Spezialitaten bei der AJAX-Unterstutzung in jQuery

Die Unterstützung von AJAX in jQuery ist natürlich unmittelbar am XHR-Objekt und dessen Methoden und Eigenschaften sowie dem grundsätzlichen Ablauf von AJAX-Anfragen orientiert. Wenn Sie verstanden haben, wie AJAX funktioniert, sind viele dieser jQuery-Methoden intuitiv klar.

Die einzelnen Schritte der AJAX-Kommunikation werden jedoch in jQuery innerhalb der verschiedenen AJAX-Methoden meist zusammen durchgeführt. Und es gibt ein paar besondere Features in jQuery zur Unterstützung von AJAX.

JSONP und Remote Requests

In Bezug auf **JSON** stellt **jQuery** eine Erweiterung mit Namen **JSONP** zur Verfügung, um Cross Server Scripting und das flexiblere Nachladen von JSON-Daten von beliebigen URLs zu erlauben. Wie schon besprochen, dürfen AJAX-Anfragen nur an die gleiche Domäne geschickt werden, von der die Webseite geladen wurde. Allerdings ist es schon immer möglich gewesen, JavaScript-Dateien von jedem beliebigen Server direkt in eine Webseite einzubinden, gänzlich unabhängig von AJAX.

Und hier kommt JSONP ins Spiel. Einerseits erzeugt man mit **jQuery** und **JSONP** ein Script und andererseits erwartet man eine angegebene Funktion, um anschließend das enthaltene JSON-Objekt zurückzugeben. Konkret wird bei JSONP ein Skript unter Verwendung der DOM-Repräsentation des **<script>-**Tags per AJAX geladen und ausgeführt.

Bei allen Skripten, die nicht von der gleichen Domain geladen werden, redet man von **Remote Requests**. Wir werden dazu später noch ein konkretes Beispiel sehen.

Das jqXHR-Objekt

Seit jQuery 1.5 stellt das Framework eine spezielle Erweiterung des nativen **XMLHttpRequest**- Objekts zur Verfügung. Es nennt sich **jQuery XMLHttpRequest** oder kurz **jqXHR**. Das Objekt wird etwa von der allgemeinsten jQuery-Methode zur Kontrolle von AJAX-Aktivitäten – **$.ajax()** – als Rückgabewert geliefert. Es stellt einmal alle klassischen **XHRMethoden** und -**Eigenschaften** zur Verfügung, mit einer wichtigen Ausnahme, denn der **onreadystatechange**- Eventhandler wird nicht direkt angeboten.

Das ist insofern kein Problem, da das neue **jqXHR**-Objekt eine neue Schnittstelle bereitstellt, die gezielt einzelne Mechanismen zur Reaktion auf den Erfolg oder Fehler einer AJAX-Anfrage gestattet.

Seine besonderen Fähigkeiten zeigen sich aber vor allen Dingen dann, wenn spezielle Formate mit AJAX nachgeladen werden. Konkret betrifft das im Wesentlichen das <script>-
Tag bei einer JSONP-Anfrage, aber auch XML. In dem Fall simuliert ein jqXHR-Objekt native XHR-Funktionalität, wann immer das möglich ist.

Grundsatzliches zu Methoden in jQuery fur AJAX-Anfragen

Wie schon behandelt, verlaufen AJAX-Requests bzw. -Anfragen immer ähnlich. Im Wesentlichen braucht man den URL der Serverdatei oder des serverseitigen Skripts, die Methode der Datenübertragung,

eventuell zum Server gesendete Daten wie Benutzereingaben, die Information, ob asynchron oder nicht, und eine Callback-Funktion für die Reaktion. Mit diesem Hintergrundwissen sind die Bedeutung und die Anwendung der Methoden jQuery.

jQuery.get(Url, Daten, Callback, Typ)

und

jQuery.post(Url, Daten, Callback, Typ)

samt der Parameter sofort klar.

Beide Methoden liefern Ihnen – wie alle Request-Methoden – ein **XHR-Objekt**. Die Daten sind optionale Wertepaare mit JSON-Format mit Daten, die zum Server gesendet werden sollen. Der optionale Typ sagt aus, ob die Kommunikation asynchron (**true** – der Vorgabewert) oder synchron (**false**) erfolgt.

Die Callback-Funktion wird ausgeführt, wenn die Anfrage erfolgreich beendet wurde. Noch zu besprechen ist eigentlich nur, wie Sie die zurückgegebenen Daten in der Callback-Funktion konkret verwenden können. Aber wenn Sie sich an bisherige Philosophien in jQuery erinnern – die Callback-Methode wird Standardparameter bereitstellen und die repräsentieren die Antwort des Servers. Auch die Methode **jQuery.getJSON(Url, Daten, Callback)** sollte damit im Wesentlichen klar sein.

Richtig verständlich wird die Sache dennoch erst in der Praxis. Wir schauen uns natürlich auf den folgenden Seiten dazu verschiedene Beispiele an.

Angabe der Art der Daten

Grundsätzlich reagieren die AJAX-Methoden intelligent auf die Art der Daten, wie sie vom Server kommen. Das Framework versucht, die Informationen über die Art der Daten sinnvoll zu verwerten, die der Server über den Header liefert. Aber auch die Dateierweiterung kann berücksichtigt werden. Wenn sich auf Grund der Informationen ergibt, dass **XML** geliefert wird, kann man auf den Rückgabewert der AJAX-Methoden ganz normal mit **XMLMethoden** oder jQuery-Selektoren zugreifen. Wenn ein anderer Typ identifiziert wird (etwa HTML), werden die Daten als Text behandelt.

Es kann nun aber vorkommen, dass die Art eines Klartextes vom Server nicht korrekt ausgezeichnet wird. Das ist insbesondere dann der Fall, wenn die Dateierweiterung von Daten auf dem Server nicht mit einem bestimmten Format verbunden ist oder ein unpassender **MIME-Type** gesendet wird. Es handelt sich also oft um eine Einstellung, die auf dem Server vorzunehmen ist. Und darauf hat man selten Einfluss, wenn man nicht gerade den Server selbst betreibt oder aber mit einem Skript den Header setzen kann. Genauso kann es sein, dass Sie gesendete Daten einfach auch anders behandeln wollen.

In beiden Fällen können Sie bei den meisten Methoden in jQuery über den Datentyp explizit angeben, wie die Daten vom Server verstanden werden sollen. In **$.GET()** oder **$.POST()** geben Sie dann beispielsweise entsprechend als letzten Parameter **xml, text, json, jsonp, script oder html** an. Auch in **$.ajax()** können Sie entsprechende Angaben machen.

1. Beim Datentyp **text** und **xml** werden die Daten von den jQuery-Methoden gar nicht verarbeitet, sondern einfach an einen Erfolgs-Handler weitergereicht. Im Hintergrund kommen dazu entweder

responseText(text) oder **responseXML(xml)** zum Einsatz.

2. Im Fall der Auszeichnung **html** sollte jedes eingebettete JavaScript innerhalb der empfangenen Daten ausgeführt werden, bevor das eigentliche HTML als String zurückgegeben wird. Vollkommen analog verhält es sich mit der Auszeichnung **script**, nur wird nach der Ausführung sonst nichts zurückgegeben.

3. Im Fall der Auszeichnung als **json** werden die Daten **geparst** und als JavaScript-Objekt bereitgestellt. Im Hintergrund läuft die Methode **jQuery.parseJSON(),** wenn der Browser das unterstützt. Andernfalls kommt der Konstruktor der JavaScript-Klasse Function zum Einsatz und nicht die sicherheitskritische native **eval()**-Funktion.

4. Bei der Auszeichnung als **jsonp** hängt das Framework einen Querystring-Parameter **callback=?** an den URL, den Sie aber auch manuell an einen URL hängen können. Der Server sollte die JSON-Daten dem Callback-Bezeichner anhängen, um eine valide JSONPAntwort zu schicken. In einigen Fällen kann es auch sinnvoll sein, dass Sie eine nachzuladende Textdatei mit der Erweiterung *.html* versehen (auch wenn das keine vollständige Webseite ist). Insbesondere gilt dies dann, wenn sich in dem Text HTML-Tags befinden, die nach dem Laden im Browser interpretiert werden sollen. Allerdings kann man in den meisten Konstellationen auf diese Dateierweiterung verzichten und konventionell *.txt* verwenden.

Vermeidung von Caching

Bei AJAX-Anfragen gibt es grundsätzlich ein Problem in Bezug auf das Zwischenspeichern (**Caching**) von Daten bei einer **GET**-Anfrage. Es kommt leider nicht selten vor, dass Daten bei einer AJAX-Anfrage vom Server nicht neu angefordert werden, obwohl es notwendig wäre. Der Browser holt die Daten einfach aus dem lokalen Zwischenspeicher (**Cache**). Das betrifft vor allen Dingen den Internet Explorer.

Bei einer händischen Programmierung hängt man als Workaround deshalb zum Beispiel an den Querystring einen zufälligen Wert oder einen **Timestamp** an, der einfach mit JavaScript generiert und im Grunde weder auf dem Server noch im Client später weiterverwertet wird. Er soll dem Browser bloß suggerieren, dass Daten neu geladen werden müssen.

Tatsächlich ist dieser Trick derzeit die einzige einfache und dennoch zuverlässige Vorgehensweise, um dieses (in der klassischen Webdatennachforderung durchaus sinnvolle) Verhalten des **Cachings** auszuschalten. Diejenigen Frameworks, die dafür explizit einen Mechanismus bereitstellen, machen auch im Hintergrund nichts anderes. In jQuery müssen Sie derzeit in so einer Situation bei den meisten Methoden wie beschrieben von Hand einen extra Parameter erstellen und an den URL hängen.

Nur die **ajax()**-Methode bietet eine entsprechende Option, um das Cachen mit einem speziellen Parameter zu verhindern.

Anwendung der Standardmethoden fur AJAX

Widmen wir uns nun den wichtigsten Standardmethoden von jQuery im Umgang mit AJAX.

$.get() und $.post()

Schauen wir uns zuerst die beiden wohl wichtigsten Methoden zur Nachforderung von Daten per AJAX genauer an. Es handelt sich um **$.get()** und **$.post().** Grundsätzlich werden beide Methoden so eingesetzt:

Listing 11.1 Schematische Form der Methoden

```
jQuery.get(
Url, [ Daten], [ success(Daten, textStatus, jqXHR) ], [ Datentyp ]
)

jQuery.post(
Url, [ Daten ], [ success(Daten, textStatus, jqXHR) ], [ Datentyp ]
)
```

Der einzig zwingende Parameter ist der **URL**, wohin die Anfrage gesendet werden soll. Die optionalen Daten sind ein Objekt vom Typ **Map** oder ein **String** mit den Daten, die zum Server gesendet werden sollen.

Der dritte Parameter ist eine **Callback**-**Funktion**, die im Erfolgsfall aufgerufen wird. Deren Parameter stehen für die Daten, die vom Server kommen sollen, einen Text mit dem Status und das **jqXHR**-**Objekt**.

Einfach Klartext vom Webserver anfordern

Betrachten wir die einfachste Anwendung – es soll Klartext vom Webserver angefordert werden. In dem folgenden Beispiel wird das mit zwei Schaltflächen per AJAX gemacht. Wir verwenden parallel **$.get()** und **$.post(),** um beide Methoden zu zeigen. Sie werden sehen, dass es im Quelltext keine Unterschiede gibt. Die Webseite *kap11_1.html* sieht so aus:

Listing 11.2 Die Webseite

```
...
<body>
<h1>AJAX</h1>
<button>Daten mit GET</button>
<button>Daten mit POST</button>

<hr/>

<div id="antwort1" class="k1"></div>
<div id="antwort2" class="k1"></div>

</body>
</html>
```

So sieht die JavaScript-Datei aus *(kap11_1_ready.js)*:

Listing 11.3 AJAX-Anfragen mit GET und POST

```
$(function(){
$("button:first").click(function(){
$.get("ajax.txt", function(data){
$("#antwort1").html(data);
});
});

$("button:eq(1)").click(function(){
$.post("ajax.txt", function(data){
$("#antwort2").text(data);
});
});
});
```

In dem Beispiel fordern wir eine einfache Textdatei ohne irgendwelche Struktur mit AJAX vom Webserver an. Beim Klick auf die erste Schaltfläche wird die Datei mit **GET** angefordert und beim Klick auf die zweite Schaltfläche mit **POST**. Funktional gibt es in diesem einfachen Beispiel wie gesagt nicht den geringsten Unterschied zwischen den beiden Methoden der Datenanforderung.

Nur intern sieht der HTTP-Header etwas anders aus und natürlich verarbeitet der Server die Daten entsprechend der jeweiligen Methode. Daten vom Client werden ansonsten in diesem ersten Beispiel nicht zum Webserver geschickt und deshalb brauchen die Methoden zur AJAX-Datenanforderung jeweils nur zwei Parameter. Der erste Parameter ist natürlich der URL, der zweite Parameter ist der Callback. Hier benutzen wir jeweils eine vollkommen gleich aufgebaute anonyme Callback-Funktion. Der Parameter an die Funktion enthält die Antwort des Webservers. Einfacher geht es kaum. Die Antwort des Webservers wird in diesem ersten Beispiel ohne weitere Aktionen direkt in die Webseite eingebaut.

Nun beinhaltet die angeforderte Textdatei eine HTML-Steueranweisung **(<u>).** Wenn Sie solche Daten anfordern, die mit HTML-Fragmenten durchsetzt sind, werden diese HTML-Fragmente nicht interpretiert, solange Sie mit der Methode **text()** arbeiten. Wir machen das bei der Anforderung per **POST**. Aber statt der Methode **text()** können Sie natürlich ebenso die Methode **html()** verwenden und dann wird die Antwort vom Browser entsprechend interpretiert.

Wir machen das bei der Anforderung über die erste Schaltfläche. In der Praxis ist es oft üblich, solche HTML-Fragmente zu schicken und diese einfach in einer Webseite einzubauen. Beachten Sie, dass Sie – bis auf wenige Ausnahmen – auf keinen Fall eine vollständige HTML-Seite schicken dürfen. Eine solche HTML-Seite ist ja bereits im Browser geladen und es sollen nur kleine Bestandteile aus dieser Seite ausgetauscht werden.

HINWEIS: *Eine AJAX-Anfrage setzt so gut wie immer voraus, dass Sie mit einem Webserver arbeiten. Sie wollen ja Daten von einem Webserver nachfordern und in die bestehende Webseite integrieren. Dazu laden Sie die Daten per HTTP nach. Sie müssen für einen realistischen Test sowohl die Webseite als auch die nachgeforderten Daten über einen Webserver in den Browser laden.*

Ein einfaches Öffnen der Webseite vom Dateisystem ist nicht sinnvoll. Sobald Sie also Ihren lokalen Webserver gestartet haben, ist er in der Regel über den URL ***localhost*** *verfügbar. Andernfalls geben Sie in der Adresszeile einfach die IP-Nummer oder den Namen des Rechners ein, auf dem der Webserver läuft.*

Daten per $.get() und $.post() zum Webserver senden

Schauen wir uns nun an, wie bei der AJAX-Anfrage Daten zum Webserver geschickt werden. Das werden im Wesentlichen Benutzereingaben sein, entweder Daten, die ein Anwender in einem Webformular einträgt oder Daten auf Grund von Aktionen, die der Anwender mit der Maus durchführt. Sie werden sehen, dass jQuery das Versenden durch Objekte vom Typ Map erheblich vereinfacht sowie vereinheitlicht.

PRAXISTIPP: *Wenn Sie Daten ohne jQuery zum Server senden wollen, wird man meist die Daten per* ***GET*** *versenden. Sollte* ***POST*** *zum Einsatz kommen, müssen die gesendeten Daten erst einmal speziell kodiert werden, was man manuell über das Setzen von einem Headerfeld machen muss.*

Wenn per POST bei jQuery ein Querystring gesendet werden soll, kodiert das Framework die Daten entweder automatisch im Hintergrund oder Sie können die Daten mit einer Methode ***jQuery.param()*** *in eine passende Form umwandeln, die auch im Hintergrund bei der automatischen Umwandlung zum Einsatz kommt. Soll das nicht erfolgen, kann man einen Parameter* ***processData*** *auf* ***false*** *setzen.*

Das kann dann sinnvoll sein, wenn Sie ein XML-Objekt zum Server senden wollen. In dem Fall sollte die ***contentType****-Option von* ***application/x-www-form-urlencoded*** *auf einen besser passenden MIME-Type geändert werden.*

Die vom Client zum Webserver gesendeten Daten müssen dort auch verarbeitet werden. Dann macht es keinen Sinn mehr, dass Sie eine reine Textdatei auf dem Server als URL angeben. Sie müssen auf dem Server ein Skript oder ein Programm aufrufen, das die Daten entgegennimmt und verarbeitet. Dieses kann als Antwort an den Client natürlich wieder reinen Klartext generieren oder auch eine beliebige strukturierte Antwort. Das spielt keine Rolle. Die Serverseite werden wir hier jedoch nicht weiter betrachten (das ist ausdrücklich nicht unser Thema), sondern wir senden einem Skript, das für uns eine Blackbox darstellt, die Daten.

Dann lassen Sie uns ansehen, wie die Webseite mit der AJAX-Funktionalität aussieht. Für dieses Beispiel benötigen wir eine etwas andere Struktur als in den meisten anderen Beispielen in dem Kapitel (kap11_2.html):

Listing 11.4 Eine Webseite zur Verifizierung von Benutzereingaben per AJAX

```
...
<body>
<h1>Bitte Benutzername und Passwort eingeben</h1>
<form>
<table><tr><td>Name</td><td><input type="text" size="30"></td></tr>
<tr><td>Passwort</td><td><input type="password"
size="30"></td></tr>
</table></form>
<button>AJAX-Login mit $.get()</button>
<button>AJAX-Login mit $.post()</button>
<div id="ausgabe"></div>
</body>
</html>
```

Sie sehen in dem Beispiel eine einfache Tabelle mit zwei Eingabefeldern, die ein wenig mit CSS formatiert wird. In das eine Feld wird der Anwender seine Userid und in das andere Feld sein Passwort eingeben. Da sich die Versendung der Daten per **GET** bzw. **POST** nicht unterscheiden, sehen Sie gleich zwei Schaltflächen, über die der Anwender die Daten abschicken kann. Das spart uns zwei im Grunde identische Beispiele für GET und POST.

Die AJAX-Funktionen sehen weitgehend so aus wie in dem letzten Beispiel. Der einzige Unterschied ist die Erweiterung um den Parameter mit den eingegebenen Benutzerdaten *(kap11_2_ready.js)*:

Listing 11.5 Das Versenden der Daten

```
$(function() {
 $("button:first").click(function() {

  $.get("kap11_2_get.php", {
   username : $("input:first").val(),
   password : $("input:last").val()
  },

  function(data) {
    $("#ausgabe").html(data);
  });
});

$("button:last").click(function() {
 $.post("kap11_2_post.php", {
  username : $("input:first").val(),
  password : $("input:last").val()
 },

  function(data) {
   $("#ausgabe").html(data);
  });
 });
});
```

Als zweiten Parameter sehen Sie jeweils die Wertepaare, die an den Server geschickt werden. Die Wertepaare werden wie in **jQuery** üblich in geschweifte Klammern notiert, also ein besagtes Objekt vom Typ **Map**. Wie üblich greifen wir mit einem Selektor auf die Eingabefelder zu. Die Methode **val()** liefert uns den Inhalt in dem jeweiligen Eingabefeld. Da wir für ein Wertepaar auch einen Namen benötigen, um die übergebenen Werte auf Serverseite abfragen und zuordnen zu können, werden die Token **username** und **password** jeweils vorangestellt.

Wenn nun ein Anwender in den Eingabefeldern die korrekten Zugangsdaten einträgt, wird die Antwort des Webservers ohne Neuladen der Webseite unterhalb des Formulars angezeigt.

XML-Daten anfordern und verarbeiten

Es ist kein großes Problem, wenn Sie per AJAX Klartext mit XML-Struktur anfordern wollen.

Sie müssen bloß einen entsprechenden URL auf eine XML-Datei setzen oder mit einem Skript oder Programm auf dem Server XML-Code generieren. Spannend wird es nur im Client. Denn da steht Ihnen als Antwort ein vollständiger DOM zur Verfügung, wenn Sie die XML-Daten mit **responseXML** entgegennehmen! In den jQuery-Methoden steht dieser DOM über den Parameter der Callback-Funktion zur Verfügung.

Wenn Sie sich das Beispiel *kap11_3.html* von der Webseite zum Buch laden, werden Sie sehen, dass wir dort eine Schaltfläche für GET und eine für POST und einen Ausgabebereich für beide Anforderungen haben. Interessant ist wieder die JavaScript-Datei *(kap11_3_ready.js)*:

Listing 11.6 Anfordern einer XML-Datei

```
$(function() {
$("button:first").click(function() {
$.get("ajax.xml", function(data) {
$("#ausgabe").text("");
for (i in data) {
$("#ausgabe").append(i + ": " + data[i] + "<br />");
}
});
});
$("button:eq(1)").click(function() {
$.post("ajax.xml", function(data) {
$("#ausgabe").text("");
$("#ausgabe").append(data.getElementsByTagName("name")[0]);
$("#ausgabe").append("<hr />");
y = data.getElementsByTagName("url");

for ( i = 0; i < y.length; i++) {
 $("#ausgabe").append(y[i].childNodes[0].nodeValue + "<br >");
}
});
});
});
```

Und das soll unsere angeforderte XML-Datei sein:

Listing 11.7 Die XML-Datei, die per AJAX angefordert wird

```
<?xml version="1.0" encoding="UTF-8"?>
 <rjs>
  <daten>
   <name beruf="Dipl Math">Ralph Steyer</name><ort>Bodenheim</ort>
  </daten>
  <webseiten>
   <url>{ HYPERLINK }>
   <url>www.autoren-net.de</url>
  </webseiten>
</rjs>
```

Wir fordern einfach eine beliebige XML-Datei an. Bei der Anforderung per **GET1** beim Klick auf die erste Schaltfläche iterieren wir in der **for**-**Schleife** jeweils über alle Elemente, die in der Antwort des Webservers (**data**) zur Verfügung stehen. Wenn Sie die Ausgabe beachten, erkennen Sie, dass die Antwort ein vollständiges **DOM**-**Objekt** ist. Damit sind alle üblichen Eigenschaften und Methoden verfügbar, die Sie auch in dem DOM einer Webseite verwenden können, und auch die jQuery-Spezialitäten.

Nun können Sie also prinzipiell den DOM der XML-Daten im Client weiterverwerten, aber das ist – wie schon angedeutet – gar nicht mal in so vielen Fällen sinnvoll, da es mit JSON eine einfachere Alternative gibt. Dennoch soll nachfolgend kurz angedeutet werden, wie Sie vorgehen können.

Grundsätzlich können Sie zur Navigation auf der Antwort des Webservers mit den üblichen Methoden **getElementById()**, **getElementsByName()**, **getElementsByTagName()**, **getElementsByTagNameNS()** arbeiten oder natürlich auch alle jQuery-Techniken zum Selektieren, Traversieren und Navigieren auf einem Baum verwenden. Dies gestattet zusätzlich in Hinsicht auf die Datennachforderung per AJAX eine sehr qualifizierte Ansteuerung beliebiger Teile der Antwort.

Nur müssen Sie gewaltig aufpassen, da der DOM in verschiedenen Browsern wie mehrfach erwähnt abweichend aufgebaut wird. Der entscheidende Punkt ist immer, dass Sie zur Auswahl eines Elements mit einer geeigneten Methode bereits möglichst nahe an den Zielknoten herankommen. Wenn Sie von der Wurzel der Antwort aus dem Baum in die Tiefe traversieren, werden Sie in verschiedenen Browsern möglicherweise abweichende Ergebnisse erhalten.

In dem Beispiel verwenden wir beim Klick auf den zweiten Button die Methode **getElementsByTagName(),** um auf der Antwort des Servers (**data)** gezielt Elemente auszuwählen.

Zuerst wählen wir das Element name aus und geben das direkt in der Webseite aus, indem wir es per **append()** an das Div-Element hängen. Es ist bemerkenswert, dass dies in der Situation den Inhalt des Elements liefert2, denn eigentlich muss man den untergeordneten Textknoten und dessen Inhalt ansprechen. Das machen wir im nächsten Schritt, um explizit den Inhalt eines Textknotens zu selektieren.

Diese Methode **getElementsByTagName()** liefert ein Array zurück und deshalb können wir mit **length** die Anzahl der enthaltenen Elemente abfragen und über Element-Array iterieren. Das nutzen wir für das url-Element in der XML-Antwort.

Nun ist der Textinhalt von einem Element wie gesagt im DOM-Konzept selbst ein Knoten und deshalb müssen wir das erste **Kindelement** nehmen, wenn wir diesen Inhalt abfragen wollen. Ein Textknoten stellt u. a. über **nodeValue** diesen Inhalt zur Verfügung.

Wenn Sie nun noch beachten, dass Sie über **nodeType** die verschiedenen Elementtypen unterscheiden können und Ihnen alle Navigationsmöglichkeiten und Filter des DOM-Konzepts sowie von jQuery zur Verfügung stehen, sollte klar sein, wie der prinzipielle Weg zur gezielten Verwertung der XML-Antwort aussieht.

Nun ist es definitiv noch einfacher, den XML-DOM zu verwerten, wenn Sie ihn in den jQuery-Namensraum holen – einfach darüber, dass Sie über **$(data)** zugreifen. Dann können Sie auf diesem jQuery-Objekt mit den üblichen Selektoren arbeiten. Aber wie gesagt, das ist und bleibt dennoch ein mühseliges Geschäft.

JSON-Daten anfordern und verarbeiten – getJSON() und parseJSON()

Wie erwähnt, bietet **JSON** fast die gleiche Funktionalität und Flexibilität wie **XML**, ohne dessen Komplexität und problematische Auswertung in verschiedenen Browsern in den Weg zu stellen. Aus diesem Grund hat sich JSON als Standardformat bei AJAX-Applikationen durchgesetzt, wenn tatsächlich eine gewisse Funktionalität zur Auswertung der Antwort im Client gefordert wird. Nun können Sie im Prinzip mit **$.get()** oder **$.post()** JSON-Daten vom Server anfordern.

Nur dann müssen Sie diese in der Regel erst einmal weiterverarbeiten, um diese zu verwenden. Dazu gäbe es beispielsweise eine native JavaScript- Funktion **eval(),** um aus JSON ein JavaScript-Objekt zu machen. Aber diese Methode ist sehr langsam, zwingt Browser, ihren Just-in-Time-Compiler abzuschalten, und gilt vor allen Dingen als extrem unsicher. Dann gibt es aber auch in aktuellen Browsern **JSON.parse()** in purem JavaScript, was sicherer und auf JSON direkt gemünzt ist. Alternativ kann man ebenfalls mit der nativen JavaScript-Klasse **Function** arbeiten oder aber die Methode **jQuery**.

parseJSON() verwenden, die in jQuery 1.4.1 eingeführt wurde und einen JSON-String in ein JavaScript-Objekt umwandelt (sofern der Browser das unterstützt).

Aber im Fall einer AJAX-Anforderung ist so ein Aufwand gar nicht einmal notwendig. Die auf dieses Datenformat spezialisierte AJAX-Methode **getJSON()** aus dem jQuery-Framework macht den Umgang mit JSON fast zum Kinderspiel.

Eine einfache Anwendung mit JSON

Wir wollen als Basis die folgende JSON-Struktur verwenden:

Listing 11.8 Die JSON-Datei, die per AJAX angefordert werden soll

```
{
"name": "Ralph Steyer",
"webseiten": ["www.rjs.de", "blog.rjs.de", "www.autoren-net.de"]
}
```

Die Details, wie die JSON-Struktur auf dem Server bereitgestellt wird, sind für uns wieder uninteressant. Die Webseite *kap11_4.html* soll wie die vorherige aufgebaut sein. Wir betrachten nur die Auswertung auf dem Client. Beispiel *(kap11_4_ready.js)*:

Listing 11.9 Anfordern und Auswerten von JSON

```
$(function() {
 $("button:first").click(function() {
  $("#ausgabe").text("");
  $.getJSON("ajax.json", function(data) {
  $("#ausgabe").append(data.name + ", " + data.webseiten[1]);
 });
});
```

```
$("button:eq(1)").click(function() {
 $.get("ajax.json", function(data) {
  $("#ausgabe").text("");
  for (i in data) {
   $("#ausgabe").append(i + ": " + data[i] + "<br />");
  }
  });
 });
});
```

Wir fordern hier Daten in Form von JSON an und die Antwort des Webservers steht wieder über **data** (den ersten Parameter der **Callback**-**Funktion**) zur Verfügung. Die extrem komfortable Situation ist nun, dass die Methode **getJSON()** die Daten in einer solchen Form liefert, dass wir über einfache Punktnotation auf die Namen der einzelnen Elemente in der JSON-Struktur zugreifen können. Alternativ nehmen wir die Array-Notation, natürlich auch verschachtelt, was bei unserer Beispielstruktur genutzt wird. Das ist einfach und zuverlässig.