HANSER

Leseprobe

zu

„JavaScript“

von Ralph Steyer

ISBN (Buch): 978-3-446-43942-9
ISBN (E-Book): 978-3-446-43947-4

Weitere Informationen und Bestellungen unter
http://www.hanser-fachbuch.de/978-3-446-43942-9
sowie im Buchhandel

2 Grundlagen und erste Beispiele

Wir beginnen mit der Erstellung von Webseiten und der JavaScript-Programmierung. Dazu möchte ich Ihnen zu Beginn dieses Kapitels direkt ein paar einfache JavaScript-Beispiele vorstellen. Bevor wir uns dann jedoch in die Details einarbeiten und zuerst auf die konkrete Erstellung von Webseiten stürzen, schaffen wir uns in diesem Kapitel ein paar weitere Grundlagen, damit die nachfolgenden Schritte leichter verständlich werden. Es erleichtert viele Folgeschritte, wenn wir bei einigen Fachbegriffen von den gleichen Voraussetzungen ausgehen. Sie lernen in diesem Kapitel etwas über:

- JavaScript mit ersten kleinen Skripten,
- das Internet im Allgemeinen,
- die Idee des Programmierens und die spezielle Situation beim Programmieren im Internet und
- die Funktion eines Browsers.

2.1 Erste JavaScript-Beispiele

Sie wollen sicher JavaScript möglichst schnell in der Praxis erleben und sehen, was man damit machen kann. Was eignet sich besser, als selbst ein paar kleine Beispiele einzugeben und auszuprobieren? Wir springen mit dieser Aufgabe direkt ins kalte Wasser. Dabei kalkuliere ich ein, dass Sie unter Umständen nicht so genau nachvollziehen können, was hier getan wird. Das klärt sich im Laufe des Buchs. Geben Sie nur den Quelltext exakt wie vorgegeben ein (mit Beachtung der Groß- und Kleinschreibung, Klammern, Semikolon etc.). Wichtig ist hier, dass Sie den Umgang mit einem Editor und der grundsätzlichen Erstellung von Quelltext sowie einem Browser üben und dabei erste Effekte sehen, die auf JavaScript beruhen.

2.1.1 Ein einfaches Mitteilungsfenster

Geben Sie den nachfolgenden Quelltext in einem Editor ein und nennen Sie die Datei *kap2_1.html*. Nun möchte ich bereits mit den ersten einfachen Beispielen einen Weg gehen,

der bei umfangreicheren Beispielen unabdingbar wird. Wir wollen mit einer Projektstruktur arbeiten, was erst einmal hauptsächlich die Strukturierung in Verzeichnissen bedeutet. Dazu möchte ich Ihnen den Einsatz einer IDE anhand von Aptana (unserer Referenz-IDE) erklären[1]. Wenn Sie also mit Aptana arbeiten, verwendet diese IDE einen sogenannten **Workspace**. Das ist ein verwaltetes Verzeichnis, in dem Sie alle Projektverzeichnisse speichern. Innerhalb des Workspace erzeugen Sie in Aptana ein neues Webprojekt mit Namen *kap2* (**File → New → Web Project**).

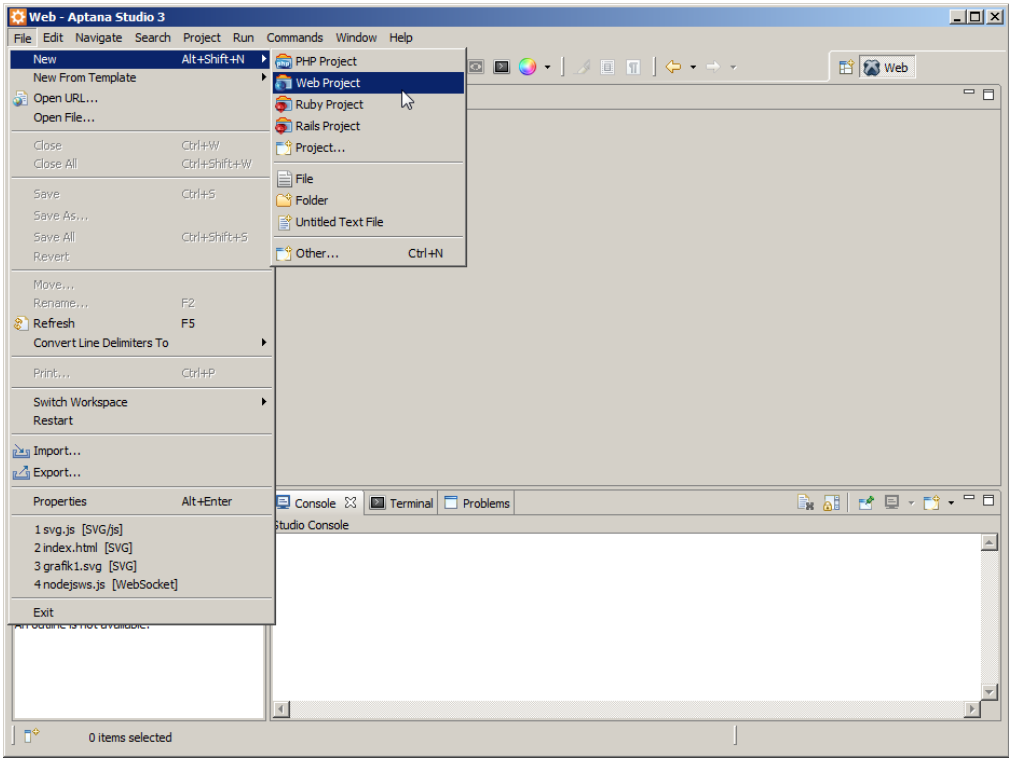

Bild 2.1 Anlegen eines neuen Webprojekts mit Aptana

Wählen Sie die Vorgabevorlage für das Projekt aus und vergeben Sie im folgenden Dialog den Projektnamen. Wir erzeugen in dem Projekt nun eine Datei *kap2_1.html*. Das geht am besten mit **File → New From Template → HTML**. Für unsere Zwecke genügt eine Webseite nach dem HTML4-Standard.

Da Aptana einen eigenen Webserver mitbringt, können Sie das Webprojekt an einer beliebigen Stelle speichern, solange es unter der Verwaltung von Aptana bekannt ist.

[1] Das soll Sie nicht daran hindern, dass Sie Alternativen wie das Visual Studio oder nur einen einfachen Editor einsetzen, wenn Sie sich damit auskennen oder Ihnen diese Programme genehmer sind.

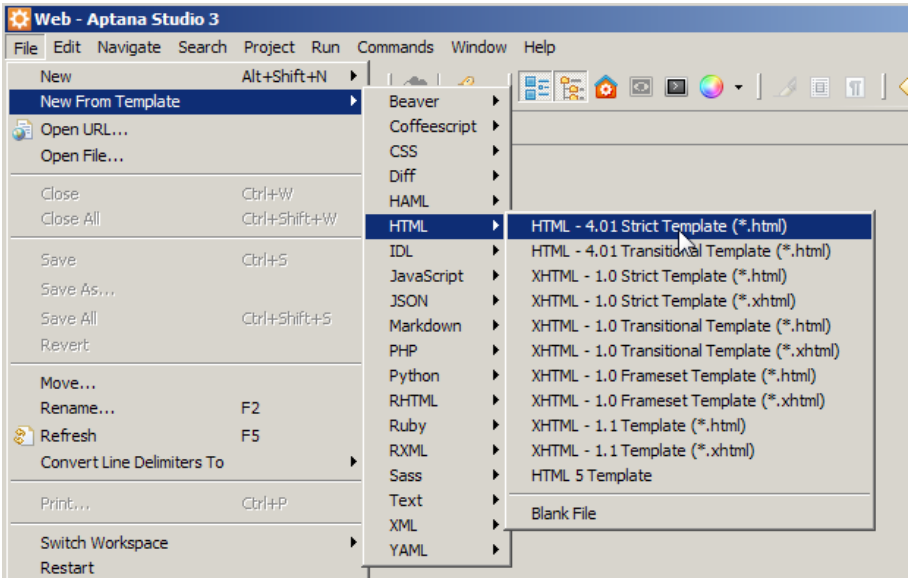

Bild 2.2 Anlegen einer HTML-Seite

Sie müssen natürlich nicht den Workspace von Aptana oder überhaupt die Projektverwaltung von Aptana verwenden, sondern Sie können auch eine andere Basisstruktur wählen, gerade wenn Sie – wie im letzten Kapitel beschrieben – einen Webserver zur Verfügung haben und diesen verwenden wollen. In dem Fall speichern Sie diese Datei (und auch alle anderen Beispieldateien in diesem Buch) in dem Verzeichnis, das über Ihren Webserver freigegeben ist. Im Fall einer XAMPP-Installation ist dies das Verzeichnis *htdocs*. Legen Sie dort am besten ein Unterverzeichnis mit Namen *kap2* an. Dort hinein soll diese Datei gespeichert werden.

HINWEIS: Die Beispieldateien der nächsten Kapitel sollten dementsprechend in analogen Unterverzeichnissen auf dem Webserver oder dem Workspace von Aptana abgelegt werden. Sie können auch den Workspace von Aptana so konfigurieren, dass das *htdocs*-Verzeichnis von Apache das Workspace-Verzeichnis darstellt.

Doch kommen wir nun zum Quelltext. Dies soll der Inhalt der Quelltextdatei *kap2_1.html* sein:

Listing 2.1 Das erste JavaScript-Beispiel

```
<html>
  <body>
    <script>
      alert("Hallo Welt");
    </script>
  </body>
</html>
```

In den Zeilen 3 bis 5 der Webseite finden Sie einen stark vereinfachten Skript-Container und die eigentliche JavaScript-Anweisung steht in der vierten Zeile. Rufen Sie die Datei mit Ihrem Standardbrowser über den Webserver auf. Falls der Webserver auf dem gleichen Rechner läuft wie Ihr Webbrowser, würde der Aufruf wie folgt lauten, wenn Sie sich an meine Empfehlung bezüglich der Wahl eines Verzeichnisses gehalten haben:

http://localhost/kap2/kap2_1.html

In Aptana können Sie die Datei auch direkt aus der IDE ausführen. Dazu müssen Sie die Datei bloß selektieren und im Menü den Befehl **Run → Run as... → JavaScript Web Application** aufrufen. Aptana startet dann seinen integrierten Webserver und stellt die Webseite darüber in dem zugeordneten Standardbrowser dar. Alternativ können Sie auch mit dem Kontextmenü arbeiten. Ihr Browser sollte in jedem Fall ein kleines Dialogfenster mit dem Text *Hallo Welt* anzeigen.

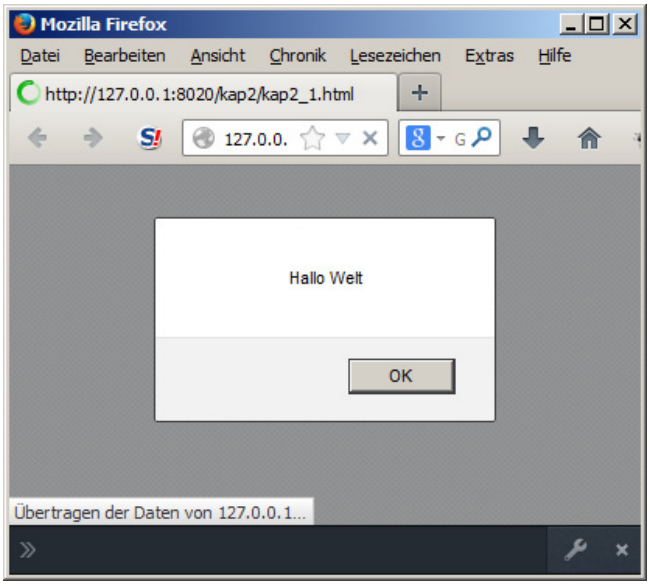

Bild 2.3 So oder so ähnlich sollte unser erstes Beispiel aussehen.

Troubleshooting

Sollte das Beispiel nicht funktionieren, überprüfen Sie zuerst Ihren Quelltext. Achten Sie vor allen Dingen auf Klammern, Hochkommata, Semikolon sowie Groß- und Kleinschreibung.

Stimmt der Quellcode exakt mit der Vorgabe überein und das Beispiel funktioniert dennoch nicht, wird eventuell JavaScript in Ihrem Browser deaktiviert sein. Dann müssen Sie JavaScript in den Einstellungen Ihres Browsers aktivieren. Wo das genau gemacht wird, ist von Browser zu Browser unterschiedlich. Mittlerweile ist auch die Tendenz in Browsern zu beobachten, dass Anwender JavaScript gar nicht mehr direkt deaktivieren können. Von daher ist die potenzielle Fehlerquelle auch nicht sonderlich wahrscheinlich. Aber eventuell haben Sie auch in Firefox das Add-on *NoScript* aktiviert. Dann müssen Sie ggf. die Ausfüh-

rung für das Skript noch erlauben (das kann in der Regel individuell für jede geladene Datei ausgewählt werden – wenn Sie die Ausführung individuell zulassen, schwächen Sie Ihren allgemeinen Schutz des Browsers damit nicht).

Sollte nach Aktivierung der JavaScript-Unterstützung das Beispiel (scheinbar) immer noch nicht funktionieren, kann es daran liegen, dass das Dialogfenster geblockt wird. Auf Grund diverser Sicherheitsprobleme im WWW verhindern einige (vor allem neuere) Browser sogenannte aktive Inhalte. Sollten Sie eine entsprechende Warnung bekommen, müssen Sie zur Ausführung des Beispiels die aktiven Inhalte für dieses Beispiel zulassen.

HINTERGRUNDINFORMATION: Der Zugriff über *http://localhost* wird in der Regel (bei den meisten Browsereinstellungen) als sicher angesehen, der Zugriff über das file-Protokoll hingegen oft nicht. Wenn Sie die Sicherheitseinstellungen Ihres Browsers nicht schwächen wollen, Sie die ständigen Warnmeldungen aber nerven, haben Sie hier ein Argument mehr, warum Sie auch bei einem lokalen Laden einer Webseite den Zugriffsweg über den Webserver wählen sollten.

2.1.2 Schreiben eines angepassten Aktualisierungsdatums

Spielen wir ein weiteres Beispiel durch, in dem das Aktualisierungsdatum der Webseite dynamisch mit JavaScript geschrieben wird. Geben Sie den nachfolgenden Quelltext in einem Editor ein (wieder ohne die vorangestellten Zahlen):

Listing 2.2 Das zweite JavaScript-Beispiel

```
<html>
  <body>
    Die Webseite wurde zuletzt am
    <script>
      document.write(new Date());
    </script>
    aktualisiert.
  </body>
</html>
```

Speichern Sie die Datei unter dem Namen *kap2_2.html*. In den Zeilen 4 bis 6 finden Sie wieder einen Skript-Container und die eigentliche JavaScript-Anweisung steht in der fünften Zeile. Beachten Sie, dass sowohl vor dem Skript-Container als auch danach reiner Text steht, der nicht zum Skript zählt. Hier sind wir noch in der HTML-Welt. Öffnen Sie die Datei in Ihrem Standardbrowser, um zu sehen, was der Quellcode bewirkt. Im Anzeigebereich des Browsers sollte ein Text zu sehen sein, der aus den statischen Textpassagen und einer dynamisch per JavaScript generierten Information (dem Tagesdatum mit einer Genauigkeit eines Bruchteils einer Sekunde) besteht.

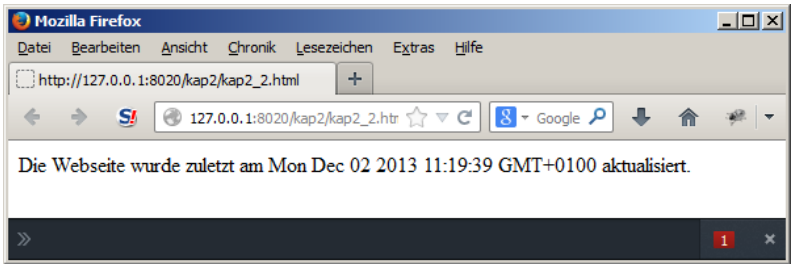

Bild 2.4 Statische und dynamisch generierte Information kann ein Anwender nicht unterscheiden.

2.1.3 Entgegennahme einer Benutzereingabe

Geben Sie für ein abschließendes Einstiegsbeispiel den nachfolgenden Quelltext in einem Editor ein und speichern Sie diesen unter dem Namen *kap2_3.html:*

Listing 2.3 Entgegennahme einer Benutzereingabe

```
<html>
   <body>
     Das ist die Webseite von
     <script>
       document.write(prompt("Ihr Name"));
     </script>
     .
 </body>
 </html>
```

In den Zeilen 4 bis 6 finden Sie wieder den Skript-Container und die eigentliche JavaScript-Anweisung steht in der fünften Zeile. Auch in diesem Beispiel steht sowohl vor dem Skript-Container als auch danach reiner Text, der nicht zum Skript zählt. Beim Öffnen der Datei in einem Browser sollten Sie ein kleines Eingabefenster angezeigt bekommen, in dem Sie Ihren Namen eingeben sollen.

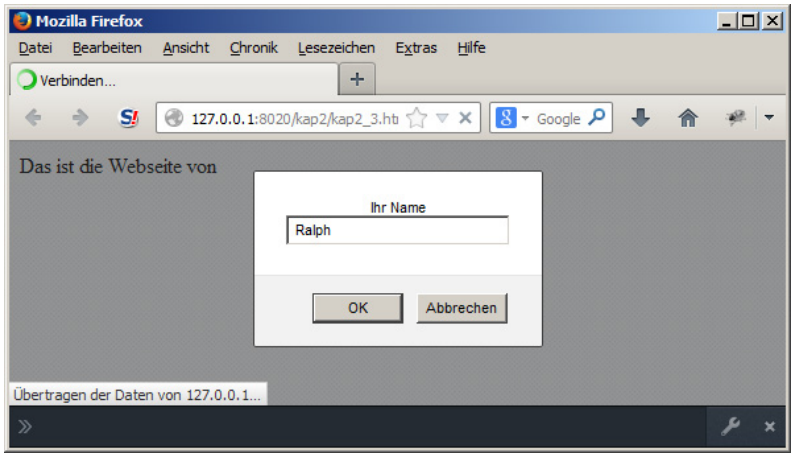

Bild 2.5 Ein Eingabedialog, der mit JavaScript generiert wurde

Im Anzeigebereich des Browsers sehen Sie nach der Eingabe eines Namens und der Bestätigung des Dialogs einen Text, der aus den statischen Textpassagen und der Eingabe durch den Anwender besteht.

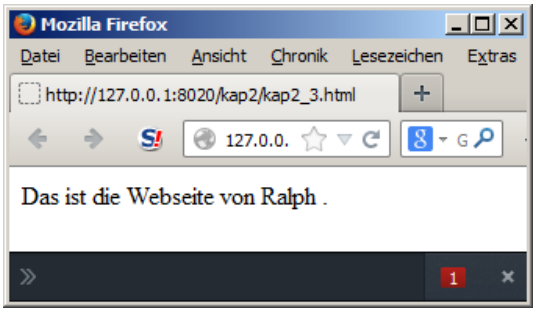

Bild 2.6
Eine weitere Seite, die aus statischen und dynamischen Informationen zusammengesetzt wurde

2.2 Einige Details zum Internet und zum WWW

Kommen wir zu ein paar Details rund um das Internet und das WWW. Ich gehe wie erwähnt davon aus, dass jeder Leser Erfahrung mit diesem Medium hat. Dennoch sollen einige Details geklärt werden, denn es gibt – trotz der Popularität des Internets – eine Reihe von Missverständnissen, die einer erfolgreichen Webprogrammierung im Weg stehen. Das Internet mit all seinen Diensten gehört als Ganzes niemandem und doch allen Internetteilnehmern zugleich. Es gibt einige Organisationen, welche sich mit den unterschiedlichsten Aspekten des Internets beschäftigen. Insbesondere auf Ebene der einzelnen Dienste existieren Organisationen zur Lenkung verschiedenster Belange. Besonders wichtig in Bezug auf die Webprogrammierung ist das **W3C** (World Wide Web-Consortium – *http://www.w3.org*), das für die Standards im WWW und damit auch HTML, Cascading Style Sheets und XML verantwortlich ist. Das W3C ist – wie alle diese Organisationen – jedoch weitestgehend auf die freiwillige Kooperation der Internetgemeinde und vor allen Dingen der Browserhersteller angewiesen. Leider aber kann durch den rein empfehlenden Charakter des W3C nicht das Problem gelöst werden, dass Vorgaben des W3C von Browserherstellern einfach ignoriert oder in einer nichtstandardisierten Weise umgesetzt werden.

2.3 Die Besonderheit bei der Webprogrammierung

Allgemein programmiert man heutzutage in einer sogenannten höheren Programmiersprache. Beispiele für solche höheren Programmiersprachen sind Basic, Java, C#, Cobol, Fortran, C/C++, Pascal oder auch JavaScript. Eine höhere Programmiersprache enthält einen Satz

von Klartextsprachelementen mit festgelegter Bedeutung, die meist der englischen Sprache angelehnt sind, und eine zugehörige Syntax. Die in einer höheren Programmiersprache geschriebenen Anweisungen (der **Quelltext**) sind reiner Klartext, der erst einmal für jeden potenziellen Zielprozessor (weitgehend) identisch sein kann. Eine höhere Programmiersprache ist bedeutend einfacher als Maschinensprache (eine Folge von Nullen und Einsen) zu handhaben, die aber letztendlich jeder Prozessor benötigt. Normalerweise erfolgt also moderne Programmierung, indem ein Programmierer Anweisungen an den Computer in einem einfachen Klartexteditor oder einer darauf aufbauenden integrierten grafischen Entwicklungsumgebung (IDE) eingibt. Bevor aus solchem Quelltext jedoch ein ausführbares Programm wird, muss irgendwann die Übersetzung in die prozessorabhängige **Maschinensprache** erfolgen.

2.3.1 Kompilierung versus Interpretation

Diese Überführung in Maschinensprache kann entweder direkt dann erfolgen, wenn der Quellcode fertig eingegeben ist. Der gesamte Quellcode wird in einem Arbeitsprozess in Maschinensprache übersetzt und das fertige Produkt wird an den Anwender weitergegeben. Dieser Prozess heißt **Kompilierung** und wird von der Entwicklungsumgebung – je nach eingestellter Option – weitgehend automatisch vollzogen. Dabei muss – je nachdem, auf welchem Prozessor das Programm laufen soll – unter Umständen für jede Plattform ein separates Programm erstellt werden.

Eine alternative Möglichkeit ist, dass der plattformneutrale Quelltext an den Anwender weitergegeben und erst Schritt für Schritt zu der Zeit übersetzt wird, wenn das Programm ausgeführt werden soll. In diesem Fall müssen Sie als Programmierer nur den Quelltext erstellen und unverändert weitergeben. Der Anwender lädt den Quelltext in ein passendes Programm und sobald eine Anweisung ausgeführt werden soll, wird sie unmittelbar davor beim Anwender übersetzt. Der Quelltext wird also zur Laufzeit eines Programms interpretiert und das Verfahren nennt sich dementsprechend Interpretation.

Nun hat sich im WWW eine der beiden Möglichkeiten zur Übersetzung des Quellcodes eindeutig durchgesetzt – die Interpretation. Aber warum?

2.3.2 Unterschiedliche Plattformen und Interpretation

Das Internet ist heterogen! Sagt Ihnen dieser Satz etwas? Heterogen? Nach Definition im Lexikon heißt das uneinheitlich, ungleichartig. In unserem Zusammenhang bedeutet dies, dass wir im Internet die unterschiedlichsten Plattformen vorfinden: Großrechner neben Apple-Computern und PCs sowie mobilen Geräten. Eine Vielzahl von unterschiedlichen Prozessoren tummelt sich im weltweiten Netz. Aber nicht nur das – Großrechnerbetriebssysteme existieren neben PC-Betriebssystemen, UNIX samt Derivaten neben Windows mit seinen vielen Ausprägungen. Auch die Kommunikationswege sind uneinheitlich.

Diese Heterogenität erzwingt eine ganz besondere Technik, auf welche Art und Weise Daten zwischen den Welten ausgetauscht werden müssen. Die Daten müssen für alle Systeme verständlich sein. Auf der einen Seite – dem Transport – übernehmen dies im Internet Pro-

tokolle wie TCP/IP, welche eine plattformneutrale Übertragung gewährleisten. Die darauf aufsetzenden Dienstprotokolle wie HTTP oder FTP gewährleisten die plattformneutralen Anwendungsumgebungen. Aber auch auf Seiten der konkreten Anwendung dürfen keine plattformbezogenen Anweisungen verwendet werden, wenn man nicht bewusst bestimmte Zielplattformen ausschließen will. Zwar gibt es auch im Internet einige Techniken, die beim Anwender bestimmte Voraussetzungen notwendig machen (etwa nur Windows und den Internet Explorer als Basis oder das Vorhandensein einer herstellerabhängigen Erweiterung) und dann eine weitreichende Funktionalität bieten. Im Internet wurden und werden solche dogmatischen Techniken jedoch kritisch gesehen. Vor allem werden Anwender ausgeschlossen, die bestimmte Produkte nicht verwenden können oder wollen. Die beste Variante für die Akzeptanz einer Technik im Internet scheint die vollkommene Plattformneutralität zu sein. Das ist auf jeden Fall bei Klartexttechniken gewährleistet, bei denen jede Anweisung auf jeder Plattform verstanden wird. Die Anweisungen werden durch ein spezifisches Programm interpretiert, das auf der jeweiligen Plattform läuft, und die neutralen Befehle werden in konkrete Prozessorbefehle übersetzt, die dann ausgeführt werden. Ich rede also vom Interpreter-Prinzip, das im Web den Königsweg darstellt.

2.3.2.1 Vorteile der Interpretation

Sprachen wie HTML oder JavaScript, aber auch XML sind exzellente Beispiele für so eine Konstellation. Sowohl HTML als auch JavaScript werden auf jeder Rechnerplattform verstanden. Über einen Interpreter – den Browser – werden die Anweisungen in die jeweiligen Prozessorbefehle übersetzt und ausgeführt (also interpretiert). Interpretation hat noch weitere Vorteile. Die Menge der zu übertragenden Daten ist kleiner, wie wenn die gesamte Funktionalität zu übertragen wäre. Außerdem ist es erheblich leichter, Änderungen durchzuführen, und die gesamte Wartung ist unkomplizierter. Der aufwendige Schritt der erneuten Kompilierung entfällt.

2.3.2.2 Nachteile der Interpretation

Die Interpretation bringt nicht nur Vorteile mit sich. Im Vergleich zu einer vollständig in Prozessorbefehle übersetzten Anwendung gibt es einige Nachteile.

Das Laufzeitverhalten von interpretierten Anwendungen ist zum Beispiel nicht gut. Da jeder Befehl erst zur Laufzeit (vollständig) übersetzt wird, wird so eine Anwendung durch die zusätzlichen Schritte langsamer, wie wenn der Prozessor sich nur mit dem Ausführen der Anweisungen zu beschäftigen hätte. Aber gerade im Fall von JavaScript bringen (fast) alle modernen Browser mittlerweile sogenannte „Just-in-time-Compiler“ mit. Damit lässt sich dieser Nachteil für moderne RIAs weitgehend aufheben.

HINTERGRUNDINFORMATION: Bei der Just-in-time-Kompilierung (JIT-Kompilierung) wird der Quellcode zwar auch erst zur Laufzeit und bei der ersten Verwendung in Maschinencode übersetzt. Dieser native Code kann aber dynamisch optimiert und schneller verarbeitet werden, wie wenn der Quellcode zeilenweise interpretiert wird. Und der native Code kann im Speicher gehalten werden, was bei einer erneuten Verwendung erhebliche Vorteile bringt.

Ein weiterer Nachteil ist es unter gewissen Umständen, dass die Befehle und Anweisungen recht leicht durch einen Anwender zu lesen sind (wenn er sich mit der verwendeten Sprache auskennt). Da Interpreter-Code in diesem Fall einfach eine Textdatei ist, ist dieser Quellcode kaum zu schützen, wenn ein Anwender Zugang dazu hat. Nicht jeder Programmierer möchte, dass seine ausgefeilten Befehlsstrukturen so leicht zu analysieren sind.

Ein großes Problem ist, dass Interpretation immer einen erheblichen **Spielraum** lässt, wie eine Anweisung zu verstehen ist. Insbesondere ist dies der Fall, wenn es sich nur um eine Beschreibungsanweisung handelt, aber grundsätzlich gilt das Problem auch für Skriptsprachen. Anhand von HTML lässt sich das Problem aber leichter verdeutlichen.

Es gibt in HTML beispielsweise eine Anweisung, dass der Anweisung folgender Text hervorgehoben dargestellt werden soll (`<strong>`), bis die Anweisung wieder aufgehoben wird (`</strong>`). Der Browser des Anwenders lädt nun eine solche Seite und kommt an den entsprechenden Befehl. Er interpretiert die Anweisung und führt sie aus. So weit, so gut. Aber wer legt fest, was *hervorgehoben* eigentlich bedeutet? Neigung der Schrift. Oder fett darstellen? Oder unterstreichen? Jede der Varianten wäre denkbar. Der Hersteller des Browsers A hat sich für die eine Variante entschieden, während der Hersteller des Browsers B eine andere vorzieht. Beide Browser werden also die Anweisung verstehen und durchführen, das Resultat wird aber bei beiden unterschiedlich aussehen. Diesen Interpretationsspielraum finden Sie bei den unterschiedlichsten Anweisungen in HTML und teilweise auch in anderen Techniken wie Style Sheets oder JavaScript. Zwar haben sich die Hersteller verschiedener Browser bei vielen Anweisungen weitgehend geeinigt. Es gibt jedoch immer noch Situationen, die von verschiedenen Browsern ganz unterschiedlich ausgelegt werden. Gerade bei neuen Befehlen dauert es lange Zeit, bis sich die verschiedenen Lager der Browserhersteller einigermaßen geeinigt haben[2]. Ein wichtiger Aspekt darf dabei auch nicht übersehen werden – Browser lassen sich individuell konfigurieren. Das betrifft Standardschriftarten, -größen, -stile, Farben usw. Auch dies führt dazu, dass es bei der Interpretation zahlreiche Darstellungen einer Seite geben kann.

Ein großes Problem beim Interpreterkonzept ist das Veralten der Interpreter. Wenn eine Interpretersprache einen neuen Befehl hinzugefügt bekommt, können die bis dahin entwickelten Interpreter diesen Befehl noch nicht kennen und ihn entsprechend nicht ausführen. Anwender mit solchen Interpretern können unter Umständen ein Programm nicht laufen oder ein Dokument nicht darstellen lassen.

[2] Ganz massiv sehen wir das im Moment bei HTML5 – konkret bei den neuen Formularelementen.