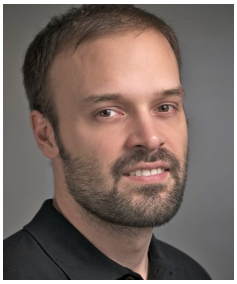
Falk Hoppe
(falk.hoppe@innoq.com)
ist Consultant bei innoQ Deutschland GmbH und seit mehreren Jahren in der agilen Entwicklung von Webanwendungen mit Ruby und Rails sowie PHP tätig. Seine Schwerpunkte liegen in der Implementierung von verteilten ergonomischen Anwendungen auf REST-Basis, inklusive einer Client-Anbindung mit modernen Frontend-Technologien.



Till Schulte-Coerne
(till.schulte-coerne@innoq.com)
ist Senior Consultant bei innoQ Deutschland GmbH, realisiert seit mehreren Jahren Webanwendungen mit Ruby on Rails, Java und PHP. Sein Schwerpunkt liegt auf der Architektur und Implementierung ergonomischer Webanwendungen.

Stefan Tilkov
(stefan.tilkov@innoq.com)
ist Geschäftsführer und Principal Consultant bei der innoQ Deutschland GmbH, wo er sich vorwiegend mit der strategischen Beratung von Kunden im Umfeld von Softwarearchitekturen beschäftigt. Er ist Autor des Buchs „REST und HTTP", Mitherausgeber von „SOA-Expertenwissen", Autor zahlreicher Fachartikel und häufiger Sprecher auf internationalen Konferenzen.

ROCA: Resource-oriented Client Architecture

Eingezwängt zwischen statusbehafteten, serverseitigen Komponenten-Frameworks auf der einen und Single-Page-Apps auf der anderen Seite könnte man meinen, die klassische Architektur von Webanwendungen hätte ausgedient. Das stimmt jedoch keineswegs: Schöpft man das Potenzial des Webs vernünftig aus, so stellt man fest, dass es gerade ohne ein Verbiegen der Grundprinzipien möglich ist, skalierbare und ergonomische Anwendungen zu entwickeln. ROCA (Resource-oriented Client Architecture) ist der Name für einen Ansatz, der diesem Muster folgt und sich vor allem durch den richtigen Einsatz von JavaScript sowie die Einhaltung von REST-Prinzipien auszeichnet.

ROCA [ROCA] ist eine Menge von Vorgaben, die zusammengenommen eine von vielen möglichen Arten definieren, auf denen man Webanwendungen realisieren kann. Dabei ist ROCA weder ein Produkt noch ein Framework, noch wird eine bestimmte Programmiersprache vorausgesetzt. Wir haben zwar nach einem Brainstorming den Namen gewählt und die Prinzipien formuliert, aber in keiner Weise etwas Neues erfunden. Die Motivation für ROCA war der Wunsch, einem Ansatz, der unserer Erfahrung nach in der Praxis viele Vorteile mit sich bringt, einen Namen zu geben und ihn klar zu definieren.

Die wesentlichen Grundpfeiler von ROCA, auf die in diesem Artikel detailliert eingegangen werden soll, sind die Einhaltung von REST-Prinzipien, die Generierung von HTML auf der Serverseite und der Einsatz von *Unobtrusive JavaScript*. Um genauer zu verstehen, welche Elemente den ROCA-Stil ausmachen, lohnt es sich, Server und Client getrennt voneinander zu betrachten.

REST: Nicht nur für Webservices

Häufig wird REST (Representational State Transfer) als alternativer Ansatz für Webservices (anstelle von SOAP, WSDL und WS-*) positioniert. Daraus könnte man schließen, dass sich REST ausschließlich auf die Anwendungs-zu-Anwendungskommunikation bezieht.

Das greift jedoch erheblich zu kurz: REST ist der Architekturstil des Webs – unabhängig davon, ob der Client ein anderer Server, eine spezialisierte native Anwendung oder eben ein Web-Browser ist. Der REST-Architekturstil definiert eine Reihe von Randbedingungen (Constraints), an die sich eine Architektur halten muss, damit sie als REST-konform bezeichnet werden kann.

Diese im Detail zu diskutieren, würde den Rahmen dieses Artikels sprengen, hierzu sei auf [REST] verwiesen. Für die Realisierung von Webanwendungen, also von HTTP-Anwendungen, die als wichtigsten Client einen Web-Browser haben, sollte der Server-Anteil einer Reihe von Regeln folgen, die sich aus den REST-Constraints ableiten lassen:

- Uniform Resource Identifiers (URIs) haben Bedeutung und identifizieren eine Hauptressource,
- die Kommunikation zwischen Client und Server erfolgt statuslos,
- die Integration von Ressourcen erfolgt über Verknüpfungen (Links) und
- optional werden mehrere Repräsentationen unterstützt.

Dass mit URIs sinnvolle Dinge identifiziert werden sollen, sollte eigentlich eine Selbstverständlichkeit sein. Das ist jedoch nicht der Fall: Viele vermeintlich moderne Webanwendungen nutzen eine URI als Adresse der Anwendung selbst. Alles, was danach innerhalb der Anwendung geschieht, verändert die URI nicht, die in der Adresszeile des Browsers angezeigt wird.

Das wiederum führt dazu, dass sich auf Inhalte keine Lesezeichen setzen lassen, die Zurück- und Vorwärtsschaltflächen des Browsers nicht mehr funktionieren, man niemandem einen Link auf ein Objekt innerhalb der Anwendung senden und Seiten nicht in einem neuen Fenster oder Tab öffnen kann – kurz: das, was der

Benutzer von seinem Browser eigentlich erwarten kann, funktioniert nicht mehr. Im Gegensatz dazu identifiziert beim ROCA-Stil die URI, die der Browser referenziert, ein Geschäftskonzept in der Anwendung. Daraus folgt auch, dass zu jedem Zeitpunkt klar sein muss, welches Konzept im Mittelpunkt steht.

Die Forderung nach einer Hauptressource mag zunächst wie eine Limitation klingen, ist tatsächlich aber ein wesentlicher und gewünschter Effekt. Zwar zeigen viele Webanwendungen Elemente oder Objekte aus verschiedenen Kontexten auf einmal an, in den meisten Fällen jedoch ist das in Wirklichkeit Zeichen eines Problems, nämlich des Wunsches, Konzepte aus Desktop-Anwendungen auf das Web zu übertragen.

Damit Browser und der Rest der Webinfrastruktur (wie z. B. Caching- oder Redirect-Unterstützung) so funktionieren, wie es deren Erfinder geplant hatten, und sie den größten Nutzen bringen, muss zumindest ein Konzept klar als das wichtigste identifiziert werden können. Zusätzlich sollten Informationen über Links erreichbar sein, welche wiederum – wie wir weiter unten sehen werden – elegant dynamisch in Voransichten aufgelöst werden können. Dies ist symptomatisch für den ROCA-Ansatz, bei dem sich vermeintliche Nachteile der Webinfrastruktur als die Vorteile herausstellen, die sie in Wirklichkeit sind.

Browser und serverseitige Webanwendung müssen statuslos kommunizieren, damit das Prinzip von per URI identifizierten Ressourcen, die als Einsprungpunkt dienen können, auch in der Praxis funktioniert. Statuslos bedeutet natürlich nicht, dass sich der Server keinen Status merken darf, sondern vielmehr, dass jeder Request alle Informationen beinhalten muss, die der Server zu dessen Verarbeitung benötigt. Damit können aufeinanderfolgende Requests von jedem beliebigen Serverknoten verarbeitet werden, da dieser keine Information über den aktuellen Status der Verarbeitung im Client vorhält.

Auch an dieser Stelle hinterfragen wir mit ROCA eine akzeptierte Weisheit, nämlich die, dass die statusbehaftete Verarbeitung ein zentraler Unterschied zwischen Websites auf der einen und Webanwendungen auf der anderen Seite ist. Nach unserer Philosophie gibt es diesen Unterschied nicht; er ist künstlich und ergibt sich zum einen aus dem Versuch, Desktop-Programmiermodelle auf das Web zu übertragen und zum anderen aus der Geschichte von Webanwendungen, die ursprünglich vollständig ohne die Möglichkeit zur Umsetzung clientseitiger Präsentationslogik auskommen mussten.

Betrachten wir *zwei Beispiele*, die auf den ersten Blick den Eindruck erwecken, sie seien ohne statusbehaftete Kommunikation nicht realisierbar: Einen Bestellprozess mit mehreren Schritten sowie einen „Wizard“, mit dessen Hilfe eine komplexe Erfassung auf mehrere Masken verteilt werden soll.

Im ersten Fall ist ein denkbarer Ansatz, die einzelnen Schritte des Bestellprozesses nicht auf einen Sitzungsstatus im Server-Prozess abzubilden, sondern auf eine eigene Ressource, die die Bestellung abbildet und eine eigene URI hat. Mit ihr wird in mehreren Schritten interagiert bis sie schließlich so weit fortgeschritten ist, dass sie abgeschlossen werden kann. Jeder Request vom Client richtet sich dabei an die per URI identifizierte Bestellung. Das bedeutet natürlich, dass diese auf dem Server entsprechend vorgehalten wird, was zusätzlichen Aufwand erfordert. Dafür ist die Bestellung, in deren Kontext man sich nun mit dem Bestellprozess bewegt, explizit anstatt sich implizit aus der Kommunikationsbeziehung zu ergeben. So kann der geneigte Kunde nun mehrere Bestellungen in mehreren Fenstern oder Tabs bearbeiten, ein Lesezeichen setzen oder einen Link darauf verschicken.

Im anderen Beispiel, dem Wizard, erscheint diese Vorgehensweise nicht sinnvoll – hier ist es für den Server möglicherweise eine überflüssige Belastung, sich die einzelnen Zwischenergebnisse zu merken. In diesem Fall kann durch den geeigneten Einsatz von JavaScript ein vom Server zum Client übermitteltes Formular in mehrere einzelne Seiten aufgeteilt und sukzessive angezeigt werden. Nachdem der Anwender alle Informationen eingegeben hat, wird der komplette Formularinhalt in einem Schritt zum Server übertragen – für den diese Übertragung so aussieht, als hätte es den Wizard-Gedanken nie gegeben.

Welche der beiden Varianten im jeweiligen Szenario die richtige ist, ist eine Frage des Anwendungsentwurfs, in dessen Rahmen entschieden werden muss, welche Granularität die Ressourcen haben sollen. Dass die richtige Antwort hierauf kaum eine einzige Ressource für die gesamte Anwendung sein kann, sollte eigentlich klar sein.

Der nächste Punkt, der Einsatz von Verknüpfungen für die Integration, betrifft im Grunde nicht nur die Serverseite, sondern das Gesamtsystem: Das jedem bekannte und harmlos wirkende Konzept eines Links ist in Wirklichkeit der mächtigste und am weitesten verbreitete Integrationsmechanismus, den die IT bislang erfunden hat. Über einen Link können Ressourcen derselben Anwendung, derselben Domäne oder aus völlig unterschiedlichen Hoheitsbereichen miteinander verbunden werden.

Die großzügige Verwendung von Links trägt dazu bei, dass der vielfältige nicht nur soziale Beziehungsgraph, den wir in unserer Wirklichkeit finden, sich in unseren Informationssystemen widerspiegelt. Gleichzeitig ergibt sich für die lose Kopplung von Komponenten ein perfekt minimalistischer Ansatz: Zwei Systeme, die nicht das Geringste miteinander zu tun haben, können für den Anwender im Dialogfluss miteinander integriert werden, obwohl dadurch nur eine sehr geringe Kopplung erzeugt wird.

Schließlich ist eine Webanwendung, die REST- und ROCA-Prinzipien folgt, nicht weit von einem RESTful Webservice entfernt: Indem man entweder darauf achtet, serverseitig erzeugte HTML maschinenlesbar zu machen oder für die Ressourcen alternative Formate (z. B. JSON oder XML) anbietet, kann ein und dieselbe Anwendung auch von anderen Clients als dem Browser verwendet werden.

Dieser letzte Punkt passt zu einer weiteren Säule des ROCA-Stils, die weitreichende Konsequenzen hat: Sämtliche Applikationslogik liegt auf dem Server, der damit dem Prinzip der Trennung von Schnittstelle und Implementierung folgend eine Zugriffsmöglichkeit auf eine gekapselte Menge von Daten und Operationen bietet.

Wenn Sie sich mit REST bereits beschäftigt haben, werden Sie in diesem Abschnitt keine Überraschung erlebt haben. Das ist Absicht: ROCA setzt auf der Serverseite nicht viel mehr voraus, als dass sich der Anwendungsentwurf an REST-Prinzipien hält. Ein beliebiger Client, egal in welcher Technologie implementiert, könnte die Schnittstelle des Servers so nutzen, dass wir die Gesamtarchitektur als RESTful bezeichnen könnten.

Anders sieht es beim Einsatz eines Browser-Clients aus, für den sich aus REST allein noch keine wesentlichen Vor-

gaben ergeben. An dieser Stelle ergänzen wir mit ROCA die Aspekte, die dafür sorgen, dass sich die Anwendung als Ganzes optimal in die Webinfrastruktur einbettet.

Client-Seite

Auf der Client-Seite stehen im menschenlesbaren Web vor allem drei Technologien im Vordergrund:

- HTML,
- Cascading Style Sheets (CSS),
- JavaScript.

All diese Technologien sind nach einem bestimmten Grundmuster gestrickt: Sie lassen Raum für Erweiterungen. Konkret befolgen sie alle das Prinzip des *Progressive Enhancement*: Jedes neue Feature wird so implementiert, dass alte Browser weiterhin funktionieren können.

Man sieht dies beispielsweise deutlich beim neuen HTML5 Standard. Dieser enthält unter anderem neue Formulartypen (z. B. <input type=email>), die so definiert sind, dass ein Browser, der diese neuen Typen nicht kennt, automatisch auf einen sinnvollen Default zurückfällt (<input type=text>). Der Fallback selbst wurde schon in HTML Version 2 [HTML2] vorgesehen und ermöglicht somit erst die neuen Erweiterungen.

Ähnliche Mechanismen greifen auch bei neuen Elementen wie dem <video>-Element. Unterstützt der Browser dieses, blendet er an dieser Stelle das entsprechende Video ein, ignoriert aber in dem Element enthaltene weitere Elemente. Ein Browser, der ein Element nicht kennt, muss dieses gemäß Spezifikation als Inline-Element behandeln. Ohne weitere spezielle Style-Sheets wird damit einfach der Inhalt des Elements angezeigt. Im Falle des <video>-Elements könnte (und sollte) dies z. B. ein Flash-Video sein.

Das Prinzip des Progressive Enhancement gilt ebenso für CSS: Ein Browser, der eine CSS-Angabe nicht versteht, wird diese ignorieren. Damit ist es möglich, ein gemeinsames Stylesheet zu verfassen, welches einen Inhalt mit ein und demselben Stylesheet in verschiedensten Browsern entsprechend ihrer CSS-Fähigkeiten formatiert.

Die Idee, Inhalte so auszuliefern, dass zunächst nur so wenig wie möglich vorausgesetzt und dann abhängig von den Fähigkeiten des Clients ergänzt wird, ist einer der Kerngrundsätze des ROCA-Stils. Dies bedeutet auch, dass alle fachlichen Inhalte so präsentiert werden, dass sie unabhängig von den Fähigkeiten des Clients noch vollständig nutzbar sind. Progressive Enhancement sollte innerhalb des ROCA-Stils so genutzt werden, dass hiervon weitgehend nur Komfort und Hilfsfunktionen betroffen sind.

Semantic HTML

HTML als Auszeichnungssprache dient in vielen Web-Frameworks eher als reiner Datencontainer denn als Strukturierungswerkzeug für die auszuliefernden Inhalte. Die semantische Bedeutung der Datenstruktur des Inhalts verliert sich in einer endlosen Schachtelung von -Elementen oder wird durch den Missbrauch von Elementen zur Darstellung verwaschen. Einen ganz extremen Weg gehen üblicherweise Single Page-Apps, die auf serverseitiges HTML vollständig verzichten und den Server als reine Datenquelle betrachten, die Inhalte per JSON an den intelligenten Client liefert.

Ähnlich wie „POJO“ (Plain Old Java Object) im Java-Umfeld nach den Komplexitätsorgien einiger Frameworks eine Rückbesinnung auf das Simple darstellen, bezeichnet „POSH“ (Plain Old Semantic HTML) [POSH] die Idee, dass sich HTML-Markup hervorragend dazu eignet, Inhalte zu strukturieren – ganz ohne sich um deren Visualisierung zu kümmern.

Hieraus folgt, dass HTML-Elemente, die rein repräsentative Zwecke haben (<font>, <i>, <b>), zu Gunsten von semantischen Elementen (<h1>, ,) zu vermeiden sind. Nach diesem Muster sauber strukturierte HTML-Seiten laden schneller, sind einfacher zu schreiben, zu verstehen und besser wiederzuverwenden. Darüber hinaus verhilft valides und semantisch sinnvolles Markup zu einem konsistenteren und fehlerärmeren Verhalten in den aufliegenden Schichten CSS und JavaScript.

Eine maschinelle Verarbeitung von HTML-Inhalten, wie zum Beispiel die Interpretation von zusätzlichen semantischen Informationen durch den Google Crawler, die dann für die strukturierte Anzeige von Suchergebnissen verwendet werden, illustriert das Potenzial von HTML in der Maschine-zu-Maschine-Kommunikation.

Zusammen mit dem oben beschriebenen Gedanken, dass eine URI eine Hauptressource identifiziert, ergibt sich ein ganz praktischer Nutzen für alle, die mit einer langsamen Netzverbindung oder ausgeschaltetem JavaScript unterwegs sind: Das Wesentliche – der eigentliche Inhalt – wird als Erstes angezeigt, anstelle eines Rahmens, der den Inhalt erst nachlädt.

CSS

Das zweite Standbein einer sauberen Architektur auf Client-Seite bildet eine mittels CSS sauber separierte Präsentationsschicht. Nahezu unabhängig vom strukturellen Format der Daten kann mit CSS eine beliebige Darstellung für den menschlichen Benutzer erstellt werden. Zum Repertoire gehören hier neben den Stärken im Layout auch ausgeprägte Möglichkeiten für Typographie und Satz sowie in modernen Browsern auch neue Fähigkeiten wie Animationen und Zeichnungen.

Die Separation der visuellen Darstellung ist eines der Kernkonzepte von ROCA-basierten Webanwendungen, um eine möglichst lose Kopplung zu den konkreten Daten zu schaffen. Änderungen im strukturellen Aufbau sollten möglichst wenig Seiteneffekte zu den implementierten Stilen haben und umgekehrt sollen Änderungen in der visuellen Repräsentation natürlich keine Auswirkungen auf die fachlichen Informationen aufweisen.

Die Kopplung von CSS an HTML-Komponenten erfolgt über CSS-Selektoren wie IDs, Klassen und Attribute. Hierdurch kann sehr feingranular reguliert werden, welche Stile in welchem Kontext eine Anwendung finden. Die Anwendung von Inline-Stilen ist im ROCA-Kontext nur durch die Manipulation aus dem JavaScript heraus erlaubt, wenn JavaScript-Verhaltensweisen Einfluss auf die optische Darstellung von CSS-Komponenten ausüben.

In der Regel sollte dies aber über die Manipulation von Klassen erfolgen und nur bei Ausnahmen durch konkrete Wertemanipulation von Eigenschaften in Inline-Stilen. Ein Beispiel wäre hier die pixelgenaue Positionierung eines Elements zu einem anderen, bei dem die konkreten Koordinaten nicht in einer Klasse abstrahiert werden können.

Neue CSS-Regeln und -Eigenschaften werden ähnlich rückwärtskompatibel erstellt, wie es bei neuen HTML-Elementen der Fall ist, sodass bei der Verwendung von CSS-Eigenschaften davon ausgegangen werden kann, dass in der Regel der Inhalt vollumfänglich erhalten bleibt, auch wenn der Client nur eine Teilmenge der Eigenschaften versteht.

Die Rolle von JavaScript

JavaScript stellt die dritte und aktuell wohl meistbeachtete Säule des menschenlesbaren Webs dar. So gut wie jeder aktuell im Umlauf befindliche Browser wird mit seinen JavaScript-Fähigkeiten beworben, insbesondere der Ablaufgeschwindigkeit, zu der in den letzten Jahren ein regelrechtes Wettrüsten zwischen den Anbietern stattgefunden hat (sehr zur Freude von Entwicklern und Anwendern).

Die Verwendung von JavaScript auf dem Client in Verbindung mit einem Server, der Applikationslogik beinhaltet, erzeugt zunächst einmal ein klassisches Client/Server-Szenario. Dies wird aber dadurch erschwert, dass der Entwickler des Servers i. d. R. keinerlei Informationen oder Kontrolle über den Client hat. Schlimmer noch: Der Entwickler muss im klassischen byzantinischen Sinne [BYZ] sogar davon ausgehen, dass der Client kompromittierend verwendet wird und genau das tut, was man an dieser Stelle eigentlich nicht will.

Neben dem definitiv nicht ROCA-konformen Ansatz vieler Web-Frameworks, diese Client-Server-Problematik vor dem Entwickler zu verstecken, gibt es in vielen heute verfügbare Webangeboten die Situation, dass die Problematik einfach weitgehend ignoriert wird. So ist oft zu beobachten, dass die Funktionsfähigkeit (und ggf. schlimmer noch die Sicherheit) des Webangebots vom Client abhängig ist.

Analog wird oft gegen fundamentale Prinzipien der Softwareentwicklung wie die Redundanzfreiheit verstoßen und die gleiche Funktionalität sowohl server- als auch clientseitig implementiert. Daher ist eine zentrale Forderung des ROCA-Ansatzes auch der Verzicht auf die Duplikation von Applikationslogik auf dem Client.

Nimmt man diese Forderung mit der ROCA-Anforderung nach vollständiger serverseitigen Applikationslogik zusammen, so folgt daraus, dass keinerlei Applikations-logik in JavaScript vorliegen darf. Dies wirft in Diskussionen über den ROCA-Stil eigentlich immer die exemplarische Frage nach Validierungen auf: „Wie sollen clientseitige Validierungen, welche natürlich Server-Roundtrips und damit Reaktionszeit sparen, umgesetzt werden?“

Zur Beantwortung dieser Frage muss eine wichtige Unterscheidung zwischen zwei Typen von Validierungen gemacht werden: *datenbasierte* und *logikbasierte Validierungen*.

Datenbasierte Validierungen sind deklarativ. Beispiele für eine solche Validierung sind die Überprüfung, ob ein Wert für ein Attribut vorhanden ist oder die Überprüfung des Formates eines Wertes anhand eines regulären Ausdrucks. Dieser reguläre Ausdruck ist ein Datenwert, welcher an das zu überprüfende Attribut annotiert ist und dadurch auch an ein entsprechendes Eingabefeld annotiert werden kann.

Per JavaScript kann nun eine generalisierte Funktionalität zur Verfügung gestellt werden, die eine Überprüfung für Eingabefelder implementiert, die eine solche Annotation aufweisen. Dieses JavaScript ist vergleichsweise einfach und absolut unabhängig von der eigentlichen Applikationslogik zu implementieren. So ist diese Funktionalität mittlerweile auch Bestandteil von modernen Browsern in Form von HTML5-Attribute `pattern` und `required`.

Ob diese Attribute unterstützt werden, kann für ältere Browser mithilfe von JavaScript überprüft werden [MOD]. Hier kann dann mit JavaScript eine manuelle Implementierung bereitgestellt werden. Datenbasierte Validierungen sind aus ROCA-Sicht völlig in Ordnung: Sie setzen keine Duplikation von anwendungsspezifischer Logik voraus.

Der andere Typ von Validierungen sind *logikbasierte Validierungen*, also vom Entwickler geschriebene Funktionalität, welche zur Überprüfung der Korrektheit der Eingaben ausgeführt wird und anwendungsspezifisch ist. Ein Beispiel könnte eine Überprüfung darauf sein, dass ein bestimmtes Eingabefeld ausgefüllt sein muss, wenn ein anderes Feld ausgefüllt ist.

Solche Validierungen nehmen üblicherweise über kurz oder lang mehr und mehr Komplexität an. Genau dies ist auch der Grund, warum eine Duplikation solcher Logik generell keine gute Idee ist: Die Wahrscheinlichkeit, dass die serverseitige Implementierung der Validierung nicht mehr mit der clientseitigen übereinstimmt, ist im Falle der Verwendung einer anderen Programmiersprache als JavaScript auf dem Server von vornherein nicht zu vernachlässigen und nimmt im Laufe der Weiterentwicklung der Anwendung sicherlich noch deutlich zu.

Verzichtet man also clientseitig auf solche Validierungen, heißt dies aber nicht, dass zur Überprüfung der Korrektheit von Dateneingaben immer zwangsläufig das jeweilige Formular abgeschickt werden muss. Dies kann z. B. auch unter Verwendung von Ajax automatisch im Hintergrund geschehen, indem der aktuelle Zustand des Formulars vorweg an den Server geschickt wird, dieser die Eingaben validiert und entsprechend mit einer Liste der fehlerhaften Eingaben antwortet. Diese Fehler können dann mittels eines ebenfalls von der Applikationslogik unabhängigem JavaScripts an den jeweiligen Eingabefeldern angezeigt werden.

Neben der Duplikation von Validierungslogik gibt es eine weitere Art von Abhängigkeit zwischen Client und Server, welche oft unnötigerweise in Webanwendungen eingebaut wird. So ist oft zu beobachten, dass das JavaScript versucht, Ausgaben des Servers nachzuahmen und umgekehrt. Soll z. B. eine vom Server erstellte Liste von Einträgen um einen weiteren neuen Eintrag erweitert werden, ohne dass die gesamte Liste neu geladen werden muss, so wird hierfür üblicherweise Ajax verwendet.

Dagegen ist grundsätzlich nichts einzuwenden. Oft wird dies allerdings so umgesetzt, dass der Server auf das Absenden des Formulars zur Anlage des Eintrags mit einem JSON-Schnipsel oder gar nur mit einem „Ok“ antwortet. Der Client sorgt dann dafür, dass eine entsprechende HTML-Struktur für den Eintrag erstellt und an die Liste angehängt wird.

Dieses Vorgehen stellt eine deutliche Vermischung von Client- und Server-Implementierung dar, welche in diesem Beispiel einfach dadurch vermieden werden könnte, dass der Server mit dem entsprechenden HTML-Schnipsel antworten sollte, das der Client dann nur noch an die Liste anhängen müsste.

Ein sehr generischer Ansatz, um Abhängigkeiten zwischen Client- und Server-Implementierungen zu vermeiden, ist *Unobtrusive („unaufdringliches“) JavaScript*. So ist die Anforderung, JavaScript ausschließlich nach diesem Ansatz zu entwickeln, eine zentrale ROCA-Forderung.

Unobtrusive JavaScript

Spricht man vom Layout von Webseiten, hat sich im letzen Jahrzehnt der nahezu allgemein akzeptierte Konsens entwickelt, dass HTML frei von Layoutinformationen sein sollte und diese ausschließlich in Form von CSS vorliegen sollten. Unobtrusive JavaScript bedeutet – vereinfacht gesagt – nichts nichts anderes, als dieses Prinzip auch auf das Verhalten zu übertragen. So erweitert JavaScript in diesem Sinne die Informationen und Struktur, welche der Ser-

ver in Form von HTML zur Verfügung stellt, um clientseitige Funktionalität.

Dies soll ebenfalls nach dem Prinzip des Progressive Enhancement erfolgen: JavaScript ergänzt die Funktionalitäten genau dann, wenn der Browser dies unterstützt. Unterstützt der Browser sie nicht, weil beispielsweise die Ausführung von JavaScript deaktiviert ist, muss die Funktionalität der Anwendung trotzdem noch nutzbar sein. Abstriche an der Ergonomie sind dabei akzeptabel.

Webseiten auch mit deaktiviertem JavaScript nutzen zu wollen, mag wie ein Anachronismus erscheinen. Es hilft jedoch, dabei nicht nur an bürokratische Regeln in konservativen Unternehmen zu denken, sondern auch an andere User Agents, wie z. B. Screen Reader oder den Google Crawler, die eine Seite allein aufgrund des strukturierten Markups interpretieren können sollten.

Aus Architektursicht viel wichtiger ist jedoch, dass sich bei diesem Ansatz eine insgesamt andere und unserer Überzeugung nach bessere Architektur ergibt: Die Prinzipien des Webs sowie das Konzept von serverseitiger Logik werden eingehalten und die Verknüpfbarkeit unterstützt – Aspekte, die bei einer Implementierung des vollständigen Clients in JavaScript auf der Strecke bleiben würden. Darüber hinaus führt die Verwendung von Unobtrusive JavaScript nicht zuletzt durch die Vermeidung von Abhängigkeiten zwischen Client und Server zu deutlich wartbarerem Code in kürzerer Zeit.

Was nützt jedoch die schönste Architektur, wenn Endanwender unzufrieden sind, weil sie vor einer Anwendung sitzen, die nicht ihren Erwartungen entspricht? Das wäre ein überzeugendes Argument gegen ROCA, wenn es korrekt wäre. Denn auch mit diesem Ansatz lassen sich genauso ergonomische und moderne Anwendungen realisieren wie mit den im Moment im Hype befindlichen Alternativen und dies sogar ohne dass dafür irgendein Mehraufwand notwenig wäre.

ROCA verbietet den Einsatz von JavaScript nicht, legt dafür aber eine klare Rolle fest. Zum einen wird es verwendet, um Komponenten zu implementieren. Dabei kann es sich um die typischen handeln, die man aus gängigen UI-Bibliotheken anderer Plattformen kennt, wie z. B. Tabellen, Tab-Controls, Bäume usw. Diese werden von einem kleinen Anteil JavaScript-Code (dem JavaScript-Glue-Code) erzeugt und dabei mit den entsprechenden Elementen aus dem HTML verknüpft.

Die Selektion des entsprechenden HTML-Elements erfolgt meist mittels eines JavaScript DOM-Frameworks oft über CSS-Selektoren. Ein gutes Beispiel für die Implementierung einer solchen Vorgehensweise sieht man zum Beispiel bei der jQuery-UI Tab-Komponente [JQUI].

Fazit

Klassische Webarchitekturen, bei denen die Geschäftslogik auf einem Server liegt, der als Antwort auf HTTP-Anfragen HTML erzeugt, sind alles andere als Schnee von gestern. Im Gegenteil: Der Schlüssel dazu besteht darin, den Browser zu nutzen, anstatt gegen ihn anzukämpfen – mit einem REST-konformen Backend, semantischem HTML, CSS und „unobtrusive“ JavaScript.

Mit diesem Ansatz lassen sich hochmoderne, ergonomische Webanwendungen entwickeln, die sich in die Architektur des Webs einfügen und so sowohl innerhalb von Unternehmensgrenzen als auch unternehmensübergreifend von den Vorteilen profitieren, die sich daraus ergeben. ■

Referenzen

[ROCA] http://roca-style.org
[REST] http://bit.ly/TmvqcO
[HTML2] http://tools.ietf.org/html/rfc1866#section-8.1.2.1
[POSH] http://microformats.org/wiki/posh
[BYZ] http://de.wikipedia.org/wiki/Byzantinischer_Fehler
[MOD] http://modernizr.com/
[JQUI] http://jqueryui.com/tabs/