Web-Apps zum Wohlfühlen

ROCA: Keine Angst vor HTML und JavaScript

Phillip Ghadir

ROCA ist ein Architekturstil zur Entwicklung anständiger und zukunftsfähiger Web-Frontends. Er umfasst eine Reihe von Empfehlungen sowohl für die Client- als auch für die Serverseite. Der ROCA-Stil erfordert von vielen Java-Entwicklern ein gewisses Umdenken. Grund genug, sich den Stil genauer anzuschauen.

Böse Zungen behaupten, dass Java-Entwickler keine ordentlichen Webanwendungen bauen können, weil sie stets versuchen, das Gute des Webs so zu kapseln, dass es bei der Entwicklung nicht mehr im Weg steht. Mit der Kapselung gehen dann aber auch wertvolle Features des Webs verloren oder stehen nur eingeschränkt zur Verfügung.

Die Welt in meinem Browser

Für uns Internetnutzer spielt sich alles in unserem Browser ab. Manche von uns verwenden gleich mehrere, um so zum Beispiel vor spezifischen Angriffen wie CSRF oder XSS geschützter zu sein (s. [CSRF] bzw. [XSS]). Andere machen alles im Browser: Sie programmieren, schreiben E-Mails, zeichnen Diagramme, bearbeiten Fotos, erstellen 3D-Modelle und lassen den Abend mit einem netten Spiel ausklingen.

Der Browser ist die meistverbreitete Laufzeitumgebung im Internet. Browser sind spezialisiert auf die Darstellung von mit CSS gestyltem HTML und auf die Ausführung von JavaScript-Programmen. Häufig liefern Webanwendungen schlechten Input, aus dem der Browser dann etwas machen muss. Browser sind sehr gutmütige Laufzeitumgebungen: Sie verstehen verschiedene Dialekte, unzählige Versionen, tolerieren sowohl Syntax- als auch Semantikfehler und schaffen es häufig, aus schlechtem Material eine brauchbare Darstellung zu liefern.

Und weil der Browser so gutmütig ist, wird er von uns Entwicklern auch recht gern gering geschätzt oder gar misshandelt.

Minderwertiges JavaScript?

JavaScript gering zu schätzen, fällt uns leicht. Das machen Script-Kiddies oder Internet-Hipster, aber niemand, der laufähige Software liefern und über Jahre warten muss. Allein das Namensanhängsel „Script“ sagt schon alles. Dabei ist JavaScript eine durchaus beachtliche Sprache, über die man Ähnliches wie auch über die Stadt Hagen sagen kann: „Es gibt auch schöne Ecken.“ – Ich denke nicht, dass Douglas Crockford beim Schreiben von „JavaScript – The Good Parts“ [Crock08] an diesen Spruch erinnern wollte.

Für uns Java-Entwickler gibt es ein paar Dinge, die wir verinnerlichen sollten. ROCA (Resource-oriented Client Architecture) fordert unter anderem den besonnenen, zurückhaltenden Einsatz von JavaScript. Deshalb lohnt es sich, die Grundlage für ein Verständnis der strukturellen Zusammenhänge zwischen den Komponenten zu legen und einige Details von JavaScript zu betrachten. Dazu zählen Sichtbarkeiten, Closures, Objekte und die Prototyp-Basierung in JavaScript. Für das Verständnis des restlichen Artikels können wir auf Prototypen und Erbung verzichten, beim Einsatz verschiedener JavaScript-Frameworks ist deren Verständnis allerdings wohl erforderlich.

Kurz: Objekte schreibt man in JavaScript einfach per `{}`. Attribute und Funktionen können literal ins Objekt hineingeschrieben werden, wie in Listing 1, Zeilen 4 - 6 zu erkennen ist. Analog können Eigenschaften auch per Zuweisung hinzugefügt werden:

```
var obj = {};
obj.name = "ASSIGNMENT";
obj.doit = function() {};
```

Global vs. lokal

In JavaScript gibt es nur zwei Sichtbarkeiten: *global* oder *lokal*. Wenn man also in JavaScript versucht, private Eigenschaften oder Funktionen zu deklarieren, kann das nur als lokale Variablen oder Funktionen innerhalb einer Funktion funktionieren.

Listing 1 zeigt die Definition der Funktion `f`, die im Methodenrumpf sowohl eine lokale Variable als auch eine lokale Funktion definiert. Die lokale Funktion ist sichtbar für das in derselben Methode definierte Objekt, das zurückgegeben wird (s. Zeile 10 in Listing 1). Außerhalb von `f()` ist der Zugriff auf `innerF()` nicht möglich.

```
 1 function f() {
 2   console.log( "f()" );
 3   var lokaleVariable = "Wert";
 4   function innerF() {
 5     console.log( "innerF()" );
 6   };
 7   return {
 8     abc : function() {
 9       console.log( "abc()" );
10       innerF();
11     }
12   }
13 }
```

Listing 1: JavaScript-Funktionen kapseln Details wie z. B. andere Funktionen

Die Tücken mit this

Obwohl von Java-Entwicklern geschriebenes JavaScript für einen Java-Entwickler recht vertraut aussehen kann, unterscheidet sich das Referenzieren von Attributen, Methoden und Objekten grundlegend in den beiden Sprachen. Java löst einen Namen stets auf, indem implizit ein „this.“ hinzugefügt wird, falls innerhalb einer Instanzmethode ein Name nicht direkt definiert ist. Für statische Methoden wird implizit „<Klassenname>.“ hinzugefügt.

Solch ein implizites Binden gibt es in JavaScript nicht. Jeder Kontext muss explizit angegeben werden. Anstatt wie in Java eine Methode derselben Instanz per `siblingMethod();` aufzurufen, zwingt uns JavaScript zu einem Aufruf per `this.siblingMethod()`.

In verschiedenen JavaScript-Frameworks findet man häufig Zeilen der Art:

```
var that = this;
```

oder auch

```
var self = this;
```

Diese Zuweisungen erfolgen vor der Definition einer Funktion oder eines Objekts im gleichen Scope. Die Bindung von

`self` ist unabhängig von dem späteren Kontext, in dem die Funktion oder das Objekt verwendet wird. `self` ist weiterhin an die Instanz gebunden, die zum Zeitpunkt der Definition per `this` referenziert war. Weil die Bindungen im umgebenden Kontext bei der Definition geschlossen werden, nennt man das Konstrukt Closure [Closure].

```
function setup() {
  var body = document.getElementById( "content" );

  if ( body !== null ) {
    body.onclick = function () {
      console.log( this.name );
    }
  } else {
    console.log( "Strange.... Can't find body!" );
  }
}
```

Listing 2: Registrieren eines Ereignis-Handlers

Listing 2 zeigt eine JavaScript-Funktion, die zu einem Element mit Id „body“ im HTML-Dokument einen Ereignis-Handler hinzufügt. Beim Click des Elements „content“ wird der Ereignis-Handler aufgerufen. Bei dessen Ausführung zeigt `this` auf das HTML-Element und nicht mehr auf den die Funktionsdefinition umgebenden Code, den das Listing zeigt.

Dank der Closures ist es dennoch möglich, Funktionen als Ereignis-Handler zu registrieren, die von anderen Methoden des definierenden Objekts abhängen.

```
var Entity = {
  init: function() {
    var obj = {
      name: null,
      pos: { x: 0, y: 0 },
      toString: function () {
        return obj.name + " at {" + obj.pos.x + ", " + obj. pos.y + "}";
      }
    };
    return obj;
  }
};
```

Listing 3: JavaScript-Objekt Entity definiert Methode init() zum Erzeugen einer Instanz mit Attributen name und pos

In Listing 3 sieht man, dass das in `init()` erzeugte Objekt einer lokalen Variablen `obj` zugewiesen wird. Innerhalb des Scopes der Funktion `init()` kann nun die Instanz per „obj“ referenziert werden. So können die Attribute „name“ und „pos“ ausgelesen werden, auch wenn das Objekt hinterher in einem ganz anderen Kontext steht.

ROCA-konforme Clients

Um ein System mit einem ROCA-konformen Frontend zu versehen, müssen folgende Regeln auf der Clientseite eingehalten werden:

Der Client erhält semantisch strukturiertes HTML, das frei von darstellungsspezifischem Markup ist. Dieses HTML muss auch für Werkzeuge wie Screenreader oder Ähnliche zugreifbar sein. Das Layout wird über CSS definiert. Sowohl HTML als auch CSS werden so verwendet, dass das Frontend auch dann funktioniert, wenn der Browser nicht alle Merkmale unterstützt.

Progressive Enhancement – das schrittweise Verbessern der Benutzungsschnittstelle – ist eine der Kernanforderungen, bei denen mächtigere Ausdrucksmöglichkeiten und Browserfunktionen die bereits funktionierende Schnittstelle noch besser nutzbar machen. Das Frontend muss auch ohne JavaScript grundsätzlich (unter Umständen auch umständlich) funktionieren. JavaScript wird ebenfalls eingesetzt, um das Frontend schrittweise zu verbessern. Daher muss JavaScript unaufdringlich eingesetzt werden.

Applikationslogik darf nicht redundant implementiert werden. (Auch die Ausrede, dass die Funktionalität sowohl auf dem Client als auch im Server laufen muss, zählt nicht.) Das Wissen über die Dokumentstrukturen darf nicht auf Client und Server verteilt werden, sondern muss eindeutig an einer Stelle definiert sein. Die einzige zulässige Verwendung des Knowhows über die Struktur ist im CSS. JavaScript und CSS werden explizit programmiert und statisch ausgeliefert und nicht programmatisch auf Anfrage erzeugt. Clientseitiges Routing oder Anpassen der URI-Zeile sollte das HTML5-History-API verwenden.

Schauen wir uns die Highlights näher an.

Semantischer Markup

Der Markup soll Daten inhaltlich ausgestalten. Dazu können die HTML-Elemente selbst helfen. Denn DL (Definition List), FIELDSET, LI (List Item), H1-H6 (Überschriften) und TABLE sind Beispiele für Elemente, die ausdrücken, welcher Art die ausgezeichneten Daten sind. Dazu bietet [mwesch] eine sehr stimmungsvolle, elterntaugliche Erläuterung.

Das class-Attribut der HTML-Elemente bietet sich an, um Daten anwendungsspezifisch und stereotypisch auszuzeichnen. Das Annotieren mit Klassen im Markup kann man sich analog zum Annotieren oder Implementieren von Schnittstellen im Java-Code vorstellen.

```
<div class="Panel">
  <h2>Kundenkurzinfo</h2>
  <dl class="Entity Customer">
    <dt>Primärschlüssel</dt>
    <dd class="attribute_value" attribute_name="primary_key">4711</dd>
    <dt>Vorname</dt>
    <dd class="attribute_value"
        attribute_name="first_name">Phillip</dd>
    <dt>Nachname</dt>
    <dd class="attribute_value"
        attribute_name="last_name">Ghadir</dd>
  </dl>
  <menu>
    <li><a href="/customers/4711/details">Detailansicht</a></li>
    <li><a href=
      "/customers/4711/scorecards/latest">letzte Selbstauskunft</a></li>
  </menu>
</div>
```

Listing 4: Die Kurzinfo einer Entität semantisch als Definitionsliste ausgezeichnet

Wie die Daten ausgezeichnet werden, hängt stark vom Anwendungsfall ab. Wenn sich der Markup bereits an die darzustellenden Informationen anlehnt, ist die halbe Miete gewonnen. Listing 4 zeigt ein ROCA-konformes HTML-Fragment. Bei der Darstellung werden die Regeln des ROCA-Stils *posh*, *accessibility* sowie *progressive enhancement* befolgt (s. Kasten „ROCA-Kurzübersicht“). Keine Spur von JavaScript verschmutzt die Daten, die so von einem Backend geliefert werden. Einzig die Verwendung eines von uns eigens erfundenen Attributs „attribute_name“ deutet darauf hin, dass wir dieses HTML nicht nur im Browser verwenden möchten.

ROCA-Kurzübersicht

Anforderungen an die Clientseite

- posh: Das gute alte semantisch-sinnvolle HTML.
- accessibility: Jede Seite muss auch per Screenreader oder Ähnlichem bedient werden können.
- progressive enhancement: CSS wird für die Darstellung verwendet. Auch wenn der Browser CSS nur eingeschränkt unterstützt, steht die Funktionalität einer Seite dennoch zur Verfügung.
- unobtrusive javascript: Die Verwendung von JavaScript erleichtert die Nutzung oder verbessert einige Aspekte, das System bleibt aber auch nutzbar, wenn JavaScript nicht zur Verfügung steht.
- no-duplication: Die Applikationslogik verbleibt auf dem Server und darf nicht auf dem Client doppelt implementiert werden.
- know-structure: Client & Server dürfen keine Annahmen über die Struktur der Daten über die initiale Struktur hinaus treffen, die vom Server geliefert wird.
- static-assets: JavaScript und CSS werden als statische Dokumente ausgeliefert. Dies erlaubt die sinnvolle Verwendung der Web-Infrastruktur – wie Caches – und fördert die Wartbarkeit.
- history-api: Wenn mit JavaScript der Zustand im Browser geändert wird, ermöglicht die [HistoryAPI] das geeignete Repräsentieren des neuen Zustands in der Browserzeile, ohne die Seite neu zu laden.

Anforderungen an die serverseitige Implementierung

- rest: Das Backend exportiert die Applikationslogik REST-konform. Das bedeutet, sie erlaubt die zustandslose Kommunikation, das Referenzieren von Ressourcen per URI, den einheitlichen Zugriff über eine universelle Schnittstelle, die Verlinkung von Ressourcen sowie mehrere Repräsentationen einer Ressource.
- application-logic: Das Backend realisiert die Fachlogik vollständig und verlässt sich nicht darauf, dass Teile – wie Berechnungen oder Prüfungen – bereits im Client ausgeführt wurden.
- http: ROCA stellt die REST-konforme Verwendung von HTTP in den Vordergrund. Die Realisierung eigener Protokolle (z. B. per Web-Sockets) ist unerwünscht.
- link: Informationselemente der Anwendung müssen separat verlinkt werden können.
- non-browser: Eine ROCA-konforme Anwendung kann auch mit einfacheren Clients wie curl oder wget verwendet werden.
- should-formats: Das primäre Format ist HTML. Ressourcen können darüber hinaus aber auch in zusätzlichen Formaten, wie JSON oder XML, angeboten werden.
- auth: Die Authentisierung erfolgt sinnvollerweise über Basic-Auth per SSL oder aber Formular-basiert mit Cookies zur Identifikation.
- cookies: Cookies werden nur zur Identifikation des Nutzers und nicht für sonstige Zwecke verwendet.
- session: ROCA-konforme Systeme erfordern keine Dialoge samt Dialoggedächtnis und speichern auch keine Session-Informationen über die Authentisierungsinformationen hinaus.
- browser-controls: ROCA-konforme Anwendungen erlauben das Verwenden der Browser-Aktionen (Neu-Laden, Zurück, Vor), ohne dabei in einen undefinierten Zustand zu geraten.

Abb. 1: Die Kurzinfo (mit CSS aufbereitet) im Browser dargestellt

Progressive Enhancement via Unobtrusive JavaScript

Mittels JavaScript können nun diese Daten um Verhalten angereichert werden, ohne dass dazu spezielle zusätzliche Auszeichnungen nötig wären. Der ROCA-Stil fordert, dass JavaScript nur optional und unaufdringlich ergänzt werden darf. Mit einem JavaScript-Framework wie jQuery (s. [JQuery]) kann man mit der Funktion `$()` auf beliebige Elemente im Dokument zugreifen und dabei CSS-Selektoren zum Referenzieren der gewünschten Elemente verwenden.

```
function setupPanels() {
  $( ".panel menu li" ).on(
    'click',
    function() {
      var url = $(this).children("a");
      if ( url !== null ) {
        window.location = url.attr('href');
      }
    } );
}
```

Listing 5: JavaScript sorgt dafür, dass auch die Box selbst anklickbar wird

Listing 5 zeigt, wie mit Hilfe eines der jQuery-Selektoren die Menüeinträge für ein Panel angepasst und genauso anklickbar gemacht werden, wie die Links. Sollte diese Initialisierung nicht ausgeführt werden, muss der Anwender genauer zielen und exakt den Link treffen. Wird JavaScript unterstützt, erleichtert das Hinzufügen des OnClick-Event-Handlers die Bedienung. Das ist Progressive Enhancement – wenn auch nur in einem ganz kleinen trivialen Schritt.

ROCA ermöglicht Komponentenbildung

Das Auszeichnen des Markups und die stereotypische Behandlung von selektierbaren Dokumentfragmenten per JavaScript bilden gemeinsam eine Komponente, mit einem Schnittstellenteil (dem Markup bzw. dem Dokument) und der Implementierung, die per Setup-Funktionalität gebunden wird. Abbildung 2 zeigt, wie ROCA-konform komponentenbasiert Frontends lose zusammengesteckt werden können.

Dadurch wird es zum Beispiel möglich, spezielle Validatoren zu realisieren, oder Markup in Komponenten mit verbessertem Verhalten zu transformieren. Im Entwicklungsteam

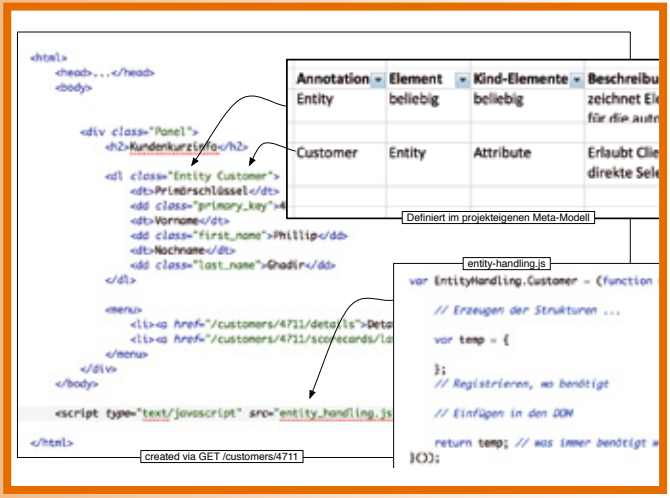

Abb. 2: Zusammenhang zwischen Meta-Modell, dem Markup und der Komponentenimplementierung

genügt es, das erforderliche Know-how auf einige Wenige zu beschränken. Nicht jeder im Projekt muss zum JavaScript-Guru werden.

Code-Duplizieren verboten

Das nächste Prinzip verbietet, dass Logik dupliziert wird. ROCA-konform realisiert das Backend die komplette Fachlogik. Häufig erfordern die Bedürfnisse an Ergonomie und Reaktionsgeschwindigkeit allerdings, dass einige Prüfungen bereits auf dem Client ausgeführt werden.

Dies ist kein Widerspruch, sofern solche Implementierungen an einer Stelle definiert und dann sowohl client- als auch serverseitig ausgewertet werden können.

ROCA auf der Serverseite

Wie gerade schon erwähnt, muss die gesamte Applikationslogik im Backend ROCA-konform realisiert sein. Das Backend darf sich nicht darauf verlassen, dass Prüfungen oder andere Aufbereitungen bereits im Client passiert sind, sondern muss Berechnungen oder Ähnliches selbst durchführen. Der Zugriff auf die Logik soll REST-konform per HTTP erfolgen. Daraus ergibt sich, dass zu jedem Zeitpunkt der Inhalt der Adresszeile des Browsers kopiert, „gebookmarked" und verschickt werden kann, um den aktuellen Client-Zustand grundsätzlich wieder herzustellen.

Die serverseitige Logik muss auch mit einfachen HTTP-Clients bedient werden können. Daher darf zum Beispiel nicht das clientseitige Ausführen von JavaScript vorausgesetzt werden. Die Authentisierung hat über HTTP-Basic-Auth oder Form-based-Auth zu erfolgen. Wer formularbasiert authentisiert, darf die Informationen zur Identifikation des Benutzers im Cookie speichern. Über den Zweck der Benutzeridentifikation hinaus dürfen Cookies nicht verwendet werden. Das bedeutet insbesondere, dass auch kein transienter Session-State gehalten werden darf. Die Anwendung muss mit einem üblichen Browser bedient werden können und dabei jederzeit die Nutzung der Browser-Aktionen Neu-Laden, Zurück und Vorwärts zulassen, ohne dabei den Nutzer zu verwirren, negativ zu überraschen oder in undefinierte Zustände zu geraten.

Zusammenfassung

ROCA ist ein Architekturstil, der sowohl Einschränkungen an die Entwicklung der Client- als auch der Serverseite formuliert. Diese Einschränkungen helfen, Anwendungen und Systeme zu bauen, die sich elegant in das World Wide Web einbetten und natürlich mit einem Web-Browser (oder bei Unterstützung auch anderer bzw. spezifischerer Clients) verwendet werden können.

Dazu setzt ROCA sehr stark auf bewährte Prinzipien der Softwaretechnik, fordert die Einhaltung des REST-Stils auf der Server- und auf der Clientseite, zumindest das Verwenden von HTML, CSS und JavaScript zur Bereitstellung der Clientsicht.

ROCA verbietet im Wesentlichen die Zauberei zur Laufzeit im Betrieb und fordert, dass Inhalte wie CSS oder JavaScript zur Entwicklungszeit zusammengestellt und ausgeliefert werden, sowie die strikte Trennung der semantisch strukturierten Daten von JavaScript und CSS. In der Konsequenz bedeutet ROCA für Entwickler, die gewohnt sind, JavaScript und HTML eher zu kapseln, ein gewisses Umdenken. Der Kasten „ROCA-Kurzübersicht" fasst die Forderungen des Architekturstils noch mal zusammen.

Links

[BSI] Barrierefreies E-Government, Bundesamt für Sicherheit in der Informationstechnik, https://www.bsi.bund.de/cae/servlet/contentblob/476832/publicationFile/28312/4_Barriere_pdf.pdf

[Closure] http://de.wikipedia.org/wiki/Closure

[Crock08] D. Crockford, JavaScript: The Good Parts, O'Reilly, 2008

[CSRF] Cross-Site Request Forgery auf Wikipedia, https://en.wikipedia.org/wiki/Cross-site_request_forgery

[HistoryAPI] http://diveintohtml5.info/history.html

[JQuery] JQuery Core, http://jquery.com

[mwesch] The machine is Us/ing Us, http://www.youtube.com/watch?v=NLlGopyXT_g

[OSArch] F. Hoppe, T. Schulte-Coerne, St. Tilkov, ROCA: Resource-oriented Client Architecture, in: OBJEKTspektrum, Online Themenspecial Architekturen 2012, http://sigs.de/publications/os/2012/Architekturen/hoppe_schulte-coerner_tilkov_OS_Architekturen_12_hgz8.pdf

[Podcast] http://www.heise.de/developer/artikel/Episode-38-Barrierefreiheit-1780121.html

[Port08] Ch. Porteneuve, Getting Out of Binding Situations in JavaScript, 1.7.2008, A List Apart, Issue 262, http://alistapart.com/article/getoutbindingsituations

[ROCAstyle] http://roca-style.org

[XSS] Cross-site scripting auf Wikipedia, https://en.wikipedia.org/wiki/Cross-site_scripting

Phillip Ghadir baut am liebsten tragfähige, langlebige Softwaresysteme. Er ist Mitglied der Geschäftsleitung bei innoQ und hat sich früh auf Architekturen für verteilte, unternehmenskritische Systeme spezialisiert. Darüber hinaus ist er Mitbegründer und aktives Mitglied des iSAQB, des International Software Architecture Qualification Board.
E-Mail: phillip.ghadir@innoq.com