Seminararbeit:
jQuery - Plugin Entwicklung

Frank Roth

31. Mai 2013

Betreuung: Prof. Gremminger

Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Inhaltsverzeichnis

1 Abstract

In Zeiten an denen webbasierte Applikationen für Desktop-PCs, Smartphones oder auch Tablets immer mehr an Bedeutung gewinnen, steigt des Verlangen von Web- aber auch App-Entwicklern nach ausgereiften und schnellen JavaScript-Frameworks. Einer der am weit verbreitetsten dieser Frameworks ist jQuery. In der folgenden Arbeit wird zu Beginn kurz auf die grundlegenden Webtechnologien HTML, CSS und JavaScript eingegangen, um später jQuery und die Plugin-Entwicklung innerhalb von jQuery zu erläutern. Dabei werden sowohl häufig gestellte Fragen, als auch Vorgehensweisen, für Fortgeschrittene jQuery-Benutzer vorgestellt. Um die Arbeits- und Funktionsweise von jQuery zu verstehen, werden nach der Vorstellung der Web-Technologien alle nötigen JavaScript-Grundlagen, wie zum Beispiel das Thema Lambda-Funktionen, ausführlich besprochen. Kapitel 5 beschäftigt sich tiefgehend mit der eigentlichen Plugin-Entwicklung und erläutert anhand von zahlreichen Beispielen die Prinzipien und den Grundaufbau von Erweiterungen.

2 Einleitung

Um einen Einblick in die Entwicklung von jQuery-Plugins zu geben, werden zunächst einige wesentliche Technologien des World Wide Web erläutert. Diese werden nach Lokalität ihres Ablaufs unterschieden. HTML, CSS und JavaScript kümmern sich grundsätzlich um die Darstellung beziehungsweise um die Manipulation des In- halts, wobei JavaScript theoretisch in der Lage ist, komplexere Anwendungslogiken zu übernehmen. Der Client entspricht im Normalfall dem Benutzer der Webanwendung, wohingegen die zentrale Applikationslogik in den meisten Fällen auf dem Server des Dienstanbieters liegt.

2.1 HTML

Hypertext Markup Language, kurz HTML, ist eine textbasierte Auszeichnungssprache, die Texte, Bilder oder Links strukturiert. HTML-Dokumente bilden die Grundlage für das World Wide Web und können in einem Webbrowser dargestellt werden. Das World Wide Web Consortium (W3C) kümmert sich um die Weiterentwicklung dieses Standards. Momentan beträgt die aktuellste Versionsnummer 4.01. Parallel dazu existiert XHTML, welches HTML 4.01 in XML1definiert. HTML hingegen wird mittels SGML2definiert (SGML $\subset$ XML). HTML5 befindet sich aktuell noch in der Entwicklung, die gängigen Browser unterstützen diesen Standard jedoch schon zum Großteil. Diese Technologie bietet eine Großzahl neuer Funktionalitäten. Zum Beispiel stellt das Canvas-Element eine 2D-Zeichenflächen zur Verfügung. Dadurch bietet sich in Kombination mit der JavaScript-Zeittfunktion eine echte Alternative zu Flash. Laut W3C soll HTML5 offiziell 2014 verabschiedet werden [10].

2.2 CSS

Cascading Style Sheets, kurz CSS, ist eine deklarative Sprache, um Dokumente wie HTML oder XML mit Stilmitteln zu versehen. HTML kümmert sich also lediglich um die Strukturierung von Informationen, wohingegen CSS die Darstellung dieser Informationen deklariert.

2.3 JavaScript

JavaScript ist eine Scriptsprache, die die Firma Netscape entwickelte, welche es ermöglicht auf eine HTML-Seite Einfluss zu nehmen. In Anlehnung an die Java Syntax und aus marketingtechnischen Gründen wurde LiveScript letztendlich in JavaScript umbenannt. JavaScript Quellcode kann sowohl direkt in einer HTML-Datei oder in einer exter- nen Datei eingebunden werden. Über das DOM3, welches die Struktur des HTML- Dokuments repräsentiert, können Elemente und Inhalte beliebig manipuliert werden. So ist es zum Beispiel möglich nach dem Laden einer Seite ein HTML-Element mit Text zu füllen. Grundlage für alle Ajax-Applikationen ist der JavaScript XMLHttpRequest, welcher HTTP-Anfragen in jeglicher Form durchführen kann. Ursprünglich wurde er im Jahre 1998 von Microsoft entwickelt um for future referenceXML-Daten mit Hilfe von

JavaScript von einem Server abzurufen. Mittels dem XMLHttpRequest können natürlich nicht nur XML-Daten abgerufen werden, sondern jegliche Art von Daten.

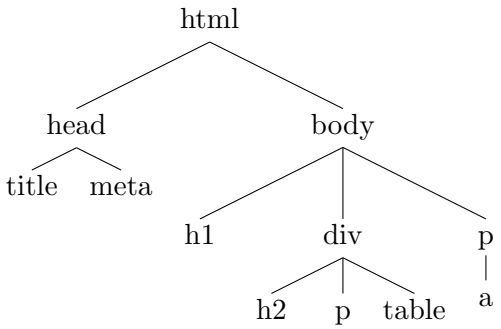

Abbildung 1: Beispiel eines DOM-Knotenbaum

2.4 Ajax

Der Begriff 'Asynchronous JavaScript and XML' - kurz 'Ajax' - entstand jedoch erst im Jahr 2005, als die ersten Ajax-Anwendungen wie zum Beispiel Google Maps auf den Markt kamen. Damit ist es möglich Daten im Hintergrund nachzuladen ohne die Seite komplett neu zu laden. Häufig taucht der Begriff 'Webseiteneffekte' auch im- mer im Zusammenhang mit Ajax auf, was jedoch genaugenommen nicht richtig ist. Webseiten-effekte haben grundsätzlich nichts mit der Ajax-Technologie zu tun. Viele JavaScript-Bibliotheken, wie auch jQuery, unterstützen jedoch das asynchrone nachladen von In-halten und bieten zusätzlich diverse Effekte wie zum Beispiel das langsame Einblenden von HTML- Komponenten an.

3 JavaScript Grundlagen

Um direkt in die Entwicklung von jQuery-Plugins einsteigen zu können, sind zunächst einige elementare JavaScript-Grundlagen zu verstehen. Anhand von kleinen Beispielen soll das Verständnis der Funktionsweise von Lambda-Funktionen und das Grundprinzip der Sichtbarkeit von Variablen erläutert werden.

3.1 Lambda-Funktion

Wie in Perl oder C++ ist es in JavaScript ebenfalls möglich, anonyme Funktionen, auch Lambda-Funktionen genannt, zu deklarieren. Im Bezug auf JavaScript bedeuted dies, dass eine Methode über einen Verweis beziehungsweise über eine Variable angesprochen werden kann. Es ist also möglich Funktionen in JavaScript-Variablen abzulegen und zu einem späteren Zeitpunkt entsprechend aufzurufen oder einer anderen Methode zu übergeben.

Quellcode 1: Lambda-Funktion - Grundgerüst

```
1  var lambdaFunktion = function(){
2      // Mach irgendwas...
3  }
```

Ein Großteil der gängigen JavaScript-Frameworks, auch jQuery, verwenden diese Gegebenheit, um einen sogenannten Callback-Mechanismus[1] zu implementieren. Das im Folgenden dargestellte Beispiel soll dieses Prinzip genauer erläutern:

[1] Callback: Quellcode, der mit Hilfe eines Arguments einer anderen Funktion übergeben werden kann und später durch diese aufgerufen wird.

Quellcode 2: Lambda-Funktion - Anwendungsbeispiel

```
1  // Erzeugung einer Lambda-Funktion mit anschliessender Zuweisung
2  var ausgabe = function(getraenk, anzahl){
3      alert("Sie habe " + anzahl + " " + getraenk + " getrunken!");
4  }
5
6  // Anlegen einer Funktion die eine Lambda-Funktion als Parameter
       uebergeben bekommt
7  function start(callbackMethode){
8      var name = "bier";
9      var anzahl = 0;
10
11     // mach irgendwas, zum Beispiel 9 Bier trinken
12     for(var i = 0; i < 9; i++){
13         anzahl++;
14     }
15
16     // Fertig mit Bier trinken, die Callback-Methode aufrufen!
17     ausgabe(name, anzahl);
18 }
19
20 // Funktion Aufrufen und Lambda-Funktion uebergeben.
21 start(ausgabe);
```

Dieser Callback-Mechanismus wird vor allem beim Event-Handling eingesetzt oder um verschiedene Komponenten, wie beispielsweise Logik und View[2], voneinander zu kapseln.

3.2 Unmittelbare Ausführung einer Lambda-Funktion

Um Lambda-Funktionen unmittelbar nach ihrer Deklaration aufzurufen, wird die Funktion mit runden Klammern umschlossen und eine Argumentenliste, die ebenfalls mit runden Klammern eingeschlossen ist, angehängt [2]. Folgendes Beispiel verdeutlicht diese Beschreibung:

Quellcode 3: Lambda-Funktion - Unmittelbare Ausführung

```
1  // Dekleration und Ausfuehrung einer Lambda-Funktion
2  (function(){
3      // Mach irgendwas...
4  })();
```

Äquivalent dazu kann die Lambda-Funktion natürlich auch unmittelbar mit Parametern aufgerufen werden [12](S.114).

[2]View: Die darstellende Komponente (Benutzeroberfläche) einer Anwendung. Diese wird häufig von der Geschäftslogik getrennt (MVC-Prinzip).

Quellcode 4: Lambda-Funktion - Unmittelbare Ausführung mit Parameter

```
1  // Dekleration und Ausfuehrung einer Labda-Funktion mit Parameter
2  (function(a){
3      // Mach irgendwas, zum Beispiel:
4      alert(a*a);     // Ausgabe: 25
5  })(5);
```

3.3 Gültigkeitsbereiche von Variablen

Das Unterkapitel 3.3 beschäftigt sich mit den Gültigkeits- und Zugriffsbereichen von im Quellcode angelegten Variablen. Prinzipiell wird in JavaScript zwischen einem globalen Gültigkeitsbereich, auch „global scope“ genannt, und einem lokalen Gültigkeitsbereich, dem „local scope“, unterschieden.

3.3.1 Globaler Gültigkeitsbereich

Würde der im Folgenden deklarierte Quellcode 5 im Browser ausgeführt werden, könnte von überall im Browserfenster (window scope) auf die Variable **tier** zugegriffen werden.

Quellcode 5: Globaler Gültigkeitsbereich

```
1  var tier = "Katze";
2
3  function welchesTier () {
4      alert(tier);
5  }
```

Die Variable **tier** wird also dem JavaScript-Objekt **window** zugeordnet. Ein Zugriff auf diese wäre also ebenfalls über den Ausdruck **window.tier** möglich.

3.3.2 Lokaler Gültigkeitsbereich

Deklariert man eine Variable innerhalb einer Funktion, so ist diese nur im Bereich des entsprechenden Anwendungsblocks sichtbar. Die Variable ist also nur im lokalen Scope sichtbar. Will man nun trotzdem von außerhalb auf die Variable zugreifen, wirft der entsprechende JavaScript-Interpreter einen sogenannte ReferenceError. Wird das Anlegen einer Variablen innerhalb einer Funktion ohne das Schlüsselwort **var** durchgeführt, so deklariert man automatisch eine globale Variable [7].

Quellcode 6: Lokaler Gültigkeitsbereich

```
1  function machIrgendwas () {
2      // Dekleration einer lokalen Variable
3      var auto = "Porsche";
4
5      // Dekleration einer globalen Variablen
6      schiff = "Yacht";
7  }
8
9  // Funktion aufrufen
10 machIrgendwas();
11
12 alert(auto); // Fehler: ReferenceError: auto is not defined
13 alert(schiff); // Ausgabe Yacht
```

Idealerweise sollte es aus Effizienzgründen vermieden werden, Variablen im globalen Gültigkeitsbereich zu verwenden, da der Zugriff auf diese relativ langsam ist. Je lokaler die Variable ist, desto schneller ist auch der entsprechende Zugriff auf diese [8].

4 jQuery

Im Laufe der Zeit entstanden einige JavaScript-Frameworks, um die Modifikation von DOM-Elementen und AJAX-Funktionalität zu vereinfachen. Hauptproblem bei der Entwicklung ohne JavaScript-Framework ist das Sicherstellen der Funktionalität unter allen gängigen Browsern. Diese sogenannte Browserkompatibilität kann ohne den Einsatz eines Frameworks nur mit sehr hohem Entwicklungs- beziehungsweise Zeitaufwand sichergestellt werden. Ein weiterer Vorteil bei der Verwendung einer JavaScript Low-Level-API[3] ist, dass spezifische Skriptsprachenkenntnisse, bedingt durch die Abstraktion der jeweiligen Bibliothek, nicht erfordlich sind.

4.1 Marktanteile

Neben jQuery existieren eine Vielzahl weiterer JavaScript-Bibliotheken. Das folgende Diagramm soll einen Überblick über die Marktanteile der jeweiligen JavaScript-Frameworks geben [11].

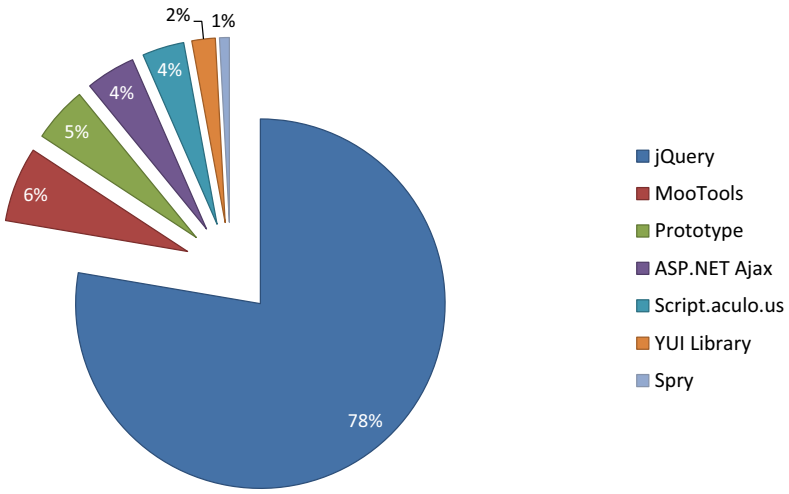

Abbildung 2: Prozentuale Anteile aller eingebundenen JavaScript-Bibliotheken, April 2013

4.2 Wie funktioniert jQuery eigentlich?

Zunächst soll ein einfaches Beispiel (Quellcode 9) die Funktionsweise von jQuery verdeutlichen. Auf den ersten Blick ist diese sehr schnell zu verstehen, was passiert jedoch hinter den Kulissen?

[3]Low-Level-API: Abstrahiert eine niedrige Entwicklungsebene.

Quellcode 7: jQuery Beispiel

```
1  $('a').css('text-decoration','underline');
```

Mit Hilfe des $-Operators ist es möglich Elemente als jQuery-Objekt anzufordern. Diese Objekte besitzen alle gängigen jQuery-Funktionen, wie beispielsweise **css()**, **attr()** oder **click()**.

Was verbirgt sich jedoch hinter dem so oft verwendeten $-Zeichen? Ein Einblick in die Implementierung von jQuery klärt diese Frage sehr schnell und unspektakulär auf. Das $-Zeichen ist nichts anderes als eine Variable, die der Implementierung des jQuery-Objektes entspricht. Beim Aufruf von $("#irgendeineID") wird die Methode jQuery.fn.init aufgerufen und eine jQuery-Objekt-Instanz zurückgegeben.

Quellcode 8: Auszug jQuery Implementierung (Version 1.9.1)

```
1  ...
2  jQuery = function( selector, context ) {
3      // The jQuery object is actually just the init constructor 'enhanced
           '
4      return new jQuery.fn.init( selector, context, rootjQuery );
5  },
6  ...
7  ...
8  // Expose jQuery to the global object
9      window.jQuery = window.$ = jQuery;
10 ...
```

Jedes jQuery-Objekt hat mit Hilfe des Objektes **$.fn** Zugriff auf alle jQuery-Objekt-Methoden.

5 Plugins

Um auf die Plugin-Entwicklung näher eingehen zu können, werden zunächst einige allgemeine Tatsachen beleuchtet, damit die Frage geklärt werden kann, was jQuery-Plugins überhaupt sind und in welchem Kontext man diese einsetzen sollte. Prinzipiell sind jQuery-Plugins nicht mehr und nicht weniger als in sich abgeschlossene Funktionserweiterungen der eigentlichen Bibliothek. Sinn und Zweck eines Plugins ist es, eine in sich abgeschlossene Funktionalität logisch vom Rest der Applikation zu kapseln. Ein Plugin hat somit keine logische Abhängigkeit nach außen. Dies hat den Vorteil, dass Plugins flexibel, leicht zu warten und wiederverwendbar sind [12](S.115).
Um diese Paradigmen einzuhalten gibt es einige Regeln, an die man sich als Plugin-Entwickler unbedingt halten sollte. Leider existieren auf dem Markt eine Vielzahl von Plugins, die dies nicht tun. So kommt es immer wieder vor, dass private Funktionen von Plugins in die globale Umgebung des Browser exportiert werden. Eine weitere Richtlinie bei der Plugin-Entwicklung ist, dass jedes Plugin stets nur eine Kernfunktionalität implementieren sollte. Bei umfangreicheren Anforderungen an das Plugin sollte die Gesamtfunktionalität in mehrere Plugins abgespalten werden.

Es macht also keinesfalls Sinn, jQuery-Plugins anderen Entwicklern nur anzubieten, um einen bestimmten Funktionsumfang zu erreichen. Vielmehr kann es bei der Plugin-Entwicklung auch darum gehen, eigene Projekt- oder Anwendungslogik mit Hilfe eines Plugins zu abstrahieren, um sich so einen besseren Überblick bei der entsprechenden Implementierung der jeweiligen Anwendung zu bewahren.

5.1 Namenskonventionen

Bei der Plugin-Entwicklung für jQuery gibt es einige wichtige Regeln, die unbedingt eingehalten werden sollten oder sogar müssen, um eine möglichst unkomplizierte Entwicklung zu gewährleisten. Plugins werden in gewöhnlichen Textdateien ausgeliefert. Für jQuery-Plugins gilt folgende Namenskonvention:

```
jquery.[Pluginname].js
```

Würde man also beispielsweise ein Plugin mit dem Namen „texteditor“ entwickeln, würde man die entsprechende JS-Datei wie folgt benennen:

```
jquery.texteditor.js
```

Häufig verwenden jQuery-Plugins eigene CSS-Dateien, um entsprechenden HTML-Elementen eine eigene Formatdefinition zu geben. Äquivalent zu der Namenskonvention des Plugins gilt hier Folgendes:

```
jquery.[Pluginname].css
```

Entsprechend für das Plugin mit dem Namen „texteditor“.

```
jquery.texteditor.css
```

Besteht die Anforderung, Bilder oder sonstige Medien mitzuliefern, sollte sich das jeweilige Plugin und die entsprechende CSS-Datei in einem eigenen Ordner befinden. Hier gilt folgende Namenskonvention:

```
jquery.[Pluginname]
```

Bilder oder Videos sollten in den entsprechenden Unterordnern „images“ oder „videos“ ausgeliefert werden.

```
jquery.[Pluginname]/images/[Bildname].png
jquery.[Pluginname]/videos/[Videoname].mp4
```

5.2 Neue jQuery-Objekt-Methode

Wie bereits in Kapitel 4.2 erwähnt, befinden sich alle jQuery-Objekt-Methoden im **$.fn**-Objekt. Will man also eine zusätzliche jQuery-Objekt-Methode implementieren, so hat man die Möglichkeit, das $.fn-Objekt wie folgt um weitere Methoden zu erweitern:

Quellcode 9: jQuery primitives Plugin

```
1  jQuery.fn.underline = function() {
2      this.css( 'text-decoration', 'underline' );
3      return this;
4  }
5
6  $('a').underline();
```

Wie in Quellcode 9 zu erkennen, wurde bei der Deklaration der Methode **underline** auf die $-Kurzschreibweise aus Kompatibilitätsgründen mit anderen JavaScript-Frameworks verzichtet. Verwendet ein JavaScript-Framework ebenfalls die $-Variable, so würden die implementierten jQuery-Plugins nicht mehr funktionieren. Auch innerhalb der Methode **underline** sollte bei einer derartigen Plugin-Dekleration auf das $-Zeichen verzichtet werden. Wie man dennoch das beliebte $-Zeichen innerhalb der jQuery-Plugin-Entwicklung verwenden kann, wird in im folgenden Kapitel erläutert.

5.3 Pseudonym $ beibehalten

Mit Hilfe einer unmittelbar ausgeführten Lambda-Funktion, wie bereits in Kapitel 3.2 erläutert, ist es möglich, die $-Variable innerhalb des Plugin-Kontextes beizubehalten.

Quellcode 10: jQuery Plugin $-Variable beibehalten

```
1  // Plugin wird innerhalb einer unmittelbar aufgerufenen
2  // Lambda-Funktion deklarieren.
3
4  // Empfangener Uebergabeparameter wird als $-Zeichen deklariert
5  (function ( $ ) {
6
7      // Nun kann das $-Zeichen problemlos verwendet werden.
8      $.fn.underline = function() {
9          this.css( 'text-decoration', 'underline' );
10
11         // Original jQuery-Objekt zurueckliefern
12         return this;
13     };
14
15 }( jQuery )); // jQuery-Objekt als Uebergabeparameter
```

5.4 Verketteter Aufruf

Ein wichtiges Paradigma bei der jQuery-Plugin-Entwicklung ist, dass die aufgerufene jQuery-Objekt-Methode wiederum das Original-jQuery-Objekt zurückliefert. Im jQuery-Funktionspool gibt es nur wenige Funktionen, wie zum Beispiel **width()**, die sich aus Funktionalitätsgründen nicht an dieses Paradigma halten. Durch Zeile 12 in Quellcode 10 wird das Original-jQuery-Objekt zurückgeliefert und kann somit unmittelbar verkettet aufgerufen werden. Das im Folgenden dargestellte Beispiel erläutert dieses Prinzip.

Quellcode 11: Verketteter Aufruf von jQuery-Methoden

```
1  $('a').underline().css('color', 'red').addClass('active');
```

5.5 Mehrere Objekte behandeln, each() verwenden

Typischerweise können HTML-Elemente ebenfalls HTML-Kindelemente besitzen. Das bedeutet, dass ein jQuery-Objekt mehrere Referenzen auf andere jQuery-Objekte besitzen kann. Falls man nicht nur das übergebene, sondern auch die beinhalteten Objekte behandeln will, bietet es sich an, folgendes **each()**-Konstrukt (siehe Quellcode 12) zu verwenden.

Quellcode 12: Jedes Element behandeln

```
1  (function ( $ ) {
2      $.fn.underline = function() {
3
4          // Jedes Element unterstreichen
5          return this.each(function() {
6              $(this).css( 'text-decoration', 'underline' );
7          })
8      };
9
10 }( jQuery ));
```

Bei der Iteration über alle DOM-Objekte wird die Callback-Methode im Kontext des aktuellen HTML-Elementes, repräsentiert durch eine Zeichenkette, aufgerufen. Um nun auf jQuery-Methoden zugreifen zu können, ist es von Nöten, mit Hilfe der Dollar-Variablen ein jQuery-Objekt aus dem entsprechenden HTML-Tag zu erzeugen.

5.6 Einstellungen und Optionen

Einstellungen oder Optionen bieten dem Benutzer die Möglichkeit, ein entsprechendes Plugin konfigurierbar zu machen. Somit erhöht sich nicht nur die Wiederverwendbarkeit, sondern der jeweilige Plugin-Benutzer hat die Möglichkeit, selbst Kontrolle über bestimmte Plugin-Funktionalitäten zu übernehmen. In der jQuery-Plugin-Welt hat sich mittlerweile ein De-facto-Standard zur Individualisierung von Plugins entwickelt. Das im Folgenden dargestellte Code-Beispiel(Quellcode 14), veranschaulicht die Behandlung von Optionen anhand eines Plugins, welches jQuery-Objekte verschiedenartig unterstreicht [12](S.123).

Quellcode 13: jquery.underline.js - jQuery Plugin mit Optionen

```
1  (function ( $ ) {
2      // Optionen koennen optional uebergeben werden
3      $.fn.underline = function( options ) {
4
5          // Standartwerte fuer das Plugin werden mit dem
6          // uebergebenen Objekt 'options' kombiniert
7          var settings = $.extend({
8              type: 'normal',
9              color: 'black'
10         }, options );
11
12         // Jedes Element unterstreichen
13         return this.each(function(){
14             if(settings.type == 'normal'){
15                 $(this).css('border-bottom', '1px solid ' + settings.
                       color);
16             } else if (settings.type == 'double'){
17                 $(this).css('border-bottom', '3px double ' + settings.
                       color);
18             }
19         });
20     };
21 }( jQuery ));
```

Das Beispiel-Plugin **jquery.underline.js** erwartet eine entsprechende Konfiguration über den Übergabeparameter **options**. Mit Hilfe der jQuery-Methode **$.extend()** können zwei Objekte zusammengeführt werden, wobei dabei lediglich das zuerst übergebene Objekt modifiziert wird [3]. In Quellcode 14 bedeutet dies, dass die Attribute des übergebenen Objektes **options** in das Objekt **settings** kopiert werden. Bei gleichnamigen Attributen werden die aktuell vorhandenen Werte überschrieben. Somit werden die übergebenen Einstellungen des Benutzers erfolgreich mit den eigentlichen Standard-Einstellungen des Plugins kombiniert.

Über „settings.[attributname]“ kann im weiteren Plugin-Verlauf auf die entsprechenden Einstellungsparameter zugegriffen werden (Zeile 14-17).

Quellcode 14: Einstellungen in einem Plugin verwenden

```
1  <html>
2  <head>
3      <script src="jquery.1.9.1.js" type="text/javascript"></script>
4      <script src="jquery.underline.js" type="text/javascript"></script>
5
6      <script type="text/javascript">
7
8      // Sobald alle jQuery Funktionen geladen worden sind
9      $( document ).ready(function() {
10
11         // Plugin verwenden, alle 'span'-Elemente
12         // doppelt rot unterstreichen
13         $("span").underline({
14             type : 'double',
15             color: 'red'
16         });
17     });
18     </script>
19 </head>
20 <body>
21     <div>
22         <span>Hochschule Karlsruhe - HSKA - 2013</span>
23     </div>
24 </body>
25 </html>
```

Um das Plugin verwenden zu können, muss man auf einem entsprechend selektierten HTML-Element oder mehreren selektierten Elementen, wie in unserem Beispiel alle span-Elemente, mit Hilfe der Plugin-Methode **underline(...)** die jeweiligen Elemente in einer beliebigen Farbe unterstreichen.

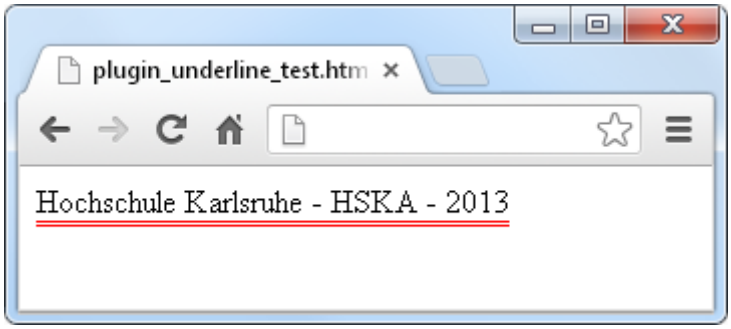

Abbildung 3: Screenshot underline-Plugin Demo

5.7 Öffentlicher Zugriff auf Plugin-Standardwerte

Bezüglich Kapitel 5.6 kann eine weitere Verbesserung an unserem bereits vorhandenen Quellcode vorgenommen werden. Damit die Standardwerte unseres Plugins von außen aus sichtbar und zugänglich sind, sollten diese in einem globalen Objekt abgespeichert werden [12](S.124). Damit der JavaScript-Namensraum nicht unnötig beschrieben wird, bietet es sich an, innerhalb des Plugin-Funktionsobjektes ein Objekt mit dem Namen [5], welches alle Standardwerte deklariert, zu implementieren.

Quellcode 15: jquery.underline.js - Globale Einstellungen

```
1  (function ( $ ) {
2      // Optionen koennen optional uebergeben werden
3      $.fn.underline = function( options ) {
4          ...
5      };
6
7      // Plugin Standardwerte global deklarieren
8      $.fn.underline.defaults = {
9          type: 'normal',
10         color: 'black'
11     };
12 }( jQuery ));
```

Interner Zugriff auf die definierten Parameter kann, wie im Folgenden gezeigt, durchgeführt werden.

Quellcode 16: Globale Einstellungen Zusammenführen

```
1 var settings = $.extend( {}, $.fn.underline.defaults, options );
```

Um das „defaults“-Objekt nicht zu überschreiben, wird das erste Argument der extend-Methode als leeres Objekt[4] definiert. Somit bleiben alle Standardwerte im Objekt „defaults“ erhalten und stehen auch noch im weiteren Skriptverlauf zur Verfügung [1].

5.8 Plugin Erweiterung offerieren

Der im Folgenden vorgestellte Mechanismus, welcher dazu benutzt werden kann, dass Entwickler ein entsprechendes Plugin erweitern können [6]. Hier im Beispiel wird eine Methode im Funktionsobjekt des Plugins „underline “ deklariert, die sich um die Formatierung des zu unterstreichenden Textes kümmern soll. Standardmäßig wird dieser unformartiert zurückgeliefert (siehe Quellcode 17, Zeile 24-26).

[4] In JavaScript wird ein leeres Objekt mit Hilfe von zwei geschwungenen Klammern definiert: {}

Quellcode 17: Öffentliche Methode zur Überschreibung deklarieren

```
1  (function ( $ ) {
2
3      $.fn.underline = function( options ) {
4          var settings = $.extend({}, $.fn.underline.defaults, options);
5
6          return this.each(function(){
7              var content = $(this).html();
8              $(this).html($.fn.underline.format(content));
9
10             if(settings.type == 'normal'){
11                 $(this).css('border-bottom', '1px solid ' + settings.
                       color);
12             } else if (settings.type == 'double'){
13                 $(this).css('border-bottom', '3px double ' + settings.
                       color);
14             }
15         });
16     };
17
18     $.fn.underline.defaults = {
19         type: 'normal',
20         color: 'black'
21     };
22
23     // Formatfunktion definieren, die von aussen ueberschrieben werden
           kann
24     $.fn.underline.format = function( txt ) {
25         return txt;
26     };
27 }( jQuery ));
```

Ein potentieller Benutzer dieses Plugins hat nun die Möglichkeit die Methode **$.fn.underline.format** zu überschreiben. In Quellcode 18, Zeile 11 wird die Formatfunktion des Funktionsobjektes „underline“ neu deklariert. Wird das Plugin, wie in Zeile 17 verwendet, so wird der Text im span „Hochschule Karlsruhe - HSKA - 2013“ nicht nur unterstrichen, vielmehr werden vor beziehungsweise nach dem Text die Zeichen „-=[“ und „]=-“ platziert.

Quellcode 18: Formatfunktion des Plugins überschreiben

```
1  <html>
2  <head>
3      <script src="jquery.1.9.1.js" type="text/javascript"></script>
4      <script src="jquery.underline.js" type="text/javascript"></script>
5      <script type="text/javascript">
6
7      // Sobald alle jQuery Funktionen geladen worden sind
8      $( document ).ready(function() {
9
10         // Ueberschreibe Format-Methode des Plugins
11         $.fn.underline.format = function(txt){
12             return '-=] ' + txt + ' [=-';
13         }
14
15         // Plugin verwenden, alle 'span'-Elemente
16         // doppelt rot unterstreichen
17         $("span").underline({
18             type : 'double',
19             color: 'red'
20         });
21     });
22     </script>
23 </head>
24 <body>
25     <div>
26         <span>Hochschule Karlsruhe - HSKA - 2013</span>
27     </div>
28 </body>
29 </html>
```

Beim Aufruf der HTML-Seite wird diese wie folgt dargestellt:

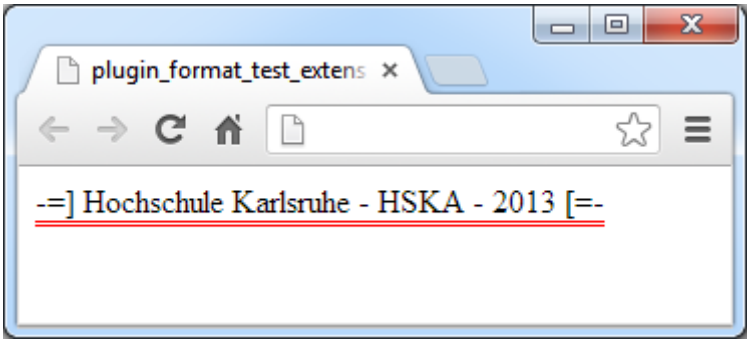

Abbildung 4: Screenshot underline-Plugin Demo mit Formaterweiterung

5.9 Private Funktionen

Dem Benutzer die Möglichkeit zu bieten Plugin-Methoden zu überschreiben, kann, wie soeben in Kapitel 5.8 besprochen, sehr hilfreich sein. Jedoch sollte man sehr gut darüber nachdenken, welche Funktionen man freigeben und welche man privat halten sollte. Falls man sich nicht sicher ist, ob man eine Methode öffentlich zugänglich machen sollte, wird aus Kompatibilitätsgründen empfohlen darauf zu verzichten [3]. Private Funktionen kann man deklarieren, in dem man im lokalen Scope des Plugins eine Methode anlegt. Folgendes Beispiel erläutert das Prinzip einer lokalen, nach außen nicht sichtbaren Funktion:

Quellcode 19: Private Funktion implementiert Debug-Ausgabe

```
1  (function( $ ) {
2      $.fn.underline = function( options ) {
3          debug( this );
4      };
5
6      // Private Funktion implementiert eine Debug-Ausgabe,
7      // kann von ausserhalb nicht aufgerufen werden
8      function debug( $obj ) {
9          window.console.log( "underline anzahl elemente: " + $obj.size()
               );
10     };
11 })( jQuery );
```

5.10 Callback-Mechanismen anbieten

Abschließend möchte ich einen ebenfalls sehr wichtigen Punkt bei der Plugin-Entwicklung ansprechen; die Verwendung von Callback-Methoden zur Kommunikation zwischen einem Plugin und der Schicht, welche das Plugin verwendet. Wie bereits in Kapitel 3.1 besprochen, besteht bei JavaScript die Möglichkeit anonyme Funktionen in die Übergabeparameterliste einer anderen Funktion zu übergeben. Zur Veranschaulichung der Vorteile eines Callback-Mechanismus innerhalb eines Plugins folgt ein modifiziertes underline-Plugin, welches eine definierte Callback-Funktion beim Klicken auf ein unterstrichenes HTML-Element ausführt [9] [4].

Quellcode 20: Underline Plugin mit Callback-Methode

```
1  (function ( $ ) {
2      $.fn.underline = function( options ) {
3          var settings = $.extend({}, $.fn.underline.defaults, options);
4
5          return this.each(function(){
6              if(settings.type == 'normal'){
7                  $(this).css('border-bottom', '1px solid ' + settings.
                       color);
8              } else if (settings.type == 'double'){
9                  $(this).css('border-bottom', '3px double ' + settings.
                       color);
10             }
11
12             // Klick-Event mit Methode verknuepfen
13             $(this).on( "click", settings.onElementClicked);
14         });
15     };
16
17     $.fn.underline.defaults = {
18         type: 'normal',
19         color: 'black',
20
21         // Deklaration einer leeren, anonymen Funktion
22         // in den Standardwerten defaults
23         onElementClicked : function() {}
24     };
25 }( jQuery ));
```

Damit die Existenz der Methode **onElementClicked** vor dem Aufruf nicht überprüft werden muss, wird in Zeile 23 in Quellcode 20 eine leere, anonyme Funktion innerhalb des defaults-Objektes deklariert.

Quellcode 21: Aufruf des underline-Plugins mit Callback-Methode

```
1  <html>
2  <head>
3      <script src="jquery.1.9.1.js" type="text/javascript"></script>
4      <script src="jquery.underline.js" type="text/javascript"></script>
5      <script type="text/javascript">
6
7      // Sobald alle jQuery Funktionen geladen worden sind
8      $( document ).ready(function() {
9
10         // Plugin verwenden, alle 'span'-Elemente
11         // doppelt rot unterstreichen
12         $("span").underline({
13             type : 'double',
14             color: 'red',
15             onElementClicked : function (){
16                 // Ueberschreiben der defaults-Methode onElementClicked
17                 $(this).css('background-color', 'yellow');
18             }
19         });
20     });
21     </script>
22 </head>
23 <body>
24     <div>
25         <span>Hochschule Karlsruhe - HSKA - 2013</span>
26     </div>
27 </body>
28 </html>
```

Beim Aufruf des underline-Plugins kann nun entsprechend die Callback-Methode **onElementClicked** individuell überschrieben werden. Hier in Quellcode 21, Zeile 15 wird beispielsweise die Hintergrundfarbe des aktuell angeklickten Elementes auf Gelb gesetzt. Klickt man nun den Schriftzug „Hochschule Karlsruhe - HSKA - 2013“ an, so wird die HTML-Seite wie folgt dargestellt:

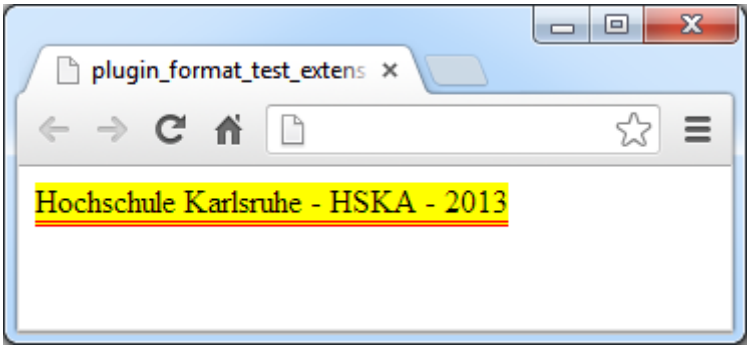

Abbildung 5: Screenshot underline-Plugin Demo nach Klick auf Schriftzug

6 Fazit

Bei der Entwicklung komplexer, auf JavaScript basierender, Webapplikationen können jQuery-Plugins einen großen Beitrag dazu leisten, dass der geschriebene Quellcode, trotz hohen Umfanges, übersichtlich und verständlich bleibt. Täglich werden neue, großartige jQuery-Plugins vorgestellt, die oft genial, jedoch sehr einfach zu benutzen sind. Der Vorteil liegt klar auf der Hand. jQuery ist eine kompakte, schnelle und vorallem intuitiv zu benutzende Bibliothek, welche die tägliche Arbeit eines jeden Webentwicklers sehr stark vereinfachen kann. Die Möglichkeit, ohne großen Aufwand, selbst Plugins zu schreiben, treibt die Community hinter jQuery täglich an, um neue und ausgefallenere Plugins zu entwickeln.

Abbildungsverzeichnis

Literatur

[1] JQuery. API - jQuery.extend(). 2013.

[2] JQuery. Functions - Immediately-Invoked Function Expression (IIFE). 2013.

[3] JQuery. Keep Private Functions Private. 2013.

[4] JQuery. Provide Callback Capabilities. 2013.

[5] JQuery. Provide Public Access to Default Plugin Settings. 2013.

[6] JQuery. Provide Public Access to Secondary Functions as Applicable. 2013.

[7] Robert Nyman. Explaining JavaScript scope and closures.

[8] Addy Osmani. PerformanceWriting Fast, Memory-Efficient JavaScript.

[9] Stackoverflow. jQuery Plugin: Adding Callback functionality. 2013.

[10] W3C. W3C Confirms May 2011 for HTML5 Last Call, Targets 2014 for HTML5 Standard. 2013.

[11] W3Techs. Usage of JavaScript libraries for websites.

[12] Jakob Westhoff. *Plug-in-Entwicklung mit jQuery.* 2010.