-Box und Orange der Bereich Outer-Box, der Abstände zu den umliegenden Elementen darstellt. Markieren Sie nun einmal das Element * ... *, und verfolgen Sie dann auf der rechten Seite die geänderte Ansicht im CSS Style Inspector. **Abbildung 4** veranschaulicht das Inspecten des Loaders.

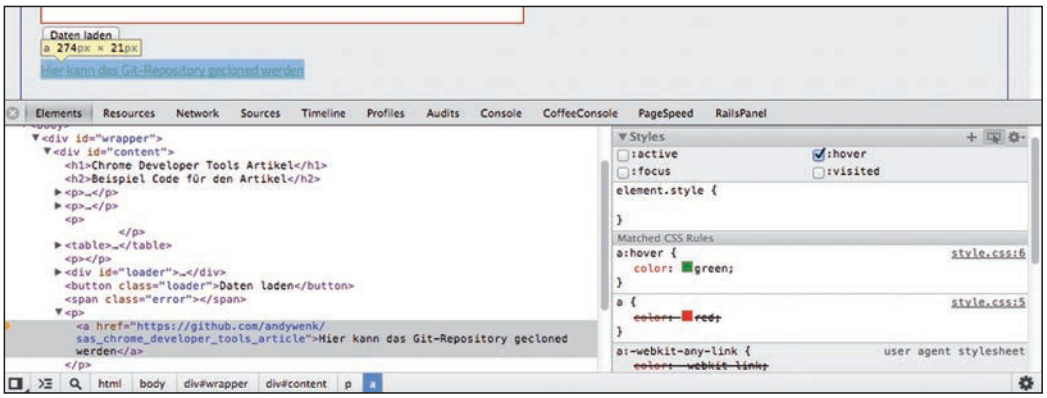

Abb. 5: Pseudo-CSS-Styles

Soweit ist dies schon mal eine sehr gute Möglichkeit, die einzelnen Elemente der Website zu untersuchen. Aber noch besser: Sie haben auch die Möglichkeit, die CSS Styles und das HTML-Markup zu verändern.

Die CSS Styles lassen sich sehr einfach auf der rechten Seite verändern. Zum einen können Sie die Werte des vorhandenen Styles ändern, indem Sie den jeweiligen Wert der Style-Eigenschaft ändern oder indem Sie im „leeren“ Bereich *element.style {}* eigene Style-Eigenschaften eintragen. Beide Änderungen wirken sich dann direkt auf das ausgewählte Element aus. Probieren Sie diese Varianten am besten selbst aus. Beachten Sie dabei, dass Änderungen, die Sie unter *element.style {}* vornehmen als *style=" " "* -Angabe an das geänderte Element im HTML-Markup geschrieben werden.

Zustände der CSS-Pseudoklassen bearbeiten

Der Bereich Styles auf der rechten Seite des Elements-Tabs weist am rechten Rand drei Symbole auf. Das mittlere heißt *Toggle Element State*. Sie können dort auswählen, ob bei den Style-Eigenschaften auch der Eintrag für z. B. *:hover*-Eigenschaften eines Links angezeigt wird. In der Beispielwebsite können Sie einmal den Link unterhalb des Buttons Daten laden ansehen, wenn Sie vorher *:hover* angeklickt haben **(Abb. 5)**.

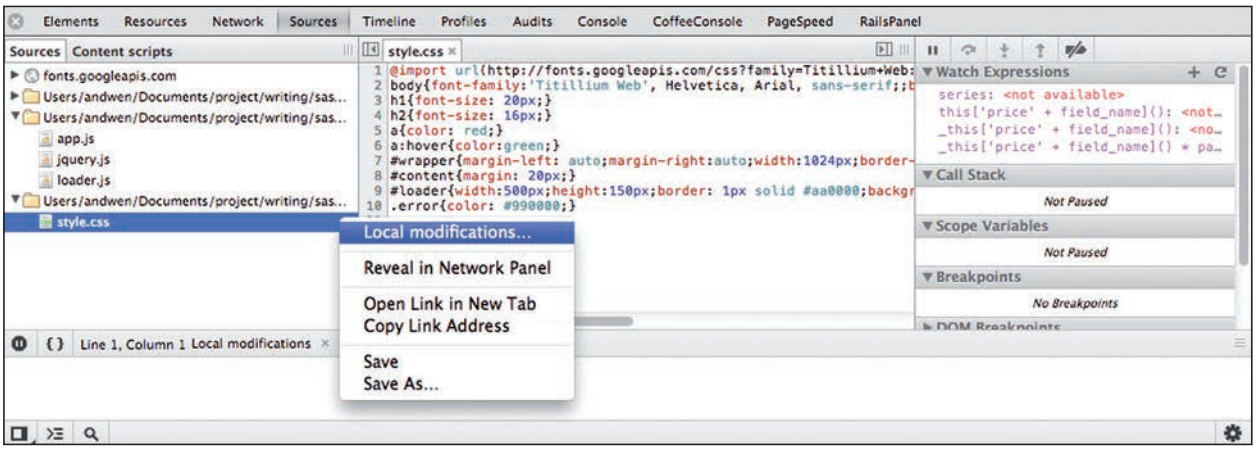

Abb. 6: Local Modifications

Ein sehr hilfreiches Tool ist der Bereich *Metrics* in der rechten Spalte unterhalb der Style-Eigenschaften. In diesem Bereich werden alle Abmessungen des gewählten Elements grafisch dargestellt. Natürlich können Sie die Werte dort ebenfalls ändern.

Kommen wir nun zur Änderung des HTML-Markups. Sie können entweder die Attribute eines Tags ändern oder den Inhalt einer gesamten DOM Node. Im ersten Fall doppelklicken Sie auf ein Tag und ändern dort die Eigenschaften und Attribute. Im zweiten Fall markieren Sie das sich öffnende Tag und wählen per rechten Mausklick im Kontextmenü den Eintrag Edit as HTML.

Beachten Sie, dass die Änderungen erst sichtbar werden, wenn Sie das Editieren beenden und den Bereich wieder schließen. Klicken Sie dazu einfach einmal außerhalb des Editierbereichs.

Die Änderungen, die Sie hier vorgenommen haben, sind nicht persistent, sondern zum Ausprobieren. Wie Sie Änderungen vornehmen und diese dann speichern können, verrate ich Ihnen im nächsten Abschnitt.

Tab Sources – CSS- und JavaScript-Dateien editieren und speichern

Eine weitere Möglichkeit, die zu einer Webseite gehörenden Dateien zu ändern, bietet der Tab Sources. Dieser Bereich ist per Default dreigeteilt: Links ist ein Dateimanager, in der Mitte ein Editor und rechts ein zum JavaScript Debugger gehörender Bereich, den wir für den Moment ignorieren. Im Dateimanager können die Dateien klassischerweise über eine Baumstruktur erreicht werden. Editierbar sind JavaScript- und CSS-, nicht aber HTML-Dateien.

Nun, das Editieren der Dateien ist ganz nett, aber wäre es nicht super, wenn ich zum einen die Änderungen verfolgen, einzelne *edits* wieder zurücknehmen und dann eine neue Version der bearbeiteten Datei in meinem Projekt speichern könnte? Ja das wäre es – und das geht. Als Aufgabe ändern wir zum einen die Hintergrundfarbe der Überschrift *h1* und außerdem die *:focus*-Eigenschaft des Links *a*.

Im ersten Schritt markieren wir im Dateimanager die Datei *style.css* im entsprechenden Ordner. Dann öffnen wir die Konsole, um die lokalen Änderungen verfolgen zu können. Dazu rechtsklicken wir die Datei *style.css* und wählen den Eintrag *Local modifications …* (**Abb. 6**).

Im zweiten Schritt nehmen wir dann die Änderungen nacheinander vor. Beachten Sie, dass die geöffnete Datei im Editor im mittleren Bereich oben als Tab erscheint. Ändern wir die Style-Angabe für *h1*, erscheint neben dem Dateinamen ein * als Zeichen, dass die Datei geändert wurde (**Abb. 7**). Die Änderung an sich ist ebenfalls sofort sichtbar.

Um die Änderungen nun verfolgen zu können, speichern Sie diese, indem Sie den Shortcut für Speichern ausführen (Mac OS X: cmd+s). Wiederholen wir dann den Vorgang auch für den Link und fügen ein Style für die Pseudoklasse *:focus* hinzu (unter Tab Elements | Element state „:focus“ aktivieren!) und speichern die Änderung, erhalten wir in der Konsole schließlich zwei Einträge wie in **Abbildung 7** zu sehen. Beachten Sie, dass die bearbeitete Datei natürlich noch nicht physikalisch gespeichert ist, sondern der Speichervorgang nur eine nicht persistente Revision erstellt, die dann in einem weiteren Schritt auch tatsächlich physikalisch gespeichert werden kann (**Abb. 8**).

Wie Sie sehen, steht in der Konsole neben dem Dateinamen *revert*, was uns ermöglicht, die Änderungen wieder rückgängig zu machen. Weiterhin sehen wir alle Revisionen des Editiervorgangs, wobei eine neue Revision jeweils beim Speichern erstellt wurde. Dabei erscheint die neueste Revision ganz oben. Klappen Sie eine Revision auf, sehen Sie zum einen die vorgenommene Änderung und einen Eintrag namens *apply revision content*. Bei einem Klick auf den Eintrag können Sie den Zustand des Editiervorgangs zu genau diesem Zeitpunkt wiederherstellen. Haben Sie z. B. zuerst die Überschrift geändert und dann den Link und klicken dann bei der Revision mit der Änderung der Überschrift auf *apply revision content*, wird die Änderung des Links wieder zurückgenommen. Sie können dies so oft vornehmen, wie Sie möchten. Für jede Änderung wird auch hier eine neue Revision erstellt.

Wenn Sie dann mit allen Änderungen zufrieden sind, können Sie die geänderte Version der Datei persistent

über den Rechtsklickdialog auf der Datei *style.css* per Auswahl des Eintrags Save as … speichern. Wow – dies ist aus meiner Sicht ein Killerfeature. Allerding ist zu beachten, dass die Developer Tools nur CSS und JavaScript anzeigen. Wenn Sie z. B. SCSS oder SASS nutzen, kommen Sie an diese Dateien nicht ran, sondern nur an eine konkatenierte CSS-Datei. Das Speichern dieser Datei nach dem Editieren in den Developer Tools macht eher weniger Sinn. Aber Sie können natürlich trotzdem die einzelnen Änderungen aus den Revisionen übertragen.

In diesem ersten großen Abschnitt haben Sie gesehen, wie die Chrome Developer Tools für die Anpassung des Layouts und der Styles einer Website hervorragend genutzt werden können. Im nächsten Abschnitt widmen wir uns nun dem debuggen von JavaScript.

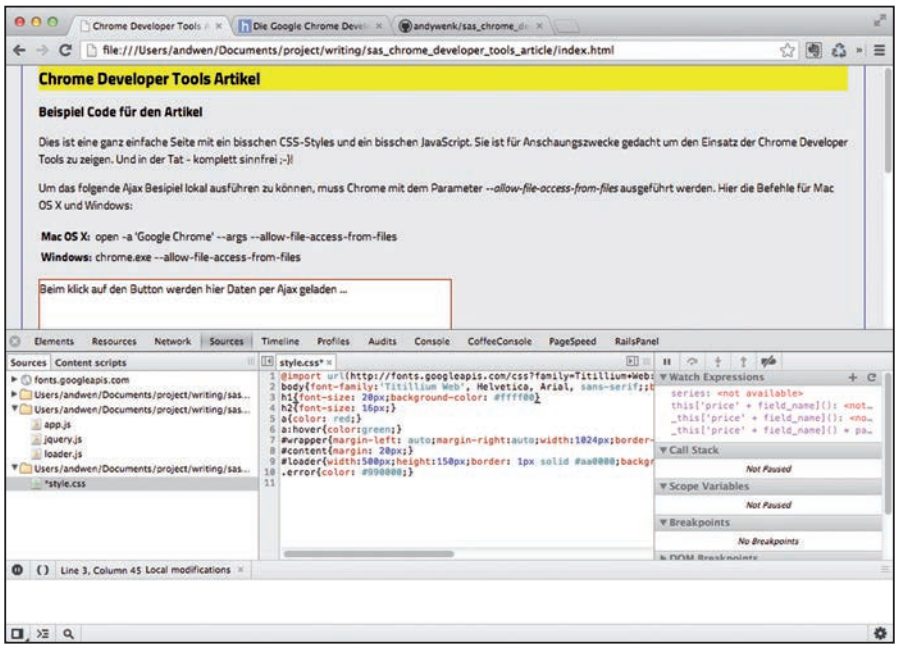

Abb. 7: CSS geändert

Der JavaScript Debugger

JavaScript ist die am meisten verwendete Programmiersprache der Welt. Wussten Sie das? Sie nutzen diese ja auch jeden Tag – zumindest, wenn Sie eine Webseite öffnen. In den letzten Jahren ist JavaScript auch spätestens seit der Vorstellung von Node.js durch seinen Erfinder Ryan Dahl auf der JSConf.eu 2009 [3] auf dem Server nicht mehr nur Spielkram, sondern wird millionenfach eingesetzt. Aber nicht zu vergessen sind auch die großartigen JavaScript Libraries und -Frameworks wie Yeoman, Meteor, AngularJS, Ember.js, Backbone.js, RequireJS, Underscore, Knockout, Three.js, npm, asm.js, Emscripten und wie sie da alle heißen. Mit JavaScript lassen sich mittlerweile unglaublich coole Dinge umsetzen und sie bringen einen riesigen Mehrwert für den Benutzer. Eines haben allerdings alle Libraries und Frameworks gemeinsam: Sie müssen debuggt werden. Im Frontend kann dies mit dem Debugger der Developer Tools geschehen. Sehen wir uns dazu ein sehr einfaches Beispiel an.

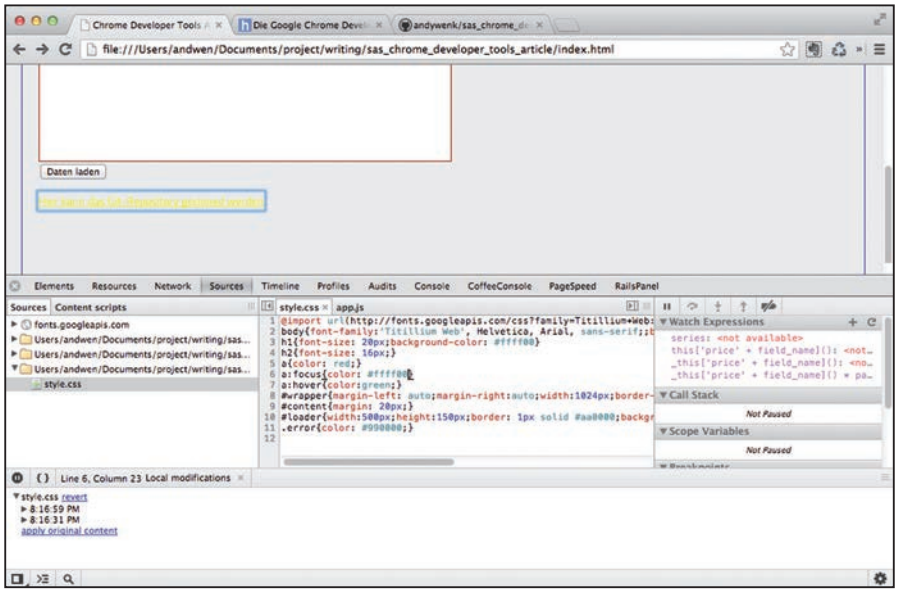

Abb. 8: Local Modifications gespeichert

Ein Click-Event debuggen

Die Beispielwebseite hat eine Funktionalität, die mit JavaScript umgesetzt ist. Sehen Sie sich dazu mal den Quellcode der Dateien *app.js* und *loader.js* im Ordner *javascripts* an. Hier der Use Case für die Funktionalität: Als Benutzer möchte ich bei einem Klick auf einen Button nach zwei Sekunden einen Text in einen Anzeigebereich laden, der mir Informationen zu den Chrome Developer Tools gibt. Auch wenn dieser Use Case frei von der Leber erfunden und trivial ist, erfordert die Implementierung schon ein wenig Überlegungen. Hier die Schritte, die getan werden müssen:

- eine Datenstruktur in einer Datei erstellen, die die Informationen enthält – hier JSON
- ein *div*-Element erstellen, in dem der Text nach dem Laden angezeigt wird
- einen Button erstellen, der das Laden der Informationen anstößt
- eine Funktionalität erstellen, die auf das Click-Event des Buttons reagiert und den Ladeprozess anstößt
- eine Funktionalität erstellen, die die Daten per AJAX lädt
- eine Funktionalität erstellen, die die Daten im *div*-Element ausgibt

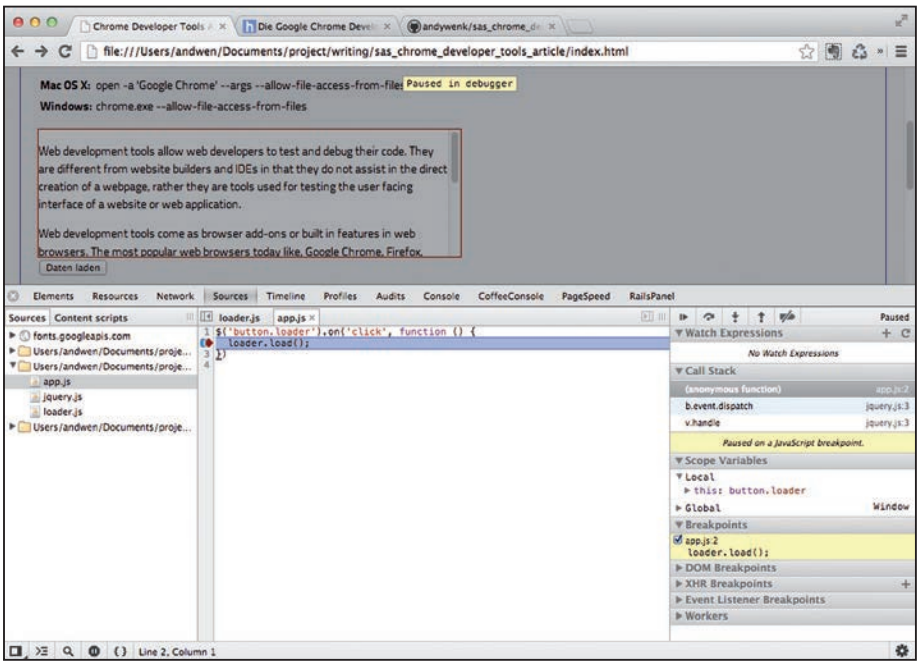

Abb. 9: Der Debugger hält in Zeile zwei

Abb. 10: Debuggersteuerung

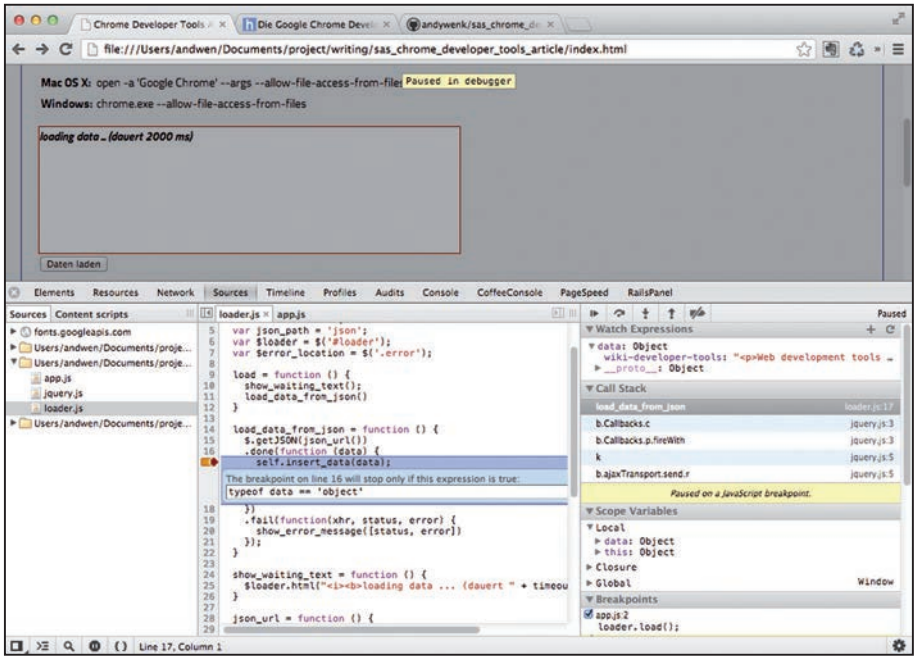

Abb. 11: Edit Breakpoint

- eine Funktionalität, die das Ausgeben der Daten im *div*-Element zwei Sekunden verzögert
- eine Funktionalität, die mögliche Fehler anzeigt

Diese Liste kann noch erweitert werden, soll für unsere Anschauungszwecke aber genügen.

Im ersten Schritt möchten wir nun die Funktionalität debuggen, die den Klick auf den Button verarbeitet. Diese ist in *app.js* zu finden. Ich verwende hier der Einfachheit halber jQuery [4]. Das Ziel beim Debuggen ist, die Ausführung des Programms an einer bestimmten Stelle anzuhalten und Variablen oder Zustände an dieser Stelle zu untersuchen. Um das Programm anzuhalten, wird ein *Breakpoint* gesetzt. Tun wir das und setzen einen Breakpoint in Zeile zwei des Skripts *app.js*. Klicken Sie dazu auf die Zahl 2 der Zeilennummerierung. Wenn wir nun den Button DATEN LADEN klicken, wird die Ausführung der Datei *app.js* genau in Zeile zwei angehalten: Im Editor der Datei *app.js* sehen Sie nun, markiert durch den dunkelroten Pfeil auf der Zeilennummerierung, dass der Debugger genau in Zeile zwei angehalten hat (**Abb. 9**). Sehr gut!

Auf der rechten Seite sehen Sie bei den Debugger Tools nun viele wichtige Informationen. Im Bereich Scope Variables können Sie Variablen, die an dieser Stelle zugänglich sind, untersuchen. Zum Beispiel *this*, was erwartungsgemäß den *button* repräsentiert. Breakpoints zeigt eine Liste aller momentan gesetzten Breakpoints. Der Bereich Call Stack zeigt, welche Methoden bis zu diesem Punkt ausgeführt wurden, wobei die Letzte die ist, in der der Breakpoint gesetzt wurde. Und schließlich können bei Watch Expressions eigene Ausdrücke angegeben werden, um Inhalte von Variablen zu sehen. Das ist z. B. beim Debuggen eines Loops sehr hilfreich und lässt das Untersuchen vieler Variablen gleichzeitig zu.

Wie geht es nun weiter? Wir sind an der Stelle, die wir untersuchen wollten, angelangt und möchten nun in der Ausführung des Skripts fortfahren. Um dies steuern zu können, befindet sich rechts oben im Bereich der Debugger Tools eine Leiste mit Symbolen (**Abb. 9**). Die Symbole bedeuten von links nach rechts:

- *Resume Script Execution* – Ausführung des Skripts wieder aufnehmen
- *Step over next Function call* – zum nächsten Funktionsaufruf springen
- *Step into next Function call* – in die nächste Funktion einspringen
- *Step out of current Function* – aus der aktuellen Funktion heraus- und zum nächsten Funktionsaufruf springen
- *Deactivate breakpoints* – alle Breakpoints daktivieren

Die Bezeichnungen sind ziemlich selbsterklärend. Wenn wir beispielsweise nur die Stelle, an der wir den Breakpoint gesetzt haben, untersuchen wollen und dann fortfahren möchten, klicken wir das erste Symbol (*Resume Script Execution*), und die Daten werden in das *div*-Element geladen.

Setzen wir nun zu Anschauungszwecken einen zweiten Breakpoint in der Datei *loader.js* in Zeile 17. Wenn es keinen Fehler gibt, war das Laden der JSON-Datei mit den Informationen für das *div*-Element erfolgreich. Wir wissen aber nicht mehr genau, wie der Key heißt, in dem die Daten stehen. Wenn wir nun auf den Button DATEN LADEN klicken, hält der Debugger zuerst in *app.js* in Zeile 2 an. Klicken wir auf den Schalter RESUME SCRIPT EXECUTION hält der Debugger wie gewünscht in *loader.js* in Zeile 17 an. Wenn wir nun noch in Watch Expressions mit dem +-Zeichen in das sich öffnende Feld *data* eintragen, können wir den Inhalt der Variable *data* ansehen und stellen fest, dass der Key *wiki-developer-tools* heißt. Genau für solche Aufgaben ist der Debugger gemacht.

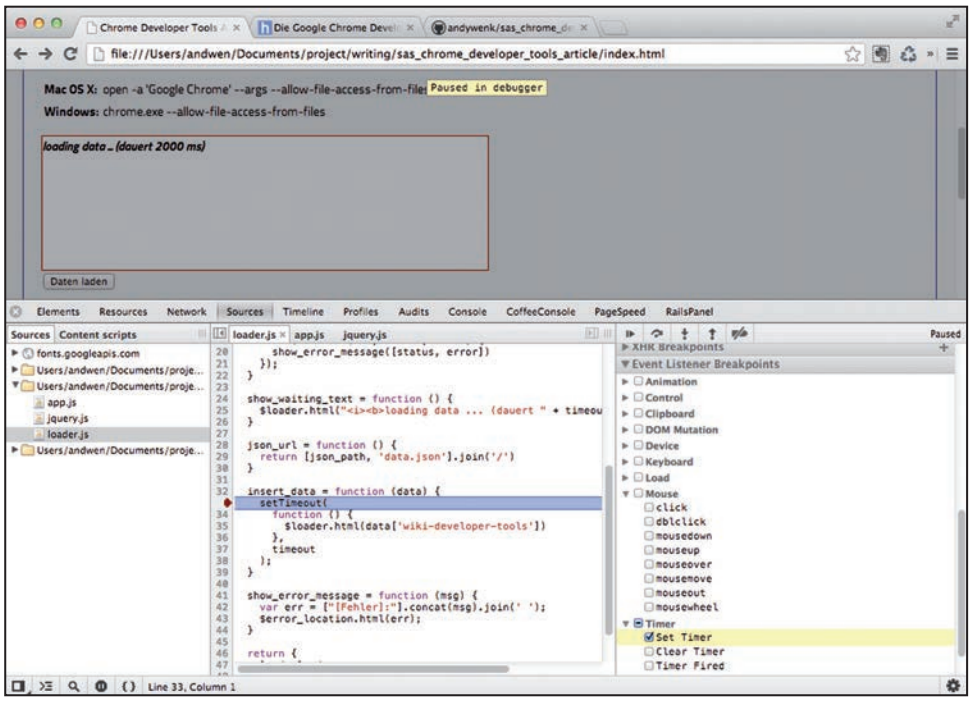

Abb. 12: Event Listener Breakpoints

Eine sehr schöne Möglichkeit, das „Anbeißen" eines Breakpoints zu verfeinern, ist das Setzen einer Bedingung. Klicken Sie dazu mit der rechten Maustaste auf den Breakpoint in Zeile 17. Es erscheint ein kleiner Dialog, in den Sie JavaScript-Code einfügen können (**Abb. 11**).

Ein sehr hilfreiches Tool, wenn ein großes Skript debuggt werden muss und es viele Schleifendurchläufe gibt, Sie aber den Code nur unter ganz bestimmten Voraussetzungen an der Stelle des gesetzten Breakpoints untersuchen wollen.

Hands on! Verwenden Sie nun zum Ausprobieren die anderen Debugger-Steuerung-Schalter, um die Ausführung des Skripts näher zu untersuchen. Sie werden sehen, dass z. B. bei Verwendung des Schalters *Step into next Function call* der Weg bis zum Anzeigen der Daten im *div*-Element sehr gut zu verfolgen ist.

Event Listener Breakpoints

Der Debugger verfügt über einen sehr hilfreichen Bereich namens *Event Listener Breakpoints* (**Abb. 12**). Hier können Sie bestimmen, dass der Debugger auf Grund von Events wie *focus*, *click* oder ähnlichen Events anhält. Setzen Sie z. B. einen Haken bei *Timer – Set Timer* und der Debugger hält exakt an der Stelle in der Datei *loader.js* an, bei der die *setTimeout()*-Methode genutzt wird.

The Big Picture ...

... ist das, was dieser Artikel bieten kann. Die Chrome Developer Tools sind cool, steigern die Produktivität und machen Spaß. Beim Schreiben des Artikels bin ich zu der Erkenntnis gekommen, dass ich nur einen kleinen Einblick in die Möglichkeiten der Google Chrome Developer Tools geben kann. So viele Themen sind es Wert, besprochen zu werden, und ebenso viele habe ich hier nicht besprochen. Ich möchte Sie ermutigen, die Tools einzusetzen und zu nutzen. Und außerdem ermutige ich Sie, den Entwicklern der Developer Tools auf den Social-Media-Kanälen zu folgen. Ein guter Start hierfür sind mit Sicherheit Addy Osmani [5] und Paul Irish [6].

Haben Sie bereits Erfahrung mit den Developer Tools gemacht? Welche Themen interessieren Sie brennend? Schreiben Sie mir doch eine E-Mail an andy@nms.de ... ich freue mich darauf!

Andreas Wenk ist Senior Developer bei der SinnerSchrader GmbH und programmiert HA-Webapplikationen mit Ruby on Rails. JavaScript ist neben seinen Mädelz und MetalPunkRock sein Steckenpferd ... Manchmal schreibt er unter http://blog.nms.de und G+ (http://goo.gl/i5ZEh).

Links & Literatur

[1] http://en.wikipedia.org/wiki/Usage_share_of_web_browsers
[2] https://github.com/andywenk/sas_chrome_developer_tools_article
[3] http://www.youtube.com/watch?v=EeYvFl7li9E
[4] http://jquery.com/
[5] https://plus.google.com/u/0/+AddyOsmani/posts
[6] https://plus.google.com/u/0/113127438179392830442/posts