Clevere Web-Formulare mit APEX und jQuery

Andreas Wismann
MT AG
Ratingen

Schlüsselworte:

Oracle Application Express, APEX, jQuery, JavaScript, DOM, CSS-Selektoren, HTML

Einleitung

Oracle Application Express verwendet „unter der Haube“ ein JavaScript-Framework, um in Browser-Formularen Annehmlichkeiten zu ermöglichen wie beispielsweise Kalender-Popups und Wertelistenfenster. Seit APEX 4.0 wird jQuery nun offiziell als integraler Bestandteil des Produkts mitgeliefert, dessen volles Potenzial dem Entwickler zur Verfügung steht. In diesem Beitrag geht es um den Einstieg in die zahlreichen spannenden Möglichkeiten, die jQuery zur Steigerung der Benutzerfreundlichkeit und Akzeptanz von Web-Formularen zu bieten hat. Der Fokus liegt auf jQuery-Entwurfsmustern, die nur minimale Änderungen des üblichen APEX-Entwicklungsprozesses benötigen und auch mit APEX 3.2.1 funktionieren.

Dieser Teil widmet sich eher den grundlegenden jQuery-Kenntnissen und zeigt die notwendigen Vorarbeiten, um eigene jQuery-Skripte im APEX-Kontext einzusetzen. Beispielhaft wird ein einfacher Anwendungsfall durchgespielt.

Im Vortrag möchte ich lieber weitere Features live demonstrieren (was auch für Nicht-Coder interessant sein könnte) und verweise für die weniger spannenden Grundlagen auf dieses Manuskript.

Die jQuery-Familie

j*Query*? Um Missverständnissen gleich vorzubeugen: Der Namensbestandteil *Query* steht nicht für „Abfrage“ im Sinne von SQL, vielmehr ist die Selektion und Manipulation von Bestandteilen des HTML-Dokuments gemeint, genauer: des DOM-Baums. Dabei verwendet jQuery eine Syntax, die weitgehend der CSS-3.0-Spezifikation entspricht (und diese zum Teil übertrifft – und das selbst in älteren Browsern wie dem Internet Explorer 6, der sich schon an CSS 2 verschluckt).

jQuery[1] normalisiert den Einsatz von JavaScript innerhalb der diversen Browserumgebungen, indem es eine schlanke (ca. 70k) API-Abstraktionsschicht anbietet. Da jQuery selbst ebenfalls in JavaScript geschrieben ist, kann sowohl mit konventionellen JavaScript-Befehlen als auch mit den zumeist mächtigeren jQuery-Pendants programmiert werden. Frei nach dem offiziellen Produktmotto „write less, do more“ wird der Entwickler dadurch in die Lage versetzt, in allen unterstützten Browsern und Plattformen[2] den selben Code zu verwenden, ohne sich mit den leidigen Ärgernissen der browserabhängigen Programmierung herumzuschlagen. Aufgrund der höchst flexiblen DOM-Selektorsyntax lässt sich gegenüber herkömmlicher JavaScript-Programmierung zudem jede Menge Entwicklungs- und Testaufwand einsparen.

[1] http://www.jquery.com/
[2] Internet Explorer ab Version 6; Firefox ab Version 2, Safari ab Version 3, Opera ab Version 9.
Für mobile Geräte siehe http://jquerymobile.com/

Man kann jQuery mit anderen, ebenfalls frei verfügbaren Frameworks vergleichen wie Dojo (welches in einigen IBM-Produkten eingesetzt wird) und Prototype. Es ist sogar möglich, wenn auch nicht unbedingt effizient, mehrere verschiedene Frameworks im selben Projekt, ja sogar in der selben Webseite einzusetzen.

jQuery existiert seit 2006 und besteht aus dem gleichnamigen Minimalpaket (in APEX 4.0 steckt die Version 1.4.2), das sämtliche Basisfunktionalität, aber auch grundlegende optische Effekte wie Ein- und Ausblenden oder Animation von Seitenelementen zur Verfügung stellt, sowie der optionalen Erweiterungsbibliothek „jQuery UI“ (Version 1.8), die Oberflächenkomponenten und komplexere Verhalten hinzufügt. Bei Bedarf können zahlreiche Plugins aus frei verfügbaren oder kommerziellen Quellen hinzugefügt werden[3]. APEX 4.0 verwendet beispielsweise das *jQuery Calendar*-Plugin, wenn der Entwickler ein Datumseingabefeld mit Kalenderkomponente auswählt.

Selbstgeschriebene jQuery-Skripte verbrauchen also im Falle von APEX 4.0 keinen nennenswerten Overhead während des Ladens der Webseite, da die benötigten Basisbibliotheken bereits mit der zuerst geladenen APEX-Seite ausgeliefert werden und der Browser alle weiteren Anforderungen aus dem Cache bedient. Im Falle von APEX 3.2 kann die Datei jquery-1.4.2.min.js entweder auf dem Anwendungsserver (im APEX-Verzeichnis /i/) zur Verfügung gestellt oder – falls dazu keine Möglichkeit besteht – unter *Gemeinsame Komponenten* hinzugefügt oder sogar live über zuverlässige Webquellen wie dem Google Content Distribution Network bezogen werden. Ein entsprechender Aufruf im `<head>` der HTML-Seite sieht wie folgt aus:

```
<script
src="http://ajax.googleapis.com/ajax/libs/jquery/1.4.2/jquery.min.js"
type="text/javascript">
</script>
```

Abgrenzung zu APEX‘ Dynamic Actions

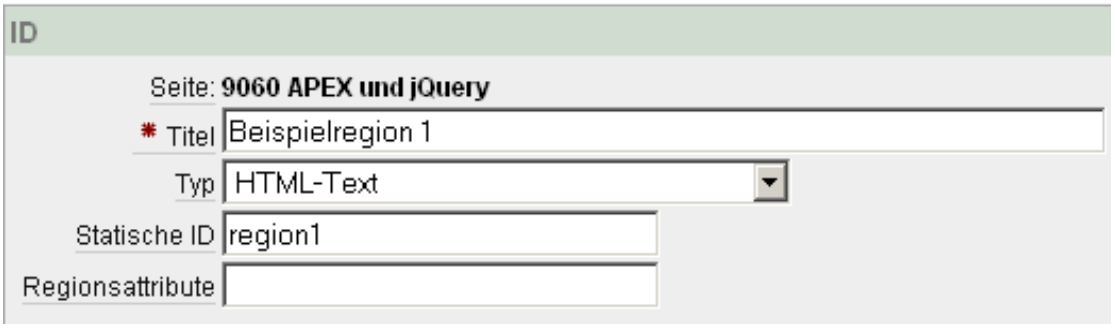
Mit APEX 4.0 führt Oracle die *Dynamic Actions* ein: praktische, vorgefertigte Verhaltensmuster, die man konkreten Seitenelementen zuordnen kann. Es handelt sich dabei um einen deklarativen Ansatz, der JavaScript-Funktionen und Ereignishandler generiert, die man mit eigenem Code ergänzen kann (aber nicht muss). Dynamic Actions sind nicht Gegenstand dieses Beitrags; gleichwohl bieten sie eine hervorragende Gelegenheit, wohldefinierte Einstiegspunkte für eigene jQuery-Skripte festzulegen.

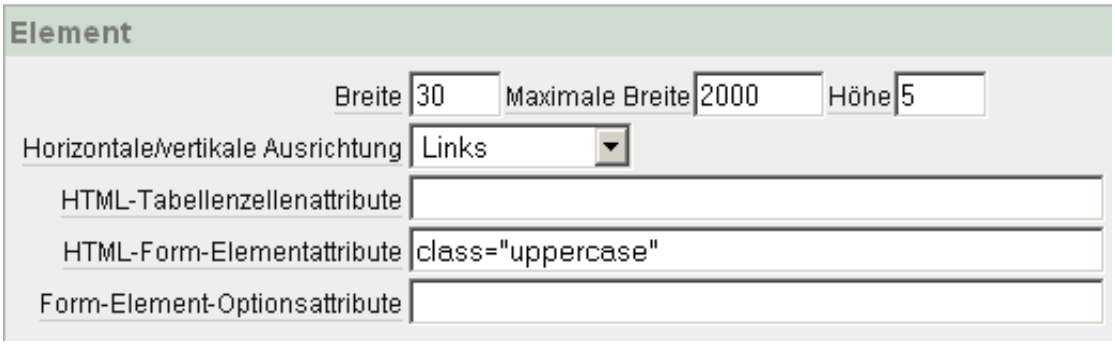
In Medias Res

Ausgehend von einer konkreten Idee, was man denn eigentlich erreichen möchte (dazu gleich mehr) wird man bei der Entwicklung mit jQuery unter APEX typischerweise in folgenden Schritten vorgehen:

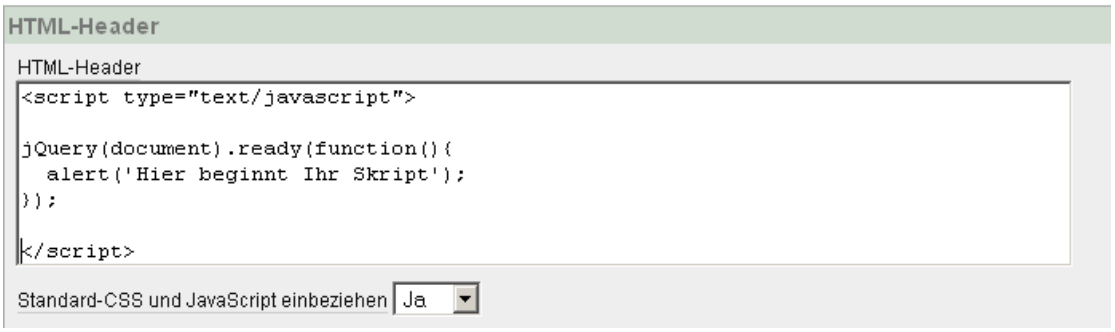
1. Die zu manipulierenden Seitenelemente (Ein- und Ausgabefelder, Berichte, Regionen etc.) werden mit einer statischen ID ausgezeichnet, sofern APEX nicht bereits eine solche vergeben hat. Für mehrere Elemente, die sich gleichartig verhalten sollen, trägt man im HTML-Form-Elementattribut ein oder mehrere selbstdefinierte *class*-Attribute ein:

[3] jQuery-Plugins sind nicht zu verwechseln mit APEX-Plugins. Letztere ermöglichen es, neue Komponenten anstelle bereits bestehender Item-Typen deklarativ zu verwenden.





2. Seitenspezifische oder anwendungsweite jQuery-Skript-Tags werden erstellt, beispielsweise im HTML-Header der APEX-Seite:



   und in diesen Skripten

3. werden die gekennzeichneten Elemente aus Schritt 1 mit jQuery-Selektoren ausgelesen

4. und mit jQuery-Methoden nach Wunsch manipuliert oder mit neuen Fähigkeiten ausgestattet.

Selektoren

Je nach verwendetem Template (in unserem Fall: „One Level Tabs“ aus dem von Oracle mitgelieferten Theme 20) generiert APEX nach den Schritten 1 und 2 typischerweise HTML-Code wie diesen:

```
...
<div id="t20ContentMiddle" >
<table class="t20Region t20ReportsRegion100" id="region1"  border="0"
cellpadding="0" cellspacing="0" summary="" >
...
<tr>
<td class="t20RegionBody">
<table id="apex_layout_39224909074645466" class="formlayout" summary="" >
<tr>
<td nowrap="nowrap" align="right">
<label tabindex="999" for="P9060_TEXT_1">
<span class="t20OptionalLabel">Text 1</span>
</label>
</td>
<td colspan="1" rowspan="1" align="left">
<input type="hidden" name="p_arg_names" value="39225527184483427" />
<input type="text" name="p_t01" size="30" maxlength="2000" value=""
id="P9060_TEXT_1"  class="uppercase" />
</td>
</tr>
</table></td>
</tr>
</table>
</div>
...
```

Den jQuery-Code schreibt man am besten so, dass er sich an diesen möglichst spezifischen Selektoren (grau hinterlegt) orientiert und nur Dinge tut, die der Datenvisualisierung bzw. Entgegennahme der eingebenen Daten dienen. Für dieses Paradigma der Programmierung ist es kennzeichnend, dass APEX selbst davon praktisch nichts mitbekommt. Das ist auch nicht nötig, denn wir manipulieren nur Abläufe im Bereich der Benutzerinteraktion.
Das heißt,

- es sind keinerlei Eingriffe in die serverseitigen Abläufe und PL/SQL-Skripte nötig (anders als beispielsweise unter dem hervorragenden Framework ApexLib[4], wo neben dem clientseitigen JavaScript auch die PL/SQL-Serverroutinen verändert werden, mit denen APEX das HTML generiert und die Formulareingaben auswertet – beide Ansätze haben ihre Vor- und Nachteile),
- die Entwicklung der Skripte kann in geeigneten Entwicklungsumgebungen und durch spezialisierte JavaScript-Entwickler erfolgen, die nur wenig bzw. gar kein APEX-spezifisches Knowhow benötigen.

Wie man im Quellcodeauszug erkennt, verwendet APEX üblicherweise den deklarierten Namen des Elements als dessen *id* im HTML-Code:

[4] http://apexlib.oracleapex.info/


```
<input type="text" ... id="P9060_TEXT_1" ... />
```

Die *id* eines HTML-Elements ist dokumentweit eindeutig und stellt immer die zuverlässigste (und oft auch die schnellste) Möglichkeit dar, auf das Element mit jQuery zuzugreifen.

Stellen wir uns nun vor, das Textitem dient der Eingabe von Daten, deren Buchstaben stets Großbuchstaben sein müssen (Foreign-Key, historisch gewachsen etc. pp). Selbstverständlich wird man diese einfache Anforderung mit APEX- und Datenbank-Bordmitteln leicht lösen können – man nehme eine APEX-Validierung und unterfüttere sie (wenn möglich) mit einem Tabellentrigger für den Insert- und Updatefall, der ein upper() ausführt. Schlimmstenfalls existiert vielleicht nur ein Constraint auf der Tabelle, dessen Verletzung eine (mehr oder weniger verwirrende) Fehlermeldung auf der nachfolgenden APEX-Seite hervorruft und das DML-Statement scheitern lässt. So weit so – gut?

Nun, es wäre viel smarter, wenn man den Benutzer schon bei der Eingabe unterstützt. Wir erstellen also ein Skript, das gezielt jedem Textfeld innerhalb der *region1* mit der Klasse *uppercase* (siehe Abbildung zu Schritt 1) ein Verhalten zuweist, das während der Eingabe alle Buchstaben in GROßBUCHSTABEN umwandelt und führende bzw. endende Leerzeichen entfernt:

```
1     jQuery(document).ready(function(){
2       // alert('Hier beginnt Ihr Skript');
3       jQuery('input.uppercase', '#region1')
4         .bind('keyup paste change', function(event){
5           var $me = jQuery(event.target);
6           $me.val(
7             jQuery.trim($me.val()).toUpperCase()
8           );
9         });
10    });
```

Wenn man so will, wurde wir hier rein syntaktisch gesehen ein einziger(!) jQuery-Befehl formuliert, nämlich in den Zeilen 1 und 10: „Führe (einmalig) eine anonyme Funktion aus, sobald das Dokument fertig im DOM-Baum geladen ist“. Diese Funktion beinhaltet:

Zeile 3: Suche alle input-Items mit der Klasse *uppercase*, die sich innerhalb eines DOM-Elements mit dem *id*-Attribut *region1* befinden (letzteres Argument ist optional, es grenzt aber die Suche auf einen bestimmten Kontext des DOM-Baums ein und kann die Skriptausführung beschleunigen, insbesondere bei großen, komplexen Dokumenten).

Zeile 4: Weise allen Items aus der Ergebnismenge in Zeile 3 einen anonymen Eventhandler zu, der auf die Ereignisse *Tastendruck*, *Einfügen* und *Wert ändern* reagiert.

Zeile 5: Deklariere eine lokale Variable im Eventhandler, die das betreffende Item aus dem *event*-Parameter ausliest und es in ein jQuery-Element umwandelt. Man käme zwar auch ohne diesen Zwischenschritt aus, dann müsste man jedoch einmal mehr die Funktion *jQuery(event.target)* aufrufen, was teurer ist als das Zuweisen und Auslesen der Variablen. Außerdem erleichtert man sich dadurch ein späteres Debuggen erheblich. Der *event*-Parameter wird von jQuery implizit übergeben.

Zeile 6: Schreibe in dieses Item mittels der jQuery-Methode *val()* den folgenden Wert:

Zeile 7: bisheriger Wert des Elements (lesende Version der Methode), getrimmt und umgewandelt in GROßSCHREIBUNG. Der Befehl *toUpperCase()* ist klassisches JavaScript, *trim()* hingegen ist eine statische Methode aus der jQuery-Bibliothek).

> Exkurs: Der Begriff GROßSCHREIBUNG, dessen kapitale Variante laut Duden eigentlich GROSSSCHREIBUNG lauten muss (es gibt kein großes ß), lädt zu weiteren Überlegungen ein: Wie wäre es hierfür mit einer Zeichenersetzung? Ein Blick in die JavaScript-Referenz zeigt: Ja, es existiert eine replace()-Methode auf String-Objekten. Nichts Böses ahnend könnte man nun ein *$me.val().replace('ß', 'ss')* einbauen, was auch zunächst scheinbar einwandfrei funktionieren wird. Ein beliebter Fallstrick. Unterschiedliche Programmiersprachen bieten analoge Befehle mit gleichartiger Signatur an, die jedoch im Detail anders „ticken“: Den Architekten von JavaScript hat es seinerzeit gefallen, *replace()* in der einfachen Verwendung nur das erste Vorkommen des Suchstrings finden zu lassen. „1ß 2ß “ wird also zu „1ss 2ß“. Und die Lösung? Man bemühe stattdessen einen regulären Ausdruck:
> *$me.val().replace(/ß/g, 'ss').*

Möchte man beim Lesen und Schreiben von Item-Werten lieber Funktionen verwenden, die im APEX-JavaScript-API definiert sind, können die Zeilen 5 bis 8 auch wie folgt umgeschrieben werden:

```
5           var me = event.target; // DOM-Node, nicht jQuery-Objekt
6           // APEX verwendet die Befehle $s und $v anstatt val():
7           $s(me, jQuery.trim($v(me)).toUpperCase());
8           // (siehe Hilfe > API Reference > JavaScript APIs)
```

Durch den Aufruf von Methoden des APEX-JavaScript-API in den eigenen jQuery-Skripten lassen sich zwar recht schlanke „hybride“ Skripte schreiben, jedoch sollte man wissen, dass sich das API wiederum auf die selben jQuery-Methoden abstützt, die wir in der ersten Version des Skripts explizit verwendet haben. Ein Vergleich der Funktionsschritte (hier: *val* gegenüber *$s*) mit einem Step-into-Debugger kann im Einzelfall eine Entscheidungshilfe sein.

Stellt sich das Skript als brauchbar und zuverlässig heraus, so dass man es gern auf anderen Seiten einsetzen möchte, so empfiehlt sich selbstredend die Zentralisierung des Codes in Form einer HTML-

mäßig referenzierten JavaScript-Datei. Dabei stellt man hoffentlich fest, dass der optionale Parameter (`'#region1',` der Dokumentkontext, auf den die Suche beschränkt wird) nun eher kontraproduktiv wirkt. Wir sollten im Funktions-Body keinesfalls festlegen, wo genau sich die Textfelder der Klasse *uppercase* befinden müssen. Demzufolge wird man die Angelegenheit generischer formulieren:

```
/* Aufruf-Beispiel:
 * jQuery(document).ready(function(){
 *   initUppercaseItems('input.uppercase');
 *   // bzw. jeder andere zutreffende CSS-Selektor:
 *   initUppercaseItems('#region1 #P9060_TEXT_1');
 * });
 */
function initUppercaseItems (uppercaseItems) {
  jQuery(uppercaseItems)       // das können beliebig viele Objekte sein
  .bind('keyup paste change',  // bei keyup kann man den Effekt sehen
    function(event){
      var $me = jQuery(event.target);
      $me.val(jQuery.trim($me.val()).toUpperCase());
  });
});
```

Eine Anmerkung zur Verwendung des Tokens „$“: Dieser Character hat in kanonischem JavaScript keine besondere Bedeutung. Er ist ein legaler Bezeichner wie jeder andere auch; allerdings hat es sich in praktisch allen modernen JavaScript-Frameworks eingebürgert, API-spezifischen Sprachelementen und Objekten ein *$* voranzustellen. In jQuery bezeichnet die alleinstehende Kurzform *$()* sogar das jQuery-Objekt an sich. Beides kann wahlweise verwendet werden:

```
$(document).ready(function() { // ... });
jQuery(document).ready(function() { // ... });  // synonym
```

Weil APEX das $-Zeichen jedoch bereits für seine JavaScript/jQuery-API in Beschlag nimmt, empfiehlt sich in den eigenen jQuery-Skripten, dieses Token nicht mehr zu verwenden und stattdessen „jQuery“ zu schreiben[5]. Dann wird auch deutlich, ob es sich um ein Skript von APEX oder vom Entwickler handelt.

Workarounds, wenn APEX schwächelt

Leider gelingt es nicht immer, genau dem Element, das man manipulieren möchte, ein *id*-Attribut oder eine Klasse zuzuweisen. Denn APEX allein bestimmt die Generierung des HTML-Codes aus den Angaben, die der Entwickler macht. Und APEX wird je nach Item-Typ auch Konstrukte erzeugen, in denen die ID nicht dem Input-Element selbst, sondern dem umgebenden HTML-Bereich zugewiesen wird. Bei der Kalender-Komponente erhält beispielsweise nicht das Eingabefeld die ihm zugewiesene statische ID, sondern die umgebende *table*. In einem interaktiven Reports ignoriert APEX die HTML-Form-Elementattribute. Es ist daher ratsam, zunächst mit einem Werkzeug wie Firebug den von

[5] siehe hierzu den Befehl jQuery.noConflict()

APEX generierten HTML-Code zu studieren, um festzustellen, wo genau (oder ob überhaupt) die ID oder das Klassenattribut eigentlich „gelandet“ ist.

Aber auch dann kann man sich auf jQuery verlassen: Hauptsache, das Element lässt sich „irgendwie“ identifizieren, dann wird es von jQuery auch zuverlässig gefunden und verarbeitet. Zu dieser kühnen Behauptung folgen nun Beispiele, die die Mächtigkeit der jQuery-Selektorsyntax wirklich nur andeuten können. Der Kürze halber wurde hier der übliche Alias *$* für jQuery verwendet:

- ```
  $('.myMRU td[4] input').attr('class', 'i5');
  /* findet alle <input>-Items, die sich jeweils in der fünften
   * (0..4) Tabellenzelle beliebiger DOM-Objekte mit der Klasse
   * "myMRU" befinden, und weist ihnen die Klasse "i5" zu. */
  ```

- ```
  $('div#mySelectlists').find('option:nth-child(2)').remove();
  /* Entfernt die jeweils zweite Zeile (<option>...</option>)
   * sämtlicher Auswahllisten innerhalb des DIV-Bereichs mit
   * der id="mySelectlists". Ein CSS-Selektor zählt 1-basiert */
  ```

- ```
  $('#hideErrors').click(function(){$('.errors').fadeTo('slow',0)});
  /* Sucht das Element mit der id "hideErrors" und weist ihm einen
   * Eventhandler zu, der beim Anklicken sämtliche HTML-Elemente
   * mit der Klasse "errors" auf der Seite findet und sie
   * gemeinsam langsam ausblendet. Dank jQuery: ein Einzeiler!!! */
  ```

- ```
  $('img:not(has(attr[title]))').each(function(){
     $(this).attr('title', $(this).attr('src'));
  });
  /* Iteriert über alle Bilder im HTML-Dokument, die noch kein title-
   * Attribut besitzen und definiert deren Zugriffspfad ('src')
   * als künftiges title-Attribut (worauf beim Überfahren jedes Bildes
   * mit dem Mauszeiger der bekannte Tooltip erscheint und
   * preisgibt, von wo das Bild geladen wurde) */
  ```

- ```
  $('#mru2 input[id^=f02]').bind('focus blur', function() {
     $(this).toggleClass('editing');
  });
  /* Findet innerhalb des HTML-Elements mit der id="mru2" alle
   * <input>-Items, deren id-Attribut mit "f02" beginnt
   * ("f02_0001", "f02_0002", ...) und sorgt dafür, dass beim
   * Betreten und Verlassen das Attribut class="editing" hinzugefügt
   * bzw. entfernt wird, worauf mit CSS-Regeln reagiert werden kann.
  ```

JavaScript-Erfahrene sehen sofort, welchen Mehraufwand es bedeuten würde, diese Aufgaben mit klassischen JavaScript-Befehlen zu lösen. Mit etwas Übung in der jQuery-API sind solche hilfreichen Mini-Programme innerhalb von Sekunden fertig und testbar.
  ```

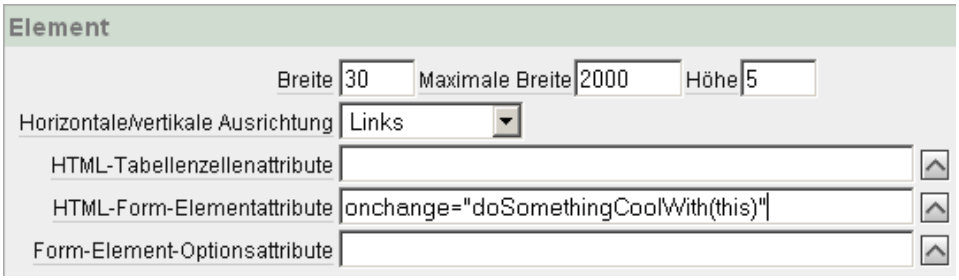
Darüber hinaus sollte man sich vergegenwärtigen, dass jedes dieser Skripte in allen verbreiteten Desktop-Browsern funktionieren wird. In einem von der MT AG durchgeführten APEX-3.2-Projekt mit fast zweihundert Masken, in dessen Verlauf eine jQuery-Bibliothek von weit über eintausend Codezeilen entstanden ist (Kommentare nicht mitgerechnet), ist kein einziger Fall aufgetreten, wo die Seite nach einem erfolgreichen Firefox-Test nicht auch im Internet Explorer funktionierte. Eventuelle Abweichungen waren rein optischer Natur und ließen sich stets auf die unterschiedlichen Implementierungen von CSS zurückführen („ ...sieht aber im IE irgendwie anders aus“) – was sich mit den einschlägigen CSS-Workarounds abstellen ließ.

Einstiegspunkte für eigene Skripte

Im Grunde ist es auch unter APEX allein Sache des Entwicklers, wo er die Ausführungspunkte für seine Skripte platziert, solange er die HTML-Spezifikationen beachtet. Ein Skript, welches das DOM durchläuft („finde alle ... und ändere ...“), ist bestens in der *$(document).ready*-Anweisung aufgehoben, die problemlos auch mehrfach an verschiedenen Stellen verwendet werden darf. Für einen flexibleren, weniger Tippfehler-anfälligen Weg kann man das Template der Seite benutzen:

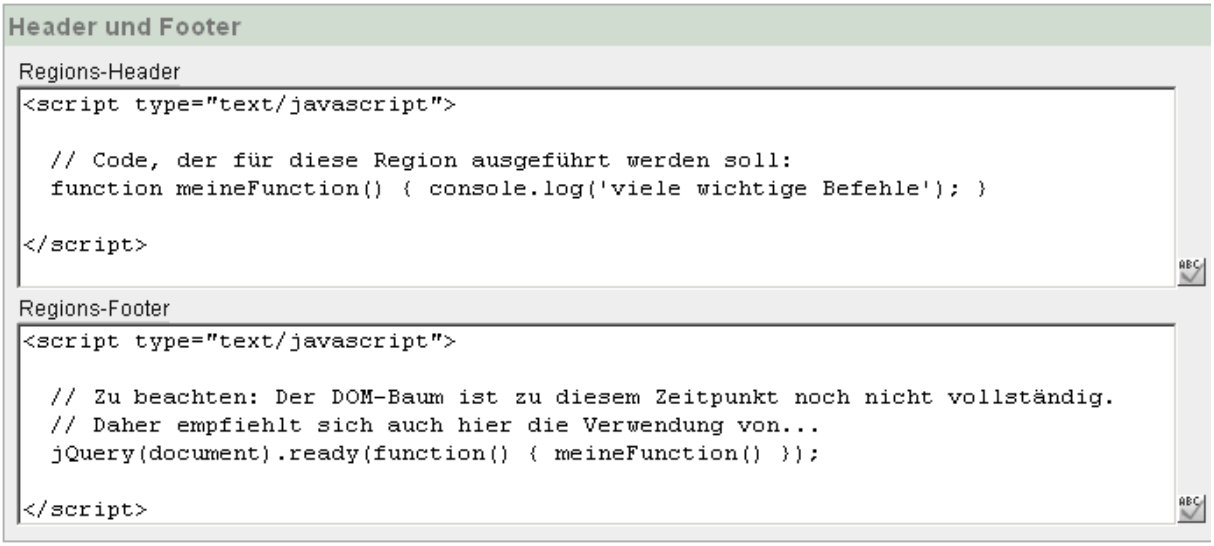
```
<script type="text/javascript">
jQuery(document).ready(function() {
  /* initPage()ausführen, wenn diese Funktion tatsächlich irgendwo im
   * Dokument definiert ist. Andernfalls fehlerfrei nichts tun. */
  if('function' == typeof(initPage)) initPage();
});
</script>
```

Jedes Verhalten, das irgendeinem Element zugeordnet werden soll, wird nun in der Funktion `initPage()` notiert, die man wiederum im HTML-Header der APEX-Seite platziert. Hier kommt der Vorteil „Zentralisierung des Codes“ zum Tragen. Für die berühmten „schnellen Ergebnisse“ ist es – bei sehr einfachen Plänen! – auch denkbar, dem Item selbst ein JavaScript-Eventattribut zuzuordnen:



Diese Herangehensweise ist für komplexere Vorhaben kaum geeignet, insbesondere weil jeweils nur ein einziges Event pro Attribut abgegriffen wird und jQuery mit dem *bind()*- und dem *live()*-Befehl über äußerst flexible Werkzeuge zur Verwaltung von Eventhandlern verfügt.

Weil `<script>`-Tags nicht nur im `<head>`-Abschnitt auftreten dürfen, sondern auch an fast beliebiger Stelle im `<body>`, darf jede APEX-Region ihre eigenen „privaten“ Skripte definieren:



Der Befehl *console.log()* ist übrigens ein guter Freund bei der Arbeit mit dem Dreamteam Firefox&Firebug: Er akzeptiert beliebig viele Parameter und kann, neben Strings, auch ganz hervorragend Objektreferenzen auflösen. Sie wollen ihn aber vor der Veröffentlichung des Projekts bitte wieder sorgfältig entfernen, da er dem Internet Explorer leider völlig unbekannt ist und eine unschöne Fehlermeldung hinterlässt, kurz bevor der IE die Skriptausführung beleidigt abbricht ;-)

Ausblick

In meinem Vortrag auf der DOAG-Konferenz werde ich an dieses Manuskript anknüpfen, um Ihnen weitere fortgeschrittene Beispiele für „clevere Web-Formulare“ live zu zeigen:

- eine clientseitige Validierung, die beim Verlassen eines jeden Items prüft, ob die Eingabe formell korrekt ist, noch bevor das Formular zum Server gesendet wird,
- ein Formular, das von selbst erkennt, ob ungespeicherte Änderungen vorliegen und daraufhin das versehentliche Wechseln zu einer anderen Seite verhindert und,
- falls die Zeit reicht: ein Datumsfeld, das rechnen kann.

Kontaktadresse:

Andreas Wismann
MT AG
Balcke-Dürr-Allee 9
D-40882 Ratingen

Telefon: +49 (0) 2102 - 309 61-0
Fax: +49 (0) 2102 - 309 61-10
E-Mail andreas.wismann@mt-ag.com
Internet: www.mt-ag.com/apex