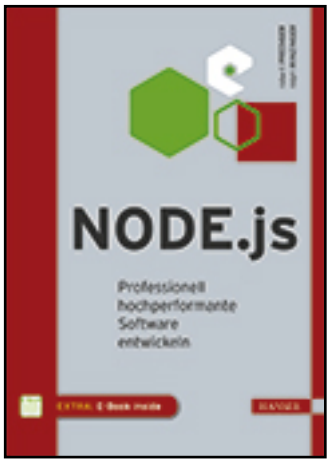
HANSER



Leseprobe

Robert Prediger, Ralph Winzinger

Node.js

Professionell hochperformante Software entwickeln

ISBN (Buch): 978-3-446-43722-7

ISBN (E-Book): 978-3-446-43758-6

Weitere Informationen oder Bestellungen unter

http://www.hanser-fachbuch.de/978-3-446-43722-7

sowie im Buchhandel.

Inhalt

Vorwort

Als Erstes möchten wir uns an dieser Stelle für Ihr Interesse an unserem Buch bedanken und die Gelegenheit nutzen, ein paar Worte über unser Buch zu verlieren. Das heißt, wir wollen uns, die Autoren, kurz vorstellen und unsere Motivation erklären, dieses Buch zu schreiben. Außerdem gibt es natürlich das eine oder andere über den Aufbau und das Format zu sagen, über die enthaltenen Beispiele und über JavaScript.

Über die Autoren ...

Es war von Anfang an klar, dass dieses Buch ein Gemeinschaftsprojekt sein soll. JavaScript – und damit auch Node.js – leben mit dem Ruf, keine professionelle Entwicklungsumgebung darzustellen. Insbesondere stößt man in der „Enterprise"-Welt auf diese Meinung. Um das objektiv zu hinterfragen, braucht es aber Beitragende aus beiden Lagern. So sind wir – Robert und Ralph – letztendlich aufeinandergetroffen, der eine seit erstaunlich langer Zeit professionell im Node.js-Umfeld tätig, der andere seit vielen Jahren als Software-Architekt in der Java-Enterprise-Welt unterwegs. Einer, der eine gewisse Erwartungshaltung mitbringt, der andere, der diese Erwartungshaltung mit den richtigen Bibliotheken, Tools und Methoden adressieren muss. Für uns war es eine fruchtbare Zusammenarbeit und bedeutete obendrein noch großen Spaß. Wir hoffen, beides in den kommenden Kapiteln weitergeben zu können.

Robert Prediger

Ich bin Robert, der Node.js-Freak. Mein größtes Projekt vor meiner Node.js-Zeit war die Entwicklung eines Accounting-Systems für internationale Hotelketten in Progress. Mit der Erstellung von Webapplikationen beschäftige ich mich seit gut 15 Jahren.

Vor knapp vier Jahren bin ich auf Node.js gestoßen und mich hat die Umgebung von Anfang an, vor allem aufgrund ihrer Stabilität und Geschwindigkeit, überzeugt und fasziniert. Mittlerweile sind diverse Projekte mit Node.js im Rahmen meiner web4biz Consulting ans Laufen gebracht worden.

Seit kurzem bin ich als Co-Founder an der whogloo Inc. beteiligt, deren Ziel es ist, eine Plattform zur Erstellung von Enterprise-Business-Applikationen zu erstellen – auf Basis von Node.js natürlich.

So habe ich in meinem täglichen Umfeld permanent mit Node.js zu tun. Das ist anstrengend, denn die Technik ist nach wie vor neu, teilweise bereits den Kinderschuhen entwachsen, dennoch sehr lernintensiv, wenn man permanent auf dem Laufenden bleiben will. Und das muss man auch, denn die Entwicklung in diesem Umfeld ist rasend schnell. Aber bereut habe ich es bisher nicht, mich komplett auf Node.js einzulassen. Es macht immer wieder Spaß und es fasziniert stets aufs Neue, mit welch teilweise einfachen Mitteln man Programme entwickeln kann, die auch einem hohen Standard genügen.

Und ich bin gespannt, wo uns die Reise mit Node.js noch hinführt.

Ralph Winzinger

Ich bin Ralph, der Java-Architekt. Ich arbeite für Senacor Technologies, ein Beratungsunternehmen mit vielen großen Kunden aus der Finanzwelt, bei denen ich Architekturen geplant, Entwicklungsprozesse eingeführt und Software erstellt habe, gerne auch mal auf der Frontend-Seite, aber hauptsächlich im Backend. EJBs, Webservices, Spring, JPA … das ist in der Regel meine Welt. Und zugegeben, ein großer Freund von JavaScript war ich lange Zeit nicht. Ich glaube, mich noch ungefähr an meine erste Reaktion auf Node.js erinnern zu können: „Serverside JavaScript? Wer braucht denn bitte eine Alert-Box auf dem Server?!?“ Wie gesagt, das war die allererste, spontane Reaktion.

Ich bin aber nicht mit der Java-Enterprise-Welt verheiratet. Es macht mir Spaß, neue Technologien zu ergründen und mir so manche Nacht mit Sourcecode um die Ohren zu schlagen. So ist es nicht verwunderlich, dass mich der Charakter von Node.js bald fasziniert hat.

„High-Scalability“ ist ein Thema, das auch im Enterprise-Umfeld immer wichtiger wird. Eine riesige Zahl von mobilen Endgeräten, die in jeder Branche zu einem extrem wichtigen Kundenkanal geworden sind, und die Entwicklung im Feld von „Internet of Things“ verbieten es geradezu, sich weiter exklusiv auf traditionelle Java Application Container und die damit verbundenen Technologien zu verlassen.

Ob sich ausgerechnet Node.js in meinen Projekten und bei meinen Kunden jemals etablieren wird, ist derzeit noch schwer zu beurteilen. Aber den Konzepten von Node.js wird sich unsere Branche sicherlich nicht verschließen können. Und ich hätte nun meine Meinung zum professionellen Einsatz von JavaScript.

… ihre Helfer …

Zum Buch haben nur wir beide beigetragen? Nein. Eigentlich sollten vier Namen auf dem Cover stehen. Zwei geschätzte Kollegen und sehr gute Freunde haben diese Erkundungstour mit uns gestartet. Teils aus privaten, teils aus beruflichen Gründen mussten aber beide unser Buchprojekt verlassen, bevor wir es zum Abschluss bringen konnten. Trotzdem haben sie wichtige Impulse gegeben und auch wichtige Inhalte beigetragen. Wir sprechen von Victor Volle und Charles-Tom Kalleppally, beide in der Java-Welt verwurzelt, sehr versiert im Umgang mit Architektur, Design und Code und ebenfalls immer bereit, ausgetretene Pfade zu verlassen und sich auch mal neuen Technologien zuzuwenden.

An dieser Stelle großen Dank an Victor und Charly!

… und ihre Motivation

Unsere Motivation haben wir in den einleitenden Sätzen ja schon skizziert: Der Node.js-Mensch wollte eine Lanze für seine Technologie brechen und der Java-Mensch „mal was Verrücktes tun“ – mit dem Ziel, die Entwicklung mit Node.js und JavaScript ein wenig ins rechte Licht zu rücken.

Wir haben die Maßstäbe aus der Java-Enterprise-Entwicklung Node.js und JavaScript angewandt, sowohl technisch als auch methodisch. Toolunterstützung für eine effiziente Arbeitsweise, Qualitätssicherung, um höchsten Ansprüchen zu genügen, State-of-the-Art Deployment und Monitoring … all das haben wir uns angesehen und die aus unserer Sicht und zum aktuellen Zeitpunkt besten oder vielversprechendsten Ansätze aus dem Node.js-Umfeld zusammengetragen.

Wir sind der Meinung, dass es für Node.js oder verwandte Technologien auch im Enterprise-Umfeld Bedarf gibt und dass sich Node hier durchaus verwenden lässt. Nichtsdestotrotz bleibt es zu einem gewissen Teil vorerst noch Kopfsache, ob man Node.js und JavaScript das nötige Vertrauen schenkt, geschäftskritische Teile der Systemlandschaft zu realisieren.

Das Zielpublikum

Wir richten uns mit diesem Buch ganz klar an Entwickler und entwickelnde Architekten. Es ist ein sehr technisches Buch und wir gehen an vielen Stellen davon aus, dass entsprechendes Wissen vorhanden ist, um den einen oder anderen Sachverhalt zu verstehen.

Es werden vor allem diejenigen Freude am Buch haben, die nicht einfach nur möglichst viel Code in möglichst kurzer Zeit produzieren wollen. Das Design des Codes, die Paradigmen der Laufzeitumgebung, aber auch der Betrieb von Enterprise-Software sind uns wichtig und sollten es auch unseren Lesern sein.

Und nicht zuletzt wünschen wir uns ein wenig Spieltrieb. JavaScript ist eine sehr flexible Sprache, Node.js ein sehr schnell wachsendes Ökosystem, in dem es immer wieder Neues zu entdecken gibt. Man kann es sich hier noch viel weniger leisten, den Blick für die aktuelle Entwicklung und aktuelle Trends zu verlieren, als das im eher trägen Enterprise-Umfeld der Fall ist.

Die Arbeit mit JavaScript, mit seinem dynamischen Typsystem und den offenen Sourcecodes führt auch ganz natürlich dazu, dass man sich immer wieder in den Tiefen von fremdem Code verliert, um herauszufinden, wie eine Bibliothek funktioniert, welche versteckten Optionen eine Funktion eventuell noch bietet.

Das alles muss man mögen. Wir mögen es und wir hoffen, unsere Leser ebenfalls.

Das Buch

Es ist nicht nur wichtig, wer ein Buch verfasst hat und weshalb ein Buch geschrieben wurde. Nachdem wir einen bestimmten internen Aufbau verfolgt haben und auch über die verwendete Formatierung die Verständlichkeit erhöhen wollten, möchten wir unsere Richtlinien an dieser Stelle kurz skizzieren.

Aufbau

Wir haben unser Buch in drei Teile gegliedert.

Der erste Teil soll dafür sorgen, dass sich unsere Leser langsam an die Welt von Node.js gewöhnen. Die Geschichte, die Idee, das Tooling stehen hier im Vordergrund. Die Entwicklung an sich ist noch kein Thema, allenfalls am Rande.

Im zweiten und weitaus größten Teil bewegen wir uns durch die Schichten und Problemfelder einer Enterprise-Anwendung. Angefangen beim Dateisystem über Webanwendungen und Service-Schnittstellen bis hin zu einer ausgefeilten Testunterstützung werden die typischen Fragestellungen anhand von vielen vorgestellten Modulen und Codebeispielen beantwortet.

Der letzte Teil kümmert sich im Wesentlichen um die Zeit nach der Entwicklung. Wie nehme ich eine solche Anwendung in Betrieb und wie halte ich sie in Betrieb? Deployment, Monitoring oder auch Performance sind die Themen in diesem Teil. Und nicht zuletzt ein Blick auf eine Alternative – ähnliche Konzepte und Performance, aber nicht (nur) JavaScript.

Formate

Bezüglich der verwendeten Formate haben wir uns zurückgehalten. Natürlich sind alle coderelevanten Passagen als solche zu erkennen. Schreiben wir beispielsweise im Text über Code, so ist das wie hier `console.log("this is code")` entsprechend gekennzeichnet. Längere Codepassagen sind natürlich nicht im Fließtext untergebracht, sondern erhalten ihren eigenen Abschnitt:

```
var fs = require("fs");
var i = 5;
console.log("this is still code")
```

Auf dieselbe Art sind auch Kommandozeileninteraktionen formatiert. Auch diese können im Fließtext (`npm install express`) oder in einem größeren Block zu finden sein.

```
$ node app.js
INFO: this is some console output
```

Und dann sind da noch Modulangaben. Beziehen wir uns im Text auf Module wie *Express* oder *restify*, so sind diese immer kursiv dargestellt. Manchmal gibt es in diesem Zusammenhang auch eine gewisse Grauzone. So kann es sein, dass man von dem Modul *npm* spricht oder aber von dem Kommando `npm`. Damit ist dasselbe Wort dann vielleicht auch im selben Abschnitt verschieden formatiert.

Beispiele

Die allermeisten unserer Beispiele sind tatsächlich funktionierender Code, der seine Korrektheit zunächst in einem Test oder wenigstens durch eine Ausführung unter Beweis stellen musste. Erst dann durfte er aus der IDE über copy & paste ins Buch springen. Natürlich können auch dabei immer wieder mal Fehler passieren, die bitten wir zu entschuldigen.

Die Codebeispiele

Die Codebeispiele in diesem Buch müssen natürlich nicht abgetippt werden; sie stehen Ihnen online auf einem GitHub Repository zur Verfügung:

https://github.com/WinzingerPrediger/node.js

Die Welt von JavaScript

Eines liegt uns noch sehr am Herzen. Wie erwähnt, ist die Welt von JavaScript und Node.js sehr kurzlebig. Ein Modul, das heute noch *das* Modul für eine bestimmte Problemstellung ist, wird morgen vielleicht schon von einem neuen Modul abgelöst, welches riesigen Zuspruch in der Community erhält und schnell zum De-facto-Standard wird.

Natürlich stellen wir im Buch ganz konkrete Module vor und laufen damit potenziell Gefahr, dass diese einige Monate nach dem Erscheinungstermin des Buchs schon nicht mehr existieren oder niemanden mehr interessieren. Wir haben aber auch immer versucht, die aus unserer Sicht relevante Funktionalität herauszustellen – welches Problem wird von dem Modul eigentlich gelöst? Mit diesen Hintergrundinformationen sollte es einfacher sein, die Eignung neuer Module zu hinterfragen oder gezielt nach Alternativen zu suchen.

„Lerne ich hier JavaScript?“

Diese Frage lässt sich einfach beantworten: nein. Vielleicht kann man sich an der einen oder anderen Stelle einen kleinen Kniff abschauen, aber wir geben hier noch nicht mal eine gezielte Einführung in JavaScript.

Es gibt sehr viel Literatur und sehr viele Quellen im Internet, die das leisten können. Wir haben uns das nicht zum Ziel gesetzt.

Viel Spaß beim Lesen, Lernen und Spielen,

Robert und Ralph

2.6 Node.js und Webservices

In den letzten Jahren hat das Netz massiv Einzug in alle unsere Anwendungen gehalten und letztendlich zu einem Paradigmenwechsel im Bereich der Softwarearchitektur geführt. Auch wenn anfänglich hauptsächlich Enterprise-Applikationen über Business-to-Business-Schnittstellen mit anderen Servern kommunizierten, so haben Endkundenanwendungen hier schnell aufgeholt. Nachdem sich im ersten Schritt das Internet auf privaten PCs etabliert hatte, wurden aus statischen Webseiten im Zuge von Web 2.0 bald interaktive, netzwerkgestützte Anwendungen und damit die Vernetzung im privaten Bereich zum Normalfall. Dieser Trend hat sich auch in Mobilfunkgeräten und -netzen fortgesetzt, so dass heute angenommen werden darf, dass Anwender immer und überall online sind – nicht nur mit normalen Computern, sondern auch mit Millionen mobilen Endgeräten.

Diese Entwicklung hat dazu geführt, dass unsere Anwendungslandschaften immer verteilter werden: Daten werden in Clouds ausgelagert, Berechnungen werden auf dafür geeigneten Servern durchgeführt und Informationen werden angefordert, sobald sie benötigt werden.

Für uns als Anwendungsentwickler oder Softwarearchitekten bedeutet das, dass wir in der Lage sein müssen, die gängigen Konzepte zum Konsumieren und Publizieren von Services zu nutzen. Grundsätzlich ist das natürlich mit jeder Plattform möglich, die es uns erlaubt, eine Socket-Verbindung anzubieten beziehungsweise zu öffnen. Aber komfortabel und effizient wird die Netzwerkkommunikation erst, wenn uns entsprechende Module und Bibliotheken möglichst viel Arbeit abnehmen.

Wenn heute Anwendungen im Netzwerk kommunizieren, dann in der Regel mit Hilfe von Webservices. Im Enterprise- und Integrationsumfeld haben sich hier in den letzten zehn Jahren SOAP-Services etabliert, die mit einer Vielzahl von Spezifikationen und Standards aufwarten und de facto von allen Plattformen, Servern oder Programmiersprachen unterstützt werden. Daneben sind in jüngerer Vergangenheit die deutlich leichtgewichtigeren REST-Services entstanden. Diese sind zwar weniger formal definiert, dafür aber viel einfacher zu benutzen und sicherlich auch ressourcenschonender.

Egal, welcher der beiden Servicearten man persönlich den Vorzug gibt, man muss mit beiden arbeiten können, weil man oftmals keinen Einfluss auf die Gegenseite hat oder sich bestehenden Rahmenbedingungen unterordnen muss. In den folgenden Abschnitten wird deshalb die Unterstützung für SOAP- und für REST-Services näher betrachtet, aus Server- wie aus Client-Sicht. Dabei geht es nicht nur darum, zwei Parteien zu erzeugen, die Daten miteinander austauschen. Sobald man von Kommunikation in offenen Netzen spricht, darf man den Aspekt der Sicherheit natürlich nicht aus den Augen verlieren.

2.6.1 SOAP-Services

Wie bereits eingangs erwähnt, sind SOAP-Services weitreichend spezifiziert und standardisiert[64] und dabei alles andere als leichtgewichtig. Ursprünglich stand SOAP zwar für

[64] https://www.oasis-open.org/standards, http://www.w3.org/2002/ws/

„Simple Object Access Protocol“, aber von „simple“ kann hier auf Entwicklungsebene meistens nicht mehr die Rede sein. Selbst im Rahmen einfachster Services müssen die ausgetauschten XML-Nachrichten genau einem, vorab in WSDL (Webservice Description Language) festgelegtem, Format entsprechen – was auf der einen Seite Interoperabilität fördert, aber auf der anderen Seite mit großem Overhead einhergeht. Es müssen nicht nur die Daten aus dem eigenen Objektmodell in entsprechende XML-Strukturen überführt werden, sondern zudem muss auch noch angegeben werden, welche Operationen mit diesen Daten ausgeführt werden sollen. Die ausgetauschten Nachrichten für einen einfachen „Hello World“-Service sehen ungefähr wie folgt aus:

```
POST /HelloService HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:envelope>
  <soap:body>
    <sayHello>
      <name>John</name>
    </sayHello>
  </soap:body>
</soap:envelope>
```

und die zugehörige Antwort:

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<soap:envelope>
  <soap:body>
    <result>Hello, John!</result>
  </soap:body>
</soap:envelope>
```

Auch wenn dieses Beispiel noch übersichtlich ist, so kann man bereits schon erahnen, wie komplex diese Dokumente werden, sobald man realistisch große und verschachtelte Objektstrukturen austauscht.

Die gezeigten Dokumente beschränken sich auf das Mindeste, was benötigt wird, um Webservices benutzen zu können. Diese Vereinfachung ist zwar hilfreich, um einen Einstieg zu finden, aber verglichen mit dem alltäglichen Einsatz leider eher unrealistisch. Die ausgetauschten Dokumente müssen nicht nur den Funktionsaufruf oder das zugehörige Ergebnis enthalten, sondern meistens auch sogenannten Policies[65] gehorchen. Diese Policies legen fest, unter welchen Bedingungen die gewünschten Daten ausgetauscht werden:

- verschlüsselt,
- signiert,
- mit Zeitstempeln versehen,

[65] http://schemas.xmlsoap.org/ws/2004/09/policy/

- mit Authentifizierungsinformation versehen,
- …

Policies werden sowohl im Server konfiguriert als auch dem Konsumenten über die WSDL bekanntgemacht. Ignoriert ein Konsument eine oder mehrere verpflichtende Policies, so werden seine Anfragen lediglich mit Fehlermeldungen quittiert.

Um Webservices erfolgreich einsetzen zu können, müssen somit einerseits XML-Dokumente erzeugt und eingelesen und andererseits passende Policies eingefordert oder erfüllt werden können.

Eines sei an dieser Stelle schon vorweg gesagt: Den Grad an Unterstützung, den man beispielsweise aus dem Java-Enterprise-Umfeld gewohnt ist, wird Node.js nicht bieten – wenigstens momentan noch nicht. Node.js ist noch jung und täglich werden neue Module entwickelt oder bestehende verbessert. Gerade deshalb ist es aber wichtig, die Grenzen zu kennen, die einem durch die Plattform gesetzt werden. Allerdings sei an dieser Stelle die Prognose gewagt, dass sich SOAP-Services nicht zum Lieblingskind der Node.js-Umgebungen entwickeln werden. Die Zeichen stehen hier ganz eindeutig auf leichtgewichtige Techniken.

Nachfolgend sei davon ausgegangen, dass der sogenannte „top down"- oder „WSDL first"-Ansatz verfolgt wird. Das heißt, dass die formale Definition der Schnittstelle gesetzt ist und alle anderen Artefakte nachfolgend daraus abgeleitet werden. Ganz unabhängig von Node.js ist dieser Weg dem „bottom up"- oder „code first"-Verfahren vorzuziehen. Da bei „bottom up" die Schnittstellendefinition aus dem Code gewonnen wird, besteht die Gefahr, dass sich das Interface durch Änderungen am Code versehentlich ändert. Das mag in kleinen Projekten nicht weiter schlimm sein, in größeren Integrationsszenarien kann das jedoch zu einem politischen Desaster werden.

2.6.1.1 Von und nach SOAP

Die erste Herausforderung besteht darin, SOAP-Dokumente zu erzeugen beziehungsweise zu parsen. In der npm-Registry und auf GutHub finden sich hierzu verschiedene Module, die sich zur Verwendung eignen. Nachfolgend werden nicht nur die Module angesprochen, die das Handling von SOAP-Dokumenten weitestgehend automatisieren, sondern auch Module, die auf niedrigerem Level nur reines XML bearbeiten. Hierfür gibt es einen einfachen Grund: In Fällen, in denen die Funktionalität der verfügbaren SOAP-Module nicht ausreicht, um einen Service zu konsumieren oder anzubieten, besteht gegebenenfalls immer noch die Möglichkeit, die SOAP-Dokumente „zu Fuß" – also als einfaches XML – zu verarbeiten. Hierzu wurden an dieser Stelle drei Bibliotheken in die engere Auswahl genommen:

- *soap*[66] – ein Modul, das vollständige Unterstützung zum Publizieren und Konsumieren von Webservices bietet. Für einfache Services werden dem Entwickler fast alle Arbeiten abgenommen, so dass man sehr schnell zum ersten Service beziehungsweise Servicecall gelangt.
- *xml2json*[67] – ein Modul, welches zwischen JSON und XML konvertieren kann – nicht mehr, aber auch nicht weniger und sehr performant.

[66] https://www.npmjs.com/package/soap
[67] https://www.npmjs.com/package/xml2json

- *ws-js*[68] – ein Modul, welches weitergehende Webservice-Standards unterstützt. Neben einem Subset von WS-Security und WS-Adressing sei insbesondere MTOM erwähnt – ein Standard zum effizienten Austausch von Binärdaten.

Im weiteren Verlauf werden lediglich Services im sogenannten „document/literal"-Stil benutzt. Das sollte keine zu große Einschränkung darstellen, da inzwischen sowieso oftmals nur noch dieser Stil angetroffen wird. Sollte man einen anderen Typ publizieren oder konsumieren müssen, so wären die nachfolgend gezeigten Bibliotheken im Hinblick darauf aber nochmals zu prüfen. Eine sehr gute Aufstellung der verschiedenen Webservice-„Styles" kann man auf IBMs DeveloperWorks finden[69].

Die WSDL wird nicht von Node.js generiert, sondern anderweitig erstellt – manuell oder über geeignetes Tooling. Da sie keinerlei Besonderheiten enthält, wurde sie hier nicht mit abgedruckt.

2.6.1.1.1 Modul „soap"

Mit aktuell über 500 Downloads am Tag ist *soap* ein vergleichsweise populäres Modul und deshalb auch in diesem Abschnitt erste Wahl, um Webservices zu behandeln. Nachfolgend wird nun ein erster einfacher Service publiziert und auch ein Client dafür programmiert. Bei der Installation ist darauf zu achten, dass mit *node-expat*[70] eine Modulabhängigkeit zu nativen Betriebssystembibliotheken besteht. Das stellt in der Regel kein Problem dar, ist aber immer gut zu wissen, um gegebenenfalls auftretende Fehler besser einordnen zu können.

Exkurs: Webservices testen mit soapUI

Bevor die Beschreibung in die Tiefen der Implementierung von Webservice-Providern und Konsumenten eintaucht, soll an dieser Stelle das Thema Test und (SOAP-)Webservices betrachtet werden.

Sobald man sich in einem Umfeld bewegt, in dem Systeme integriert werden, können auch einfachste try-and-error-Tests nicht mehr ohne größeren Aufwand durchgeführt werden. Im Falle von SOAP-Services steht hier sofort das aufwendige Protokoll im Weg, das verhindert, dass ein publizierter Service „mal eben" aufgerufen oder ad-hoc ein Dummy-Service zur Verfügung gestellt wird.

An dieser Stelle soll deshalb kurz soapUI von SmartBears[71] vorgestellt werden: soapUI ist ein sehr umfassendes Test-Tool, mit dem Services diverser Protokolltechnologien – unter anderem auch die im vorliegenden Kontext relevanten Technologien SOAP und REST – getestet werden können. Ausgehend von einer WSDL können mit Hilfe weniger Mausklicks Request-Skelette generiert werden, um einen existierenden Service aufzurufen. Die Aufrufparameter werden komfortabel in die vorgenerierten XML-Strukturen eingetragen und der Request wird auf Knopfdruck abgesetzt. Abschließend kann man die eintreffende Antwort automatisch sowohl inhaltlich als auch im Hinblick auf Protokollkonformität prüfen lassen.

[68] https://www.npmjs.com/package/ws-js
[69] http://www.ibm.com/developerworks/library/ws-whichwsdl/
[70] https://www.npmjs.com/package/node-expat
[71] http://www.soapui.org/

Wird kein Server, sondern ein Client entwickelt, so kann soapUI hierfür auf Basis der WSDL einen entsprechenden Mock-Service bereitstellen, der mit vorab definierten Antworten auf die eingehenden Requests antwortet.

soapUI steht in einer kostenlosen Variante zum Download für Mac OS, Windows und Linux zur Verfügung. Darüber hinaus existieren noch Plug-ins für die Entwicklungsumgebungen Eclipse, Netbeans und Intellij IDEA. Abgerundet wird das Toolset durch die Integrationsmöglichkeit in Maven, welche letztendlich auch den Einsatz im Rahmen eines Continuous-Integration-Systems ermöglicht.

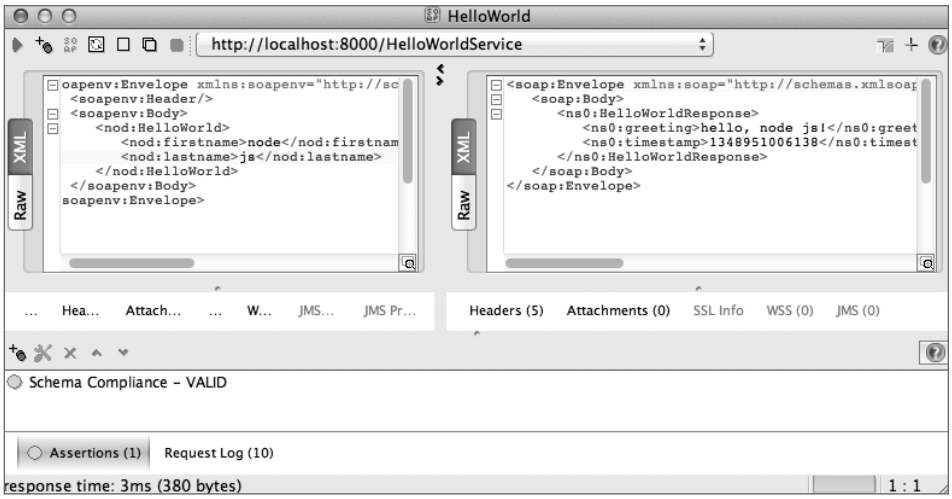


Bild 2.12 Schema-Validation mit soapUI

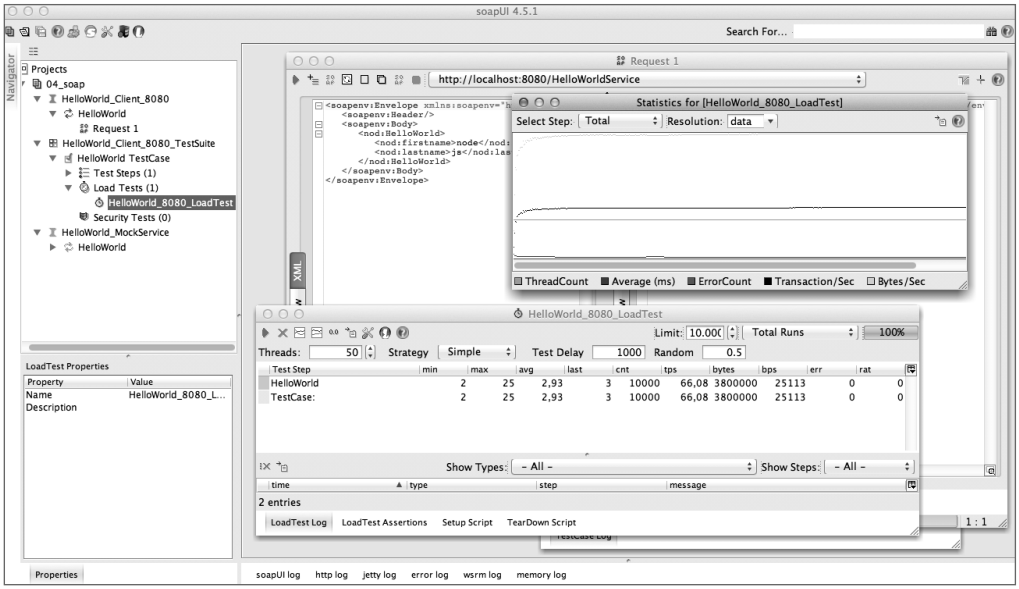


Bild 2.13 Lasttest mit soapUI

Neben soapUI bietet SmartBears zudem noch das ergänzende Tool loadUI an, mit dessen Hilfe man die in soapUI definierten Testfälle im Rahmen von Lasttests verwenden kann.

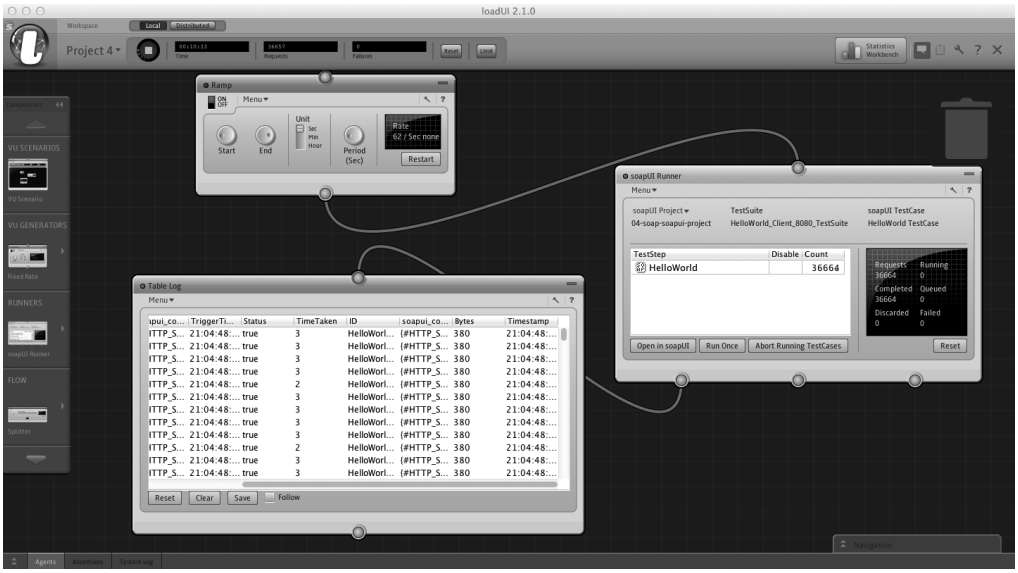


Bild 2.14 Lasttest mit loadUI

Grundsätzlich bleibt jedoch zu sagen, dass der Funktionsumfang von soapUI und loadUI immens ist und insofern gar nicht versucht werden soll, das Toolset detailliert zu beschreiben. Sollte man allerdings im Bereich SOAP-Services tätig sein und auf Qualitätssicherung Wert legen, so sei dringend empfohlen, soapUI und loadUI genauer in Augenschein zu nehmen und eventuell in die eigene Toolchain zu integrieren.

2.6.1.1.2 Hello, Node.js! – eine Beispielanwendung

Hello, Node js! – Server

Der erste Server wird mit wenigen Schritten und nicht viel mehr Zeilen Code up and running sein:

- einen HTTP-Server starten,
- einen SOAP-Handler mit passenden Callbacks erzeugen,
- den SOAP-Handler mit dem Server verknüpfen, so dass alle Requests, die der Server empfängt, vom Handler abgearbeitet werden.

Der HTTP-Server wird gestartet, wie es aus unzähligen „Hello, World!"-Beispielen bekannt ist: Das Modul *http* wird geladen, über `createServer()` wird eine Instanz erzeugt und diese lässt man auf einem Port seiner Wahl (in der Regel aus Berechtigungsgründen jenseits von 1024) lauschen. An dieser Stelle können ganz normale Callbacks für das Request-Handling installiert werden, allerdings werden Requests, die dem Namensschema der Webservices entsprechen, später immer vom SOAP-Handler übernommen. Grundsätzlich steht aber

einer Kombination mit anderen Modulen wie *node-static*[72] oder *Express* nichts im Wege. Das hat natürlich den Vorteil, dass eine komplette Anwendung mit statischen Inhalten, Front-end-Framework, Middleware und Webservices tatsächlich als eine einzige Anwendung entwickelt und installiert werden kann. Ob man das möchte, sei dahingestellt, der Trend geht ja sinnvollerweise eher in Richtung feingranulare Anwendungsstrukturen. Wenn man sich jedoch den Anspruch von Node.js ins Gedächtnis ruft, sehr viele Anfragen in einer einzelnen Instanz abwickeln zu können, macht das unter Umständen auch Sinn. Und je einfacher das Deployment-Szenario ist, desto einfacher sind auch die Konzepte für Failover und Clustering.

```
var http = require('http');
var server = http.createServer(function(request, response) {
  console.log("incoming (non-webservice) request: "+request.url);
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.end("some content\n");
});
server.listen(8000);
```

Wenn man sowieso nur Webservice-Requests behandeln möchte, könnte man den Requesthandler an dieser Stelle theoretisch auch komplett entfallen lassen. Allerdings werden dann Anfragen, die nicht dem Webservice-URL-Schema entsprechen, nicht verarbeitet und der Client wäre blockiert, bis er in einen Timeout läuft. Im Zeichen guten Stils empfiehlt es sich also, wenigstens mit einem HTTP-Code „404 Not Found“ zu antworten. Außerdem erscheint es für die Analyse eventueller Fehler ebenfalls sinnvoll, ein knappes Log-Statement – eventuell sogar als Warnung – abzusetzen, falls ein Request nicht vom Webservice-Handler abgearbeitet wird.

Nachdem nun ein Server gestartet werden kann, wird noch ein SOAP-Handler installiert:

```
var soap = require('soap');
soap.listen(server, '/HelloWorld', serviceCallback, wsdl);
```

Hierzu wird das Modul *soap* geladen und direkt die exportierte Funktion `listen()` aufgerufen. Diese erwartet als Parameter den HTTP-Server, einen Root-Context, die Service-Implementierung und natürlich die WSDL des Webservice. An dieser Stelle – und bei der Definition der Servicecallbacks – ist Vorsicht geboten: Die hier gewählten Namen müssen ihre Entsprechung in der WSDL haben, da es sonst nicht möglich ist, den eingehenden Request auf den Callback abzubilden. Für den Root-Context bedeutet das, dass hier der Wert des „location“-Attributs (hier „HelloWorldService“) aus dem Element `<soap:address/>` angegeben werden muss:

```
...
<wsdl:port name="HelloWorldSOAP12port_http"
           binding="ns0:HelloWorldSOAP12Binding">
  <soap12:address location="http://www.node-book.de/HelloWorldService"/>
</wsdl:port>
...
```

[72] https://www.npmjs.com/package/node-static

Der letzte Schritt, der einen noch vom lauffähigen Webservice trennt, ist die Einrichtung eines Callbacks, der den eingehenden Request bearbeiten soll. Eigentlich handelt es sich hierbei auch nicht um einen einzigen Callback, sondern um eine Struktur, die für alle in der WSDL definierten Operationen Callbacks zur Verfügung stellt. Die Struktur orientiert sich dabei an den Elementen, die in der WSDL auftreten:

```
{
   <service_name> : {
     <port_name> : {
       <operation_name> : function(args) {
           // do something
       }
     }
   }
}
```

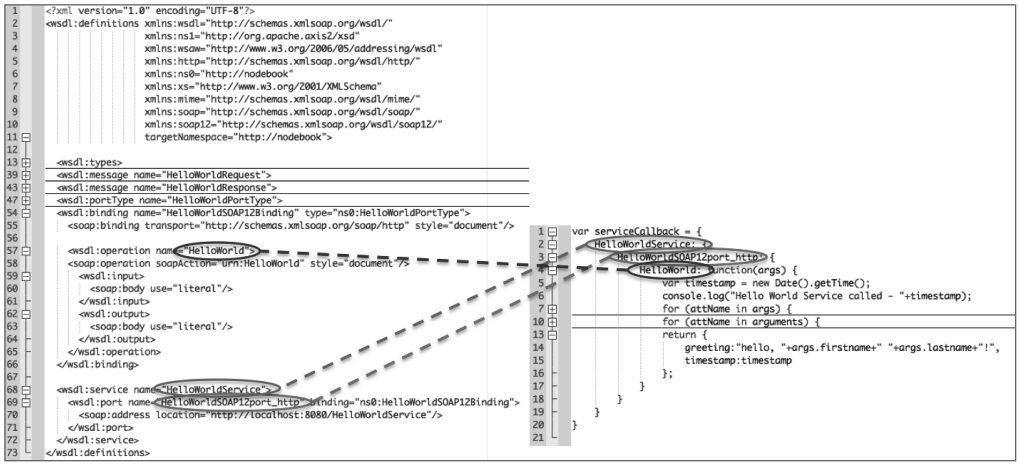


Bild 2.15 WSDL-JavaScript-Mapping

In einer WSDL können sich übrigens mehrere Ports zu einem Service und natürlich mehrere Operationen zu einem Port befinden. Entsprechend komplex kann die Callback-Struktur auch werden. Der übergebene Funktionsparameter `args` enthält die über den Webservice übertragenen Aufrufparameter nach JSON transferiert. In vorliegendem Beispiel sieht der SOAP-Aufruf wie folgt aus:

```
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
               xmlns:nod="http://nodebook">
  <soap:Header/>
  <soap:Body>
    <nod:HelloWorld>
      <nod:firstname>node</nod:firstname>
      <nod:lastname>js</nod:lastname>
    </nod:HelloWorld>
  </soap:Body>
</soap:Envelope>
```

Die Namespaces werden bei der Übersetzung nach JSON ignoriert, so dass letztendlich

```
args = {firstname:'node', lastname='js'}
```

entsteht und verarbeitet werden kann. Nach der Verarbeitung werden die Rückgabewerte analog aufbereitet. Die WSDL sieht für die Response eine Sequence aus zwei Elementen vor: greeting und timestamp. Das heißt, dass der Handler ein JSON-Dokument liefern muss, welches diese beiden Attribute besitzt:

```
retval = {greeting:'hello, node js!', timestamp='1348855315319'}
```

Und schon freut sich ein potenzieller Konsument über einen freundlichen Gruß …

Hello, Node js! – Client

Natürlich sollen Services nicht nur angeboten, sondern auch konsumiert werden. Das Modul *soap* kann auch in diesem Fall zur Seite stehen.

Zentrales Element ist, wie auch im vorangegangenen Abschnitt, die WSDL-Beschreibung des Service. Aus ihr wird ein Client erzeugt, mit dessen Hilfe anschließend die Operationen des Providers aufgerufen werden können. Aus Programmiersprachen wie Java ist man es gewöhnt, dass in einem vorgelagerten Prozess entsprechende Hilfsklassen erzeugt werden, die anschließend die API des Service im Code zur Verfügung stellen. Alternativ hierzu gibt es oftmals auch die Möglichkeit, komplett dynamisch zu arbeiten. In diesem Fall wird die WSDL erst zur Laufzeit eingelesen und mangels vorab generierter Klassen werden die Operationen per Reflection aufgerufen. Das bringt zwar weniger Overhead im Build mit sich, allerdings ist der Methodenaufruf via Reflection weder komfortabel noch elegant.

Da JavaScript eine dynamisch typisierte Sprache ist, kann man beide Vorteile vereinen: Ohne vorausgehenden Build-Schritt wird zur Laufzeit ein Client erzeugt. Anschließend stehen am Client-Objekt aber die tatsächlichen Methoden unmittelbar zur Verfügung und nicht nur ein mittelbarer Behelfsaufruf über Reflection – natürlich immer mit der Einschränkung, die JavaScript inhärent mit sich bringt, dass kaum eine IDE Hinweise geben wird, wenn die Webservice-Operationen oder eine API im Allgemeinen falsch aufgerufen werden.

```
var soap = require('soap');

var url = 'http://localhost:8088/mockHelloWorldService?wsdl';
var args = {firstname: 'node', lastname: 'js'};

soap.createClient(url, function(err, client) {
  if (!err) {
    client.HelloWorld(args, function(err, result) {
      if (!err) {
        console.log("greeting : "+result.greeting);
        console.log("timestamp: "+result.timestamp);
      }
    })
  }
});
```

Das obige Codebeispiel zeigt einen Webservice-Client, der einen über soapUI publizierten Mock-Service aufruft. Hierzu muss lediglich die WSDL in *soapUI* eingelesen und per Mausklick ein Service daraus erzeugt und gestartet werden. Jeder SOAP-Service liefert unter Angabe des Query-Parameters `?WSDL` seine eigene Beschreibung zurück. Genau diese Funktionalität nutzt das Modul *soap*, wenn ein Client-Objekt erzeugt wird: Zum einen wird die URL des Service übergeben, zum anderen eine Callback-Funktion, die aufgerufen wird, wenn das Client-Objekt bereit zur Nutzung ist. Das Client-Objekt selbst wird ebenfalls in diesem Callback übergeben, so dass man an ihm dann im letzten Schritt die eigentliche Operation aufrufen kann – auch das wieder mit einem Callback, so dass die Anwendung nicht blockiert, solange auf die Antwort des Service gewartet wird. Ein sehr wichtiges Detail, wenn man bedenkt, dass Services aufgerufen werden, die sich weder im eigenen Netzwerk noch unter eigener Kontrolle befinden.

Der Client kann aber natürlich nicht nur mit Hilfe einer vom Server geladenen WSDL erzeugt werden. Oftmals werden Servicebeschreibungen explizit ausgeliefert und clientseitig lokal gespeichert. Zudem ist der zusätzliche Netzwerkverkehr zum Abrufen der WSDL nicht wünschenswert, die Servicebeschreibung sollte sich ja bei solch „offenen“ Schnittstellen sowieso eher selten ändern. Ein weiteres Detail, welches man in der Regel gerne beeinflussen würde, ist die Adresse des Service. Wie oben bereits gezeigt, ist diese in der WSDL enthalten und wird per Default vom Client verwendet, um einen Service aufzurufen. In der Praxis findet man an dieser Stelle sehr oft aber nicht die Adresse, unter der ein Service tatsächlich erreichbar ist. Vielmehr ist das die Adresse, die von einem Tool zur Compile- oder Laufzeit automatisch erzeugt wurde. Sie unterscheidet sich von den Adressen, die benutzt werden müssen, nachdem der Service hinter Firewalls und Loadbalancern verschwunden ist.

Ein Client ist zwangsläufig mehr potenziellen Fehlerquellen ausgesetzt als ein Server, da sowohl im Rahmen der Erzeugung des Clients als auch beim Operationsaufruf über ein Netzwerk kommuniziert wird. Netzwerke beziehungsweise Server haben die unangenehme Eigenschaft, manchmal auszufallen, falsch adressiert zu werden oder aus anderen Gründen nicht erreichbar zu sein. Aus diesem Grund erhalten die beiden Callbacks im Code zusätzlich noch ein Error-Objekt, welches Auskunft darüber erteilt, ob der Aufruf überhaupt erfolgreich war. Auf typische Fehlersituationen wie beispielsweise „kein Verbindungsaufbau möglich“ (`ECONNREFUSED`) oder „Server unbekannt“ (`ENOENT` beziehungsweise `ENOTFOUND`) kann somit angemessen reagiert werden. Letzteren Fehler wird man übrigens sehen, wenn im „Port“ der WSDL ein unbekannter Server übertragen wird.

Weitere Fehler können natürlich trotzdem in Form von Exceptions auftreten, unter anderem, wenn unter der übergebenen URL keine WSDL, sondern etwas anderes – also sehr wahrscheinlich HTML – gefunden wird. Das ist oftmals der Fall, wenn eine falsche URL eingetragen wurde oder beim Deployment des Servers etwas schiefgegangen ist:

```
assert.js:102
  throw new assert.AssertionError({
        ^
  AssertionError: Unmatched close tag: ul
  at WSDL._parse (/node_modules/soap/lib/wsdl.js:900:9)
```

SOAP-Services? Aber mit Sicherheit!

Einer der Vorteile von Webservices besteht ja bekanntlich darin, dass zur Verknüpfung zweier Kommunikationspartner auf vorhandene Protokolle und Infrastruktur zurückgegriffen wird – in der Regel HTTP und HTTPS. Selbst wenn Services nicht nur in-House, sondern auch unternehmensübergreifend verwendet werden, ist es für den IT-Betrieb somit vergleichsweise einfach, die benötigte Server-Konfiguration zur Verfügung zu stellen. Aber auch im Intranet sind für den Service-Provider oder Service-Konsumenten oftmals schon diverse Sicherheitsrichtlinien einzuhalten. Spätestens jedoch wenn man mit der Außenwelt kommuniziert, sollte man sich weitergehende Gedanken zu den Themen Authentifizierung, Autorisierung und Verschlüsselung machen:

- Wer ist der Client, der im Moment versucht, eine Verbindung zum Service herzustellen?
- Wenn sichergestellt ist, wer der Client ist, darf dieser Client die gewünschte Operation überhaupt ausführen?
- Wenn der Client berechtigt ist, den Aufruf durchzuführen, wie verhindert man, dass Aufruf oder Antwort verändert oder protokolliert werden?

Glücklicherweise bietet das Modul *soap* für diese Fragestellungen entsprechende Antworten – teils über Funktionalitäten im Modul, teils über Funktionalitäten aus den verwendeten Untermodulen.

Authentifizierung und Autorisierung sind explizit Bestandteil des Moduls. Die Identität des Aufrufers kann über ein WS-Security Username-Passwort Token transportiert werden. Auf Client-Seite wird hierzu einfach ein entsprechendes Objekt instanziiert und am SOAP-Client gesetzt, bevor der Aufruf der Servicemethode erfolgt:

```
soap.createClient(url, function(err, client) {
  if (!err) {
    perflog("after create");
    client.setSecurity(new soap.WSSecurity('username', 'password'));
    client.HelloWorld(args, function(err, result) {
      // in case of an error
    })
  }
}
```

Das ist schon alles, um den SOAP-Request mit entsprechenden WS-Security-Header-Informationen zu versehen. Serverseitig reicht es, die vorgegebene Callback-Funktion `authenticate()` zu definieren, um dann auf diese Informationen zugreifen zu können:

```
soapServer.authenticate = function(security) {
  user = security.UsernameToken.Username;
  password = security.UsernameToken.Password;

  return true;
};
```

`authenticate()` bekommt im Rahmen eines eingehenden Requests eine Datenstruktur übergeben, in der unter anderem Username und Passwort aus dem Webservice Security Header enthalten sind. Die Prüfung der Kombination aus Benutzername und Passwort obliegt natürlich der Anwendung und wird hier nicht weiter betrachtet. Ist die Kombination korrekt, so muss man letztendlich den Wert `true` zurückgeben, ansonsten `false`.

Alternativ kann zumindest auf Client-Seite auch mit Basic Authentication gearbeitet werden. Statt des oben gezeigten `WSSecurity`-Objekts muss dann allerdings ein Objekt vom Typ `BasicAuthSecurity` gesetzt werden. Serverseitig ist die Verwendung von Basic Authentication momentan noch nicht vorgesehen. Die API bietet hier weder entsprechende Callbacks noch den Zugriff auf den HTTP-Request, um den Authentication Header selbst auszulesen und zu prüfen. Es bliebe in diesem Fall allenfalls die Möglichkeit, innerhalb des *soap*-Moduls zu patchen und somit Zugriff auf den Request und Authentifizierungsinformation zu erhalten.

An dieser Stelle noch ein Hinweis, ohne den die Erwähnung von Basic Authentication nicht vollständig wäre: Die im Header übertragenen Daten sehen zwar verschlüsselt aus

```
Authorization: Basic dXNlcm5hbWVCQTpwYXNzd29yZEJB
```

sind aber tatsächlich nur einfach Base64-enkodiert – können also mit einem einfachen Befehl wieder in Klartext verwandelt werden.

```
$ echo dXNlcm5hbWVCQTpwYXNzd29yZEJB | base64 --decode
usernameBA:passwordBA
```

Die Verwendung von Basic Authentication muss also immer Hand in Hand mit einer HTTPS-Verschlüsselung gehen, da der Header sonst von einem Angreifer mitgelesen und für eigene Requests wiederverwendet werden könnte.

Die Verwendung von SSL-gesicherten Verbindungen ist hingegen sowohl auf Client- wie auch auf Server-Seite problemlos möglich. Das *soap*-Modul bedient sich hier der Standardmodule beziehungsweise man muss sich beim Publizieren von Services sowieso selbst um Start und Konfiguration eines passenden Servers kümmern. Das kann statt eines ungesicherten HTTP-Servers auch ein sicherer HTTPS-Server sein.

```
var https = require('https');

var options = {
  key: fs.readFileSync('nodejs.pem'),
  cert: fs.readFileSync('nodejs.pem'),
  passphrase: "nodejs"
};

var server = https.createServer(options,
  function(request, response) {
    console.log("incoming (non-webservice) request: "+request.url);
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.end("some content\n");
  });
server.listen(8443);
soapServer = soap.listen(server,
  '/HelloWorldService',
  serviceCallback, wsdl
);
```

Weitere Details hierzu sind in der Dokumentation zum Modul *https* zu finden[73].

[73] http://nodejs.org/api/https.html

SOAP-Version, Schema-Compliance und andere Einschränkungen

Dass man vergleichsweise schnell zu einem funktionierenden Webservice gelangt, sollte nicht darüber hinwegtäuschen, dass das Modul auch Einschränkungen mit sich bringt. Aktuell wird nur SOAP in der Version 1.1 unterstützt und selbst hier treten bei genauerer Betrachtung noch kleine Schwierigkeiten auf: Die XML-Fragmente, die aus den JSON-Objekten erzeugt werden, besitzen keine Namespaces. Das fällt oftmals gar nicht auf und wird inhaltlich auch in der Regel kein Problem bereiten. Wenn die Konformität mit XML-Schemata von einem der Kommunikationspartner allerdings streng geprüft wird, werden sicherlich Fehler auftreten – für den Bedarfsfall steht auf GitHub[74] ein entsprechender Patch bereit. Wie ein solcher Patch sinnvoll eingesetzt wird – sollte er noch nicht in einer neueren Modulversion vorhanden sein – kann im Abschnitt 1.5 nachgelesen werden.

Des Weiteren ist es auch nicht möglich, Teile des verwendeten XSD-Schemas auszulagern, da `<xsd:import/>` derzeit leider noch nicht unterstützt wird. Auch hier kann ein wenig Handarbeit Abhilfe schaffen. Die WSDL kann dann von einem Client natürlich nicht mehr direkt vom Server gelesen werden, sondern muss in einer gepatchten Version lokal vorgehalten werden.

Auch wenn diese Einschränkungen keine grundsätzlichen Showstopper darstellen, so bleibt doch zu hoffen, dass zukünftig noch Nachbesserungen erfolgen. Man muss sich das konkrete Umfeld und Szenario, in dem man die SOAP-Services verwenden möchte, im Vorfeld durchaus etwas genauer ansehen. Gerade die größtenteils fehlende Unterstützung der WS-* Standards kann im Enterprise-Umfeld sehr schnell dazu führen, dass eine auf Node.js basierende Lösung nicht in die engere Auswahl genommen wird.

Wenn die Unterstützung durch Frameworks nicht ausreicht, um einen SOAP-Service zu benutzen, könnte es manchmal ein letzter Ausweg sein, selbst für die richtigen XML-Strukturen zu sorgen. Theoretisch ist das in beliebiger Tiefe möglich, praktisch empfiehlt es sich aber nicht, einen großen, komplexen Service auf dieser Basis zu realisieren. Die Schwierigkeit liegt ja letztendlich nicht im Generieren spitzer Klammern, sondern im Erzeugen des Inhalts, den diese Klammern umgeben. Wer sich schon mal mit Themen wie Verschlüsselung, Signatur und Kanonikalisierung beschäftigt hat und die zugehörigen Spezifikationen kennt, weiß, dass man so etwas normalerweise nicht gerne selbst macht. Einige Hilfestellungen und Bibliotheken, die hierfür hilfreich sein können, wurden in der Einführung in die XML-Verarbeitung in Node.js – Abschnitt 2.6.3 – beschrieben.

2.6.2 REST-Services

Während SOAP-Services seit Anfang der 2000er-Jahre überall in Enterprise-Architekturen Einzug gehalten haben, konnten sich REST-Services erst in den letzten etwa fünf Jahren immer mehr etablieren. Dabei ist REST keine wesentlich jüngere Technologie im Vergleich zu SOAP – beide entstanden um die Jahrtausendwende. Es ist gut möglich, dass unter anderem das, was man inzwischen als Vorteil von REST sieht, früher zur Zurückhaltung in der Adaption beigetragen hat: wenig Spezifikationen und Standards und aufgrund des geringen technischen Overheads auch de facto keine Toolunterstützung.

[74] https://github.com/milewise/node-soap/pull/12

Was zum Teil ebenfalls der Verbreitung entgegengewirkt haben könnte, sind die Paradigmen, die hinter REST stehen. Die meisten SOAP-Services realisieren letztendlich einen einfachen Funktionsaufruf: ein Service, mehrere Servicemethoden, Eingabeparameter und Rückgaben – Konzepte, die jedermann seit Jahrzehnten bekannt sein dürften.

REST führte hingegen neue, ungewohnte Paradigmen ein: Ressourcen, Verben, Hyperlinks oder beispielsweise Content-Negotiation. Zudem ist REST nicht nur eine Servicetechnologie, sondern ein Architekturstil und die Auswirkungen sind bei einer sinnvollen Einführung im Unternehmen natürlich entsprechend groß. Scheut man diese Aufwände und ignoriert man die REST-Prinzipien, so läuft man sehr schnell Gefahr, eine einfache RPC-Schnittstelle mit Hilfe von REST-Technologien nachzubauen. Beispiele für Services, die den Namenszusatz REST nicht verdienen oder gerade mal „accidentally RESTful" sind, gibt es genügend – auch wirklich namhafte.

An dieser Stelle soll aber gar nicht im Detail auf die Paradigmen hinter REST eingegangen werden, sondern lediglich darauf hingewiesen werden, dass diese gut verstanden sein sollten, wenn man eine entsprechende Architektur plant. Eine sehr gute und praxisnahe Einführung findet man beispielsweise im Buch „Rest und HTTP"[75].

Was im vorliegenden Buch allerdings durchaus näher betrachtet wird, ist die Unterstützung, die im Node-Umfeld zur Verfügung steht, um den REST-Prinzipien gerecht zu werden – sowohl server- wie auch clientseitig. Eines der prominentesten Module ist in diesem Zusammenhang derzeit *Restify*[76].

2.6.2.1 Von Nomen, Verben und Routen

Einfach gesprochen werden in einer REST-Architektur fachliche Entitäten auf sogenannte Ressourcen abgebildet. Nachdem an dieser Stelle von Entitäten und nicht von Aktivitäten oder Prozessen gesprochen wird, wird man im Zusammenhang mit Ressourcen oft auf „Nomen" treffen („Firma", „Mitarbeiter", „Reisekostenabrechnung", „Quittung", ...).

Diese Ressourcen werden dann mit Hilfe eines definierten Satzes von „Verben" manipuliert. In der Praxis verstecken sich die fachlichen Entitäten hinter URIs beziehungsweise URLs und sind über HTTP ansprechbar. Die vom HTTP-Protokoll definierten Verben wie `GET`, `POST`, `PUT` und `DELETE` werden dann serverseitig auf die Ressourcen geeignet angewandt.

Ganz nebenbei ergibt sich hierdurch auch eine erste Indikation, ob eine Schnittstelle in Richtung REST tendiert: Enthält eine URI auch Verben, so sollte man das Design eventuell nochmals überdenken.

Wie sieht nun die URI einer solchen Ressource konkret aus? In der Regel wird man hier verschiedene fachliche Komponenten wiederentdecken, die die eigentliche Ressource eindeutig machen. Stellt man sich beispielsweise die Mitarbeiterverwaltung einer Firma vor, so könnte es folgende Ressourcen geben:

```
http://acme.com/employee/0815/expensereport/201212
http://acme.com/employee/0815/expensereport/201212/receipt/17
```

[75] http://rest-http.info/
[76] https://www.npmjs.com/package/restify

Würde man auf eine der URLs das Verb `GET` anwenden, dann würde der Server vermutlich mit der Reisekostenabrechnung Dezember 2012 oder der zugehörigen Quittung mit Nummer 17 antworten. Ein `POST` auf die zweite URL ohne die abschließende laufende Nummer würde eine neue Quittung Nummer 18 anlegen, ein `PUT` auf `http://acme.com/employee/0815/expensereport/201301` wird die Abrechnung für Januar 2013 anlegen. In beiden Fällen würden vermutlich noch zusätzliche Daten im Body des Requests übertragen. Ein `DELETE` auf eine der URLs löscht dann die jeweilige Ressource wieder.

Was sollte also ein Modul bieten, das hier serverseitig unterstützen möchte? Wie man den oben gezeigten URLs schnell ansieht, handelt es sich nicht um sogenannte Endpoints, die wie bei SOAP-Services auf eine Funktion abgebildet werden und weitere Informationen per URL-Parameter oder HTTP-Body erhalten. Ein Teil der benötigten Parameter sind vielmehr bereits in der URL verschlüsselt.

Das Modul sollte somit die Möglichkeit bieten, parametrisierte „Routen“ zu definieren und diese mit verschiedenen HTTP-Verben verknüpfen zu lassen.

```
var restify = require('restify');
var server = restify.createServer();

server.use(function(req, res, next) {
  console.log('incoming request');
  return next();
});

server.get('/employee/:emp_num/expensereport/:rep_num/receipt/:rec_num', function
(req, res, next) {
  console.log(req.params.emp_num);
  res.send('some content');
  return next();
})
```

Der Restify-Server wird erzeugt und anschließend wird eine erste Route definiert, wobei Teile der URL durch Parameter-Tokens ersetzt werden können. Auf die einzelnen URL-Fragmente kann man anschließend im Callback über das Request-Objekt genauso zugreifen wie auf andere Attribute des Requests wie beispielsweise HTTP-Header.

Mit Hilfe der gezeigten Route kann bereits auf „implizit“ übertragene Daten zugegriffen werden. Implizit deshalb, weil im eigentlichen Sinne gar keine Datenübertragung stattfindet. Da REST-URLs aber im Vergleich zu SOAP-URLs eine inhaltliche Semantik mit sich bringen, darf durchaus von Datenübertragung gesprochen werden, wenn Fragmente der URL zur nachgelagerten Verarbeitung verwendet werden. Neben dieser impliziten Übertragung gibt es natürlich auch noch die Möglichkeit, explizit Daten zu übertragen. Wie oben schon knapp angedeutet, werden im Falle von `POST`- und `PUT`-Aufrufen meistens im Body Daten mitgegeben, die dann als neue Ressource angelegt oder zum Aktualisieren einer bestehenden Ressource herangezogen werden.

```
server.post('/employee', function (req, res, next) {
  console.log("data: "+req.body);
  res.send(201, "created");
  return next();
});
```

Mit der gezeigten Route kann eine neue Ressource unter `/employee` angelegt werden. Wie diese Ressource heißt, kann sich eventuell aus den übertragenen Daten ergeben oder auch vom Server bestimmt werden. Im letzteren Fall würde der Server natürlich nicht nur mit „created" antworten, sondern sinnvollerweise auch die zugehörige URL mitliefern. Testen lässt sich die Ressource beispielsweise mit dem kleinen Kommandozeilen-Tool `curl`[77]. Es steht auf Linux, Unix und MacOS standardmäßig zur Verfügung und ist auch für Windows kostenfrei erhältlich:

```
curl -H "Content-type: application/json" -d '{"name":"John"}' -X POST http://
localhost:8080/employee
```

Allerdings dürfte sich leider herausstellen, dass für den Body keine Daten angezeigt werden. Das liegt an der Asynchronität, mit der man es im Node.js-Umfeld immer zu tun hat. Wie bei der Benutzung des eingebauten HTTP-Moduls muss man darauf achten, dass der eingehende Request und die übertragenen Daten nicht als geschlossene Einheit empfangen werden. Der eingehende Request signalisiert sozusagen nur den Beginn der Übertragung, die einzelnen Blöcke von übertragenen Daten werden auch hier über das Event „data" signalisiert und müssen entsprechend verarbeitet werden. Man könnte nun einen internen Puffer anlegen und alle empfangenen Datenpakete hintereinander speichern, bis das Ende der Übertragung mittels Event „end" angezeigt wird. Allerdings liefert *Restify* für diese Funktionalität bereits ein Plug-in („bodyParser"), welches lediglich aktiviert werden muss. Wie das gemacht wird, ist im Abschnitt 2.6.2.4 beschrieben. Sobald das Plug-in aktiv ist, wird der Callback für die oben definierte Route erst aufgerufen, wenn auch die Daten des Requests komplett eingetroffen sind, und auf `req.body` kann wie oben angedeutet zugegriffen werden.

Die dritte Möglichkeit, Daten zu übergeben, liegt in der Verwendung von URL-Parametern. Auch wenn die fachlichen Ressourcen an sich immer über ihre URL identifiziert werden, so kann es trotzdem sinnvoll sein, über Query-Parameter weitergehend auf das gelieferte Ergebnis Einfluss zu nehmen. Der nachfolgende Request wird mit der kompletten Liste der Angestellten beantwortet – sortiert wie im Parameter angegeben:

```
http://acme.com/employees?sort=descending
```

Auch hier bietet ein Plug-in die benötigte Hilfestellung. An dieser Stelle ist es das Plug-in „queryParser", welches dafür sorgt, dass auf die Parameter zugegriffen werden kann. Ist es aktiv, so werden die URL-Parameter mit in die params-Liste des Requests aufgenommen.

Ein eingehender Request kann also nicht nur von einem, sondern von einer beliebigen Anzahl von Callbacks bearbeitet werden. Man unterscheidet zwischen globalen Handlern – Plug-ins – und für die Route spezifischen Handlern. Plug-ins werden mittels der Methode `use()` direkt am Server registriert. Man könnte sich Plug-ins in etwa wie Interceptoren oder Aspekte vorstellen, die intern mit den Routen verbunden werden. Damit erscheint es dann auch einleuchtend, dass diese Handler nur dann ausgeführt werden, wenn tatsächlich eine Route für den eingehenden Request existiert.

Neben dem Request-Objekt erhält ein mit einer Route oder einem Handler definierter Callback zudem noch ein Response-Objekt und den nächsten Handler in der Kette. Man muss

[77] http://curl.haxx.se/

jedoch dafür Sorge tragen, dass nur ein Handler die Rückgabe an den Client schreibt, ein weiterer Versuch würde mit einem Laufzeitfehler quittiert. Bei Bedarf lässt sich das erzwingen, indem man dem Folgehandler den Wert `false` oder eine Fehlerinstanz übergibt. Mehr hierzu auch im Abschnitt 2.6.2.3 zur Fehlerbehandlung.

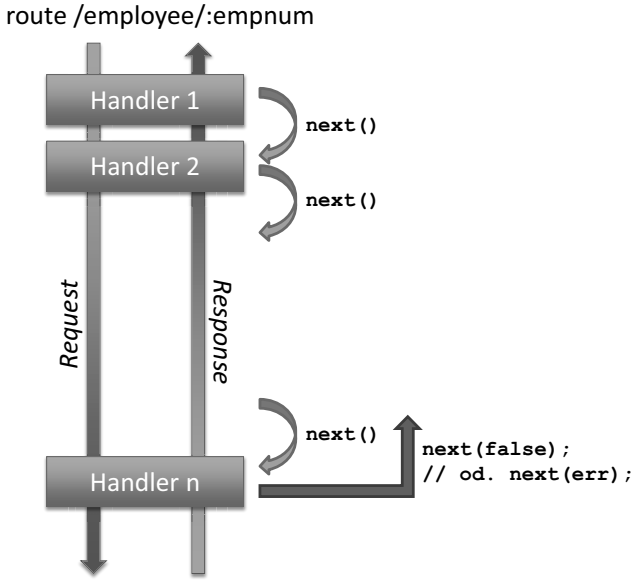


Bild 2.16 Routing und Handler in Restify

Die Route – also die URL, auf die reagiert werden soll – kann als einfacher String, als Regular Expression oder als komplexes Objekt übergeben werden, falls man die Route genauer spezifizieren beziehungsweise konfigurieren möchte. Unter anderem ist es hier möglich, eine Version dieser Route anzugeben.

2.6.2.1.1 Versionierung

Wer Services publiziert, wird sich hin und wieder auch Gedanken zum Thema Versionierung machen müssen. Es ist zwar empfehlenswert, nach Möglichkeit immer nur eine Version eines Service aktiv zu haben, aber manchmal lassen sich parallel existierende Dienste nicht vermeiden. Es gibt aus technischer Sicht verschiedene Möglichkeiten, dieses Problem zu adressieren. Man könnte beispielsweise eine Versionsnummer in der URL verschlüsseln oder den Content-Type entsprechend versionieren. Restify bietet die optionale Möglichkeit, Routen mit Versionsnummern zu versehen. Ein eingehender Request wird immer daraufhin untersucht, ob er in den Header-Daten eine bestimmte Version einer Route einfordert oder nicht und ob diese von einer der definierten Routen bedient werden kann. Dabei gelten folgende Regeln:

- Besitzt eine Route keine Versionsnummer, so kann sie jeden eingehenden Request bedienen.
- Besitzt ein eingehender Request keine Versionsnummer, so kann er von jeder passenden Route bedient werden.

- Können mehrere Routen einen Request bedienen, so tut dies die Route, die als Erstes definiert wurde.

Die Versionsnummer wird gemäß Semantic Versioning Specification (SemVer)[78] übergeben – ein Format, das vermutlich jedem geläufig ist, beispielsweise aus Apache Maven oder *npm*. Nachfolgend sehen Sie eine Pseudoressource, die in Version 1.0.0 und 2.0.0 vorliegt und mit verschiedenen Versionsvorgaben angefragt wird.

```
var restify = require('restify');
var server = restify.createServer();

server.get({url: '/versioned/service', version: '1.0.0'},
  function(req, res, next) {
    res.send("v1");
    return next();
  });
```

```
var restify = require('restify');
var server = restify.createServer();

server.get({url: '/versioned/service', version: '2.0.0'},
  function(req, res, next) {
    res.send("v2");
    return next();
  });
```

```
$ curl -s -H 'accept-version: 2.0.*' -w "\nHTTP: %{http_code}\n" localhost:8080/
versioned/service
  "v2"
  HTTP: 200

$ curl -s -H 'accept-version: <2.*' -w "\nHTTP: %{http_code}\n" localhost:8080/
versioned/service
  "v1"
  HTTP: 200

$ curl -s -H 'accept-version: >1.1.0 <3.0.0' -w "\nHTTP: %{http_code}\n"
localhost:8080/versioned/service
  "v2"
  HTTP: 200
```

2.6.2.2 Ansichtssache? Verhandlungssache

Teil des REST-Paradigmas ist die Content-Negotiation. Wo eine normale Webanwendung hart kodiert – meistens anhand einer URL – HTML-Seiten (`text/html`) oder beispielsweise eine PDF-Druckansicht (`application/pdf`) zurückgibt, ist in einer REST-Anwendung die Aufbereitung der Daten „Verhandlungssache“. Der aufrufende Client schickt im Accept-Header alle von ihm verstandenen Datenformate. Falls mehr als ein Format übergeben wird, bemüht sich der Server, ein möglichst weit vorne in der Liste befindliches Format zu erzeugen. Der Client sollte also sein Wunschformat an erster Stelle platzieren.

[78] http://semver.org

Im Gegensatz zu einigen anderen vergleichbaren Frameworks erfolgen bei *Restify* die Erzeugung des Contents und die Überführung in das benötige Format erfreulich stark getrennt. Die in den Routen konfigurierten Handler kümmern sich lediglich um die abzuarbeitende Logik. Die Aufbereitung der Daten wird in sogenannten Formattern vorgenommen, die in keiner Beziehung zu den Routen stehen, sondern direkt im Server-Objekt bekanntgemacht werden. Drei Formate werden bereits standardmäßig unterstützt (`application/json`, `application/octet-stream`, `text/plain`), weitere können einfach über zusätzliche Formatter registriert werden.

Ein Formatter ist eine Callback-Funktion, die mit einem Medientyp oder Mime-Type registriert wird. Wird die Antwort in der Server-Logik mit `send()` in Richtung Client abgeschickt, so wird unter anderem der erwartete Content-Type mit der Liste von Formattern abgeglichen. Wird ein passender Formatter gefunden, so hat dieser Zugriff auf Request, Response und Body und kann entsprechende Formatierungen bzw. Transformationen vornehmen. Im Prinzip steht es dem Formatter völlig frei, den Inhalt des Body in ein beliebiges Format zu überführen. Auch wenn man völlige Freiheit genießt, so sollte man sich der Tatsache bewusst sein, dass man sich in einem Formatter befindet. Es versteht sich von selbst, dass an dieser Stelle ausschließlich an der Darstellung des Inhalts, nicht aber am Inhalt selbst gearbeitet wird.

Nachfolgend wird ein (sehr, sehr einfacher) CSV-Formatter für den MIME-type `text/csv` registriert. Zuerst wird von der Möglichkeit Gebrauch gemacht, auf die Response zugreifen zu können, und ein HTTP-Header gesetzt. Falls der Body ein Objekt enthält, werden die Attributnamen und Attributwerte in ein einfaches CSV-Format überführt.

```
var csvFormatter = function(req, res, body) {
  res.setHeader('header-strings', 'excluded');

  if (body instanceof Object) {
    retval = "";
    for (attName in body) {
      retval += (attName+","+body[attName]+"\r\n");
    }
    return retval;
  }
}

options = {
  formatters: {
    'text/csv': csvFormatter
  }
};

var server = restify.createServer(options);
```

Zwei Hinweise noch an dieser Stelle: Für den Header gilt natürlich die gleiche Anmerkung wie schon für den Content. Hat der Inhalt des Headers etwas mit dem Datenformat zu tun, so kann er hier gesetzt werden, ansonsten nicht. Der zweite Hinweis bezieht sich auf den Body der Nachricht. In ihm muss sich nicht unbedingt der Inhalt befinden, den man erwartet. Wird die Verarbeitungslogik mit einem Fehler abgebrochen, so ist dort beispielsweise ein Error-Objekt enthalten.

2.6.2.3 Fehlermeldungen

Verwendet eine REST-basierte Anwendung HTTP als Implementierung, so sollten natürlich auch eventuelle Statusmeldungen mittels der entsprechenden HTTP-Codes übertragen werden. Wann welcher Code verwendet werden muss, soll nicht Gegenstand dieses Buchs sein. Eine Auflistung der Codes und ihrer Bedeutung gibt es an vielen Stellen im Internet wie beispielsweise im RfC für das Hypertext Transfer Protocol, federführend eingebracht von Roy Fielding[79]. Natürlich ist es mit einer einfachen Auflistung der Codes nicht getan, die Codes müssen auch richtig eingesetzt werden. Auch hierfür sei nochmals auf die eingangs erwähnte Literatur verwiesen.

Wie werden nun mittels *Restify* Fehler- beziehungsweise Statuscodes übermittelt? Im einfachsten Fall wird gar kein Code explizit übermittelt, sondern die Verarbeitung einfach durchgeführt und eine Antwort gesendet. Diese Antwort wird automatisch mit dem Code 200 (OK) versehen. Ein anderes mögliches Szenario ist, dass während der Request-Verarbeitung ein unerwarteter Fehler – quasi eine unchecked Exception – auftritt und die Verarbeitung abgebrochen wird, ohne dass in der Implementierung hierauf gesondert reagiert wird. *Restify* übernimmt das Message-Attribut des Fehlers dann in die Rückantwort und setzt den Code 500 (Internal Server Error).

Interessanter sind die Fälle, in denen Codes speziell gesetzt werden müssen, um den Status einer Ressource (201, 404, ...) zurückzuliefern. Diese Codes können entweder der `send()`-Methode an der Response direkt mitgegeben werden oder sie werden explizit über das Attribut `statusCode` gesetzt.

```
server.get('/error/simple/0', function (req, res, next) {
  res.send("hello, world!");
  return next(false);
});

server.get('/error/simple/11', function (req, res, next) {
  res.statusCode = 201;
  res.send("created something");
  return next(false);
});
```

Grundsätzlich gelten dieselben Einschränkungen und Möglichkeiten, wie sie in anderen Modulen vorzufinden sind, die mit dem Objekt `http.ServerResponse` arbeiten.

Darüber hinaus definiert *Restify* zudem noch die Klassen `HttpError` und `RestError`, um Fehler komfortabler erzeugen und abfangen zu können. Für alle Http-Fehlercodes (4xx, 5xx) sind Konstanten definiert, die dem Fehlernamen aus der Spezifikation entsprechen. Anbei ein Beispiel für den Code 402 – Leerzeichen entfernen und „Error“ anhängen:

```
// create 402 - Payment Required error
var err = new restify.PaymentRequiredError(
  "$99",
  {amount:99, command:'pay me!'}
);
```

[79] http://www.w3.org/Protocols/rfc2616/rfc2616.txt

Die angebotenen `RestError`-Klassen leiten sich von `HttpError` ab und repräsentieren gängige Fehler aus dem REST-Umfeld. Nachfolgend die Liste der originalen Statuscodes und der abgeleiteten REST-Fehler:

- 400 (Bad Request): BadDigestError, InvalidContentError, InvalidHeaderError, InvalidVersionError, RequestExpiredError
- 401 (Unauthorized): InvalidCredentialsError
- 403 (Forbidden): NotAuthorizedError
- 404 (Not Found): ResourceNotFoundError
- 405 (Method Not Allowed): BadMethodError
- 406 (Not Acceptable): WrongAcceptError
- 409 (Conflict): InvalidArgumentError, MissingParameterError
- 412 (Precondition Failed): PreconditionFailedError
- 429 (Too Many Requests, RfC 6585): RequestThrottledError
- 500 (Internal Server Error): InternalError

```
server.get('/error/simple/12', function (req, res, next) {
  return next(new restify.PaymentRequiredError(
    "$99",
    {amount:99, command:'pay me!'}
  ));
});

server.get('/error/simple/13', function (req, res, next) {
  return next(new restify.MissingParameterError(
    "oops, something's missing"
  ));
});
```

Wie im unteren Code-Fragment zu sehen ist, kann jedoch mit einem `RestError` lediglich eine Nachricht übertragen werden. Nur der `HttpError` kann auch noch ein Objekt im Body transportieren.

Da nicht immer bekannt ist, welche Fehler aus den Tiefen der Businesslogik zu erwarten sind, kann zudem ein Event verwendet werden, um immer geregelt auf einen Fehler zu reagieren. Statt eines allgemeinen „Internal Server Error“ kann mit Hilfe des Ereignisses „uncaughtException“, welches vom Restify-Server erzeugt wird, eine beliebige passende Fehlermeldung zum Client übermittelt werden:

```
server.on("uncaughtException",
  function (request, response, route, error) {
    // default behaviour: response.send(error);
    response.send(new restify.ServiceUnavailableError("unexpected error"));
  });
```

2.6.2.4 Plug-ins

Neben den allgemeinen Serverfunktionalitäten liefert *Restify* zudem Plug-ins mit aus, wie sie auch aus *Express* bekannt sind. Plug-ins sind im Wesentlichen Funktionen, die als globaler Handler oder theoretisch auch für eine spezifische Route installiert werden können.

Letzteres ist natürlich nur bedingt sinnvoll, da in spezifischen Routen auch spezifische Logik abgearbeitet werden sollte. Die mitgelieferten Plug-ins decken einige Standardanforderungen ab, die man ansonsten sicherlich früher oder später selbst entwickeln müsste. Neben Funktionalitäten wie dem Parsen der übertragenen Daten oder dem Auslesen von Zeitstempeln oder Authentifizierungsinformationen existieren auch interessante Plug-ins, die eine Serverüberlastung verhindern beziehungsweise das Verarbeitungsvolumen optimieren sollen. Auf eine Auswahl dieser Plug-ins gehen die folgenden Abschnitt etwas näher ein, um die Anwendung dieses Konzepts grundsätzlich zu zeigen. Abschließend wird noch demonstriert, wie man ein eigenes Plug-in schreibt und es in die Handler-Kette aufnimmt.

bodyParser

Wie bereits erwähnt, werden der eingehende Request und die mitgeschickten Daten nicht als Einheit betrachtet. Um nicht selbst auf „data"- und „end"-Events des Requests reagieren zu müssen, kann das Plug-in „bodyParser" aktiviert werden. Für bestimmte Content-Types übernimmt es die Behandlung der Ereignisse und stellt die Daten letztendlich im Attribut `body` des Requests zur Verfügung. Das Plug-in kann mit den Typen `application/json`, `application/x-www-form-urlencoded` und `multipart/form-data` umgehen. Andere Formate werden im Normalfall ignoriert, was dazu führt, dass `body` beim Aufruf des Callbacks wieder undefiniert sein wird. Erachtet man es aus Server-Sicht sogar als unzulässig oder falsch, andere Datenformate zu senden, so kann man das Plug-in auch so starten, dass es mit einer Fehlermeldung reagiert, wenn auf den Content-Type nicht reagiert werden kann:

```
server.use(restify.bodyParser({rejectUnknown:true});
```

Wird ein Request mit nicht weiter typisierten Daten oder einem unbekannten Format geschickt, so antwortet der Server in diesem Fall mit einem HTTP-Code 415 (Unsupported Media Type).

Das Plug-in kann leider nicht einfach für andere Datentypen erweitert werden. Ist es erforderlich, anwendungsspezifische Typen zu verarbeiten, so muss hierfür ein eigenes Plug-in geschrieben werden. Das mitgelieferte Plug-in kann dann entweder entfernt werden oder es muss so betrieben werden, dass Fremdformate nicht hart abgewiesen werden. Hinweise zum Erstellen eigener Plug-ins folgen im Abschnitt 2.6.2.4.1 „Custom Plug-ins".

Analog zum Plug-in „bodyParser" gibt es noch einen Parser, der die Daten aus Query-Parametern übernehmen kann. Wird dieses Plug-in aktiviert, so findet man die Parameter anschließend ebenfalls im Attribut `params` des Requests. Achtung! Es kann theoretisch sein, dass sowohl ein URL-Parameter als auch ein Query-Parameter den gleichen Namen haben und sich somit gegenseitig überschreiben würden. In so einem Fall brechen die Plug-ins die Verarbeitung ab und der aktuelle und auch eventuell weitere Parameter gehen verloren. Dieselbe Situation ergibt sich auch beim Parsen von übergebenen Formularen. Dieses Verhalten kann auch umgangen werden, indem den Parsern ein Objekt {`overrideParams:true`} übergeben wird. Allerdings stellt sich durchaus die Frage, ob das Überschreiben von URL-Parametern wirklich sinnvoll ist.

throttle

Sobald man Intranets oder Virtual Private Networks verlässt und mit einer Anwendung öffentlich erreichbar ist, muss man sich Gedanken zur zu erwartenden Server-Last machen und die Systemlandschaft entsprechend planen. Normalerweise hat man zwar auch eine Idee von den zu erwartenden Lastprofilen, aber leider keine Gewissheit. Zumal sich oft zeigt, dass ganz banale Effekte in der Planung unbeachtet bleiben: Angenommen, man betreibt ein beliebiges Online-Angebot, das – aus welchen Gründen auch immer – aus Performancesicht langsam an seine Grenzen gelangt. Ein Peak in der Benutzung führt nun zu etwas mehr Verkehr und sorgt dafür, dass das System zwar noch stabil ist, aber langsamer reagiert. Im Idealfall hat man nur geduldige Kunden und das Problem löst sich von selbst, wenn der Peak wieder abklingt. Im weniger idealen Fall denken Kunden, sie könnten ihre Wartezeit durch erneute Klicks verkürzen. Tatsächlich steigt damit die Last auf dem Server nochmals deutlich an und bringt eventuell das System in einen instabilen Zustand, aus dem es sich von selbst nicht mehr erholen kann. Der Grund hierfür hat sich vorab nicht aus technischen Parametern ableiten lassen. Man tut also gut daran, grundsätzliche Absicherungsmechanismen vorzusehen. Dies gilt umso mehr, wenn man eine öffentliche Schnittstelle zur Verfügung stellt, da man unter Umständen nur wenig Einfluss darauf hat, wer die Schnittstelle wann und in welcher Häufigkeit aufrufen wird.

Das Request-Aufkommen in Web-Frontends ist in der Regel ganz natürlich über die Kontraktionsgeschwindigkeit der Muskeln im durchschnittlichen Zeigefinger begrenzt. Befeuern hingegen Prozesse eine REST-Schnittstelle, so sollte man besser serverseitig selbst für eine entsprechende Begrenzung sorgen. In Fällen, in denen die vorhandene Infrastruktur nicht für die gewünschte Absicherung sorgt, kann das Plug-in „throttle“ verwendet werden. Mit diesem ist es möglich, die Frequenz eingehender Requests anhand verschiedener Kriterien zu begrenzen. Zudem können aber auch Ausnahmen definiert werden, so dass privilegierte Konsumenten hiervon nicht beeinträchtigt werden.

```
var server = restify.createServer();

server.use(restify.throttle({
  burst: 50,
  rate: 10,
  ip: true
}));
```

Die Parameter `burst` und `rate` dienen zur Konfiguration der Drosselung. Hinter dem Plug-in steht der Algorithmus „Token Bucket“ – ein Eimer mit Tokens – was ein wenig zur Erklärung der Parameter beiträgt: `burst` steht für die maximale Anzahl der Tokens, mit denen der Eimer gefüllt werden kann. Jeder eingehende Request nimmt ein Token aus dem Eimer, solange noch Tokens vorhanden sind. Die fehlenden Tokens werden anhand des Parameters `rate` ersetzt. Ist zum Zeitpunkt des Eintreffens eines Requests kein Token im Eimer, so wird der Request mit einem HTTP-Code 429 (Too Many Requests) abgelehnt, ohne die Businesslogik und damit eventuell weitere, nachgelagerte Systeme zu beeinträchtigen.

```
HTTP/1.1 429 Too Many Requests
{"message":"You have exceeded your request rate of 10 req/s."}
```

Soweit die Theorie – aber wie lässt sich das nun möglichst einfach testen? Der Apache Webserver beinhaltet das Tool ApacheBench (ab). Auf Unix-Systemen steht es oftmals schon zur Verfügung und auch unter Windows kann es mit dem kompletten Apache Server installiert werden. Es bietet die einfache Möglichkeit, einen Server mit einer bestimmten Anzahl von (parallelen) Requests unter Stress zu setzen.

```
ab -n 10000 -c 100 http://10.211.55.2:8080/ping
```

ApacheBench stellt nun 10000 Anfragen in 100 parallelen Threads an den REST-Server. Ping ist in diesem Fall mit einer künstlichen Bearbeitungszeit von zwei Sekunden ausgestattet. Mit obiger Konfiguration sollte man also erwarten, dass 50 Requests sofort akzeptiert werden. Ab dann werden die Requests abgewiesen, bis nach etwa zwei Sekunden die akzeptierten Requests fertig bearbeitet sind. Mit einer Rate von zehn Tokens pro Sekunde wird der Eimer wieder aufgefüllt und weitere Requests werden akzeptiert.

```
Server Software:         restify
Server Hostname:         10.211.55.2
Server Port:             8080

Document Path:           /ping
Document Length:         60 bytes

Concurrency Level:       100
Time taken for tests:    9.622 seconds
Complete requests:       10000
Failed requests:         77
   (Connect: 0, Receive:  0, Length: 77, Exceptions: 0)
Write errors:            0
Non-2xx responses:       9923
Total transferred:       5915615 bytes
HTML transferred:        596535 bytes
Requests per second:     1039.33 [#/sec] (mean)
Time per request:        96.216 [ms] (mean)
Time per request:        0.962 [ms] (mean, across all concurrent requests)
Transfer rate:           600.42 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd]  median   max
Connect:        0    1   2.7       1      44
Processing:     1   73 221.0      53    2639
Waiting:        1   73 220.9      52    2639
Total:          2   74 221.2      54    2640
```

Der Server hat die 10000 Requests in knapp zehn Sekunden „behandelt“. Tatsächlich wurden bis auf 77 Anfragen alle Requests abgewiesen. Eine kleine Anmerkung an dieser Stelle: Dass in obiger Statistik die 77 Requests als „failed“ eingestuft werden, liegt daran, dass ApacheBench eine andere Länge der Antwort erwartet hat. Wird mehrmals dieselbe URL angefragt, so wird die Länge der ersten Antwort als Referenz verwendet. Die erste Antwort ist im obigen Beispiel aber schon der erste HTTP-429-Fehler inklusive entsprechendem Meldungstext. Alle korrekten Antworten werden fortan leider als fehlerhaft eingestuft.

Diese kleine Ungenauigkeit in der Auswertung soll nicht darüber hinwegtäuschen, dass das gewünschte Ziel erreicht wurde: Selbst bei extremen Spitzenbelastungen können die nachgelagerten Systeme ausreichend geschützt werden.

conditionalRequest

Eine weitere Möglichkeit, Last aus einem Server zu nehmen, ist es, nichts zu tun, wenn nichts zu tun ist. Zugegeben klingt das sehr banal, aber trotzdem skalieren viele Anwendungen eher über Technik als über Design. Die Idee ist, dass ein Client eine Ressource nur dann aktualisiert, wenn sie sich auch wirklich geändert hat. Die Zauberworte sind in diesem Fall „ETag“, „Zeitstempel“ und „bedingte Verarbeitung“.

Im Zusammenhang mit REST-Services und HTTP bieten sich zwei Möglichkeiten an, zu entscheiden, ob eine vollständige Verarbeitung des Requests und Auslieferung des Ergebnisses notwendig ist.

Die erste ist der Einsatz von sogenannten ETags (Entity Tags), die einen Vergleich zwischen der beim Client bereits vorhandenen und der zu sendenden Ressource bieten. Die Funktionsweise ist ganz einfach: Wenn eine Ressource angefragt wird, liefert der Server sie aus und sendet zudem ein ETag im Header mit. Der Client speichert die Ressource und das ETag. Wenn der Client die Ressource aktualisieren möchte, sendet er im Header der Anfrage das ETag wieder an den Server. Stellt der Server fest, dass das ETag (und damit die Ressource) noch aktuell sind, antwortet er lediglich mit dem HTTP-Code 304 (Not Modified), ansonsten sendet er eine Aktualisierung. Ein typischer Einsatzbereich wären Daten, die sich nur selten und unregelmäßig ändern, aber so groß sind, dass man sie nicht unnötig übertragen möchte oder deren komplette Errechnung sehr aufwendig ist.

Die zweite Möglichkeit geht davon aus, dass Daten eine gewisse Zeitlang gültig bleiben. Werden serverseitig beispielsweise Reports durch nächtliche Batches erstellt, so macht es keinen Sinn, diese Reports mehrmals täglich anzufordern. Oftmals wird auch eine Kombination beider Verfahren verwendet, so dass der Client an sich nur eine Anfrage stellt, wenn die Ressource ihr Verfallsdatum überschritten hat, der Server aber anhand des ETags trotzdem nochmals prüft, ob das Senden notwendig ist.

Mit diesem Vorgehen wird nicht nur CPU-Zeit im Server gespart, sondern auch Bandbreite auf dem Weg zum Client. Gerade im Hinblick auf mobile Geräte lohnt es sich also durchaus zu prüfen, ob für manche Services ETags eingeplant werden sollten.

ETags können übrigens auch für schreibende Zugriffe beispielsweise über `POST` und `DELETE` verwendet werden. In diesem Fall lässt sich damit für die REST-Services eine Semantik realisieren, die mit der von Optimistic Locking bei Datenbankoperationen vergleichbar ist.

Die Aktivierung des Plug-ins verläuft genauso wie auch bei anderen Plug-ins – über ein `server.use()` wird es global installiert. Natürlich setzt man es vor die Handler für die spezifischen Routen, da die ja gegebenenfalls gar nicht ausgeführt werden sollen. Was allerdings tatsächlich beachtenswert ist, ist die Tatsache, dass das Plug-in davon ausgeht, dass eine vorab aktive Komponente bereits Datenfelder im Response-Objekt gesetzt hat. Für die Überprüfung eines ETags wird beispielsweise `response.header('ETag')` mit den Inhalten von `request.headers['if-match']` verglichen.

2.6.2.4.1 Custom Plug-ins

Die Entwicklung von eigenen Plug-ins ist denkbar einfach. Die Funktion, die sich hinter einem globalen Handler versteckt, entspricht im Wesentlichen denen, die als Routenhandler bereits gezeigt wurden:

```
server.get('/echo/:name', function(req, res, next) {
  console.log("name: "+name);
  return next();
})
```

Die Route wird mit einer Funktion assoziiert, die als Parameter den Request, die Response und den nächsten Handler erhält. Analog wird auch eine Funktion für einen globalen Handler – hier eine Log-Funktion – definiert:

```
return function log(req, res, next) {
  console.log("["+options.prefix+"] incoming request");
  return next();
}
```

Die meisten Plug-ins werden aber nicht direkt als Funktion installiert, sondern indirekt mit Hilfe einer erzeugenden Funktion, deren Rückgabe die eigentliche Handlerfunktion ist. Was ist der Grund hierfür? Werden der Konstruktorfunktion Parameter übergeben, so stehen diese Parameter später innerhalb des Handlers zur Verfügung – Stichwort „Closure“. Im oben gezeigten Beispiel wird jede Zeile des Logs mit einem Präfix versehen. Dieses Präfix und gegebenenfalls noch andere Optionen wurden dem Konstruktor übergeben und stehen somit auch Funktionen zur Verfügung, die innerhalb der Konstruktorfunktion definiert werden. Und das, selbst wenn die Ausführung der Konstruktorfunktion beendet ist:

```
function logger(options) {
  if (!options) {
    options = {};
  }
  if (!options.prefix) {
    options.prefix = '[]';
  }

  return function log(req, res, next) {
    console.log("["+options.prefix+"] incoming request");
    return next();
  }
}
```

Die Funktion `logger()` ist die Konstruktorfunktion, `log()` der globale Handler. Installiert wird das Plug-in mittels `server.use(logger(…));`, wobei `logger()` die Funktion `log()` zurückgibt, die fortan Zugriff auf das `options`-Objekt hat.

2.6.2.5 Sicherheit und Authentifizierung

Wie auch schon bei den SOAP-Services, soll auch für REST-Services das Thema Security nicht zu kurz kommen. Im Bereich SOAP gibt es neben den über das Protokoll definierten Sicherheitsstandards – in den meisten Fällen HTTP mit SSL und Basic Auth oder Client Certificates – noch eine Vielzahl von WS-*Spezifikationen, die für Sicherheit sorgen, wenn sie denn vom verwendeten SOAP-Stack unterstützt werden. REST beschreibt hingegen kein konkretes Protokoll, sondern einen Architekturstil. Es geht also weniger um das Erfüllen strikt definierter Vorschriften, als darum, in der Denkweise von Architekten Platz für neue Paradigmen zu schaffen.

Auch die meisten REST-Architekturen bedienen sich des HTTP-Protokolls, wodurch die oben erwähnten Technologien SSL, Basic Auth und Client-Zertifikate zur Verfügung stehen. Richtig eingesetzt bieten diese Konzepte auf jeden Fall schon ein gewisses Maß an Sicherheit – letztendlich basiert ja unser gesamter Internetverkehr darauf. Es gibt zwar sehr kontroverse Diskussionen darüber, wie sicher sie tatsächlich sind, aber für die meisten Anwendungen dürften sie ausreichen. Man kommt aber nicht umhin, sich Gedanken darüber zu machen, in welchem Umfeld die Services angeboten beziehungsweise konsumiert werden, und daraus abzuleiten, wie viel Sicherheit man tatsächlich benötigt.

Außerdem ist nicht jedes Sicherheitskonzept für jedes vorliegende Szenario geeignet. Client-Zertifikate sorgen zwar dafür, dass der Server weiß, wer ihn kontaktiert. Allerdings trägt diese Lösung natürlich nur, wenn es sich nicht um einen öffentlichen Service handelt, die Anzahl der Konsumenten also begrenzt ist. Letztendlich müssten in einer solchen Situation die Zertifikate explizit ausgetauscht beziehungsweise verteilt werden. In den folgenden Abschnitten werden noch weitere Methoden gezeigt, mit denen sich ein Client gegenüber dem Server authentifizieren kann.

Im Zusammenhang mit SOAP-Services wurde bereits gezeigt, wie ein SSL-gesicherter Server verwendet werden kann. Da auch *Restify* letztendlich auf den Modulen *http* und *https* aufbaut, funktioniert die Verwendung nahezu analog. Die Entscheidung, welche Art von Server gestartet wird, trifft *Restify* anhand der übergebenen Parameter. Befinden sich ein Zertifikat und ein Schlüssel unter ihnen, wird intern ein HTTPS-Service gestartet.

```
options = {
  certificate: fs.readFileSync('server.crt'),
  key: fs.readFileSync('server.key')
};

var server = restify.createServer(options);
// will launch https service
```

Die Kommunikation mit dem Server ist nun schon in einem ersten Schritt abgesichert. Der Client kann sich sicher sein, dass er mit dem richtigen Server verbunden ist, und beide Partner können sich sicher sein, dass niemand die Kommunikation beobachten oder gar verändern kann.

Basic Auth

Im Gegensatz zu *soap* bietet *Restify* genügend Flexibilität, um in den Request-Cycle einzugreifen und Basic Authentication zu nutzen. Das Modul *http-auth*[80] unterstützt sowohl „Basic Auth“ als auch das weniger verbreitete „Digest Auth“. Nach wenigen Konfigurationsschritten kann es in den normalen Request/Response Callback integriert werden. Da *Restify* globale Handler unterstützt, die für jede Route durchlaufen werden, wäre das auch eine geeignete Stelle, um die Anwendung entsprechend abzusichern. Und an dieser Stelle liegt auch schon das Problem im Zusammenhang mit *soap*: Dieses setzt sich nämlich selbst als exklusiver Handler für HTTP-Requests und es besteht keine Möglichkeit, einen weiteren Handler zwischenzuschalten.

[80] https://www.npmjs.com/package/http-auth

Ein kleiner Tipp am Rande: In Java müsste man nun auf Konzepte wie ThreadLocals zurückgreifen, um den Benutzernamen an Folgehandler weiterzugeben. JavaScript ist jedoch eine dynamische Sprache und wir können Objekte wie beispielsweise den Request einfach um weitere Attribute anreichern – völlig transparent für alle weiteren involvierten Komponenten.

```
var auth = require('http-auth');

var basic = auth({
  authRealm : 'secure stuff',
  // nodeuser/nodepassword
  authList : ['nodeuser:{SHA}u3/O9Eeb95tx3I2OyK3By5hH3yI=']
});

server.use(function(req, res, next) {
  basic.apply(req, res, function(username) {
    console.log(username + " did log in");
    req.authenticatedUser = username;
    return next();
  });
  console.log("access not granted");
  return next(false);  // Verarbeitung abbrechen
});
```

`apply()` prüft die Benutzerdaten im eingehenden Request und ruft den übergebenen Callback, wenn Name und Passwort verifiziert werden konnten. Für den Handler bedeutet das, dass er an den nächsten Handler übergibt. Waren die Zugangsdaten nicht korrekt, so schreibt *http-auth* entsprechende Fehlercodes in die Response und bereitet diese für den Rückversand an den Client vor. In diesem Fall darf nicht an den nächsten Handler übergeben werden, weil die Response nicht mehr weiterverarbeitet werden darf. Diese Thematik wird nochmals im Abschnitt 2.6.2.3 „Fehlermeldungen“ aufgegriffen.

Nun sind die ausgetauschten Nachrichten sicher und sowohl Server als auch Client „namentlich“ bekannt – eine gute, wenn nicht sogar ausreichende Ausgangslage für die meisten Services.

In der Verwendung von REST-Services wird man jedoch regelmäßig noch auf zwei weitere Anforderungen beziehungsweise Anwendungsszenarien stoßen:

- Die übermittelten Nachrichten sind gar nicht geheim, eine „kostspielige“ Verschlüsselung ist also nicht notwendig. Die Daten dürfen aber auch nicht von einem potenziellen Angreifer verändert werden können.
- Der Serviceanbieter möchte andere Anwendungen integrieren, eventuell sogar komplett auf eine eigene Benutzerverwaltung verzichten.

HMAC

HMAC ist ein „hash based message authentication code“ – im Wesentlichen eine Art digitale Unterschrift. An dieser Stelle sollen keine Implementierungsdetails erläutert werden, da man diese bei Bedarf und Interesse im Internet[81] nachlesen kann, nur die grundlegende Funktionalität soll kurz beschrieben werden:

[81] http://en.wikipedia.org/wiki/Hash-based_message_authentication_code

Sender und Empfänger teilen sich ein „Geheimnis“, besitzen also einen gemeinsamen Schlüssel. Der Sender verknüpft die Nachricht mit dem Schlüssel und gegebenenfalls noch weiteren Informationen und errechnet einen Hash-Wert zu dieser Verknüpfung. Dieser Wert wird dann mit der Nachricht im HTTP-Header an den Empfänger gesendet. Der Sender führt dieselbe Verknüpfung und Hash-Wert-Berechnung durch und prüft, ob sein Hash-Wert mit dem übertragenen identisch ist. Hier lässt sich auch schon der Unterschied zu einer echten digitalen Signatur erkennen: Sender und Empfänger sind beide in der Lage, denselben HMAC zu produzieren. Wer von beiden eine Nachricht tatsächlich erstellt hat, kann also nicht ermittelt beziehungsweise bewiesen werden.

Da sich beide Kommunikationspartner ein Geheimnis teilen, muss man sich Gedanken dazu machen, wie der Schlüssel sicher verteilt werden kann und ob das Vorgehen überhaupt für einen Betrieb mit vielen Konsumenten geeignet ist. Die Anfragen an Amazons Cloudservices werden beispielsweise über HMAC authentifiziert[82]. Der notwendige Schlüssel (API-Key) kann nach der Anmeldung im Entwicklerportal über eine gesicherte Verbindung geladen werden. Die Anwendung muss dann zur Laufzeit Zugriff auf den Schlüssel haben, um beispielsweise Anfragen an Amazons SimpleDB zu authentifizieren. Eine weitere Anmeldung ist nicht mehr notwendig. Alternativ ist ein Szenario denkbar, in dem kurzlebige Schlüssel nur für die Dauer einer Sitzung ausgetauscht werden. Allerdings muss dann immer zu Beginn einer Sitzung eine Anmeldung am Server erfolgen.

Die Erzeugung eines HMAC ist aus technischer Sicht eigentlich trivial und kann mit Node.js-Bordmitteln erledigt werden. Interessanter ist jedoch die Frage, welche Informationen in den zu berechnenden Hash-Wert einfließen sollen. Es gibt lediglich einen Standard für die Errechnung des Codes[83] basierend auf einer gegebenen Information, nicht aber dafür, welche Information sinnvoll ist. Diese ist im Wesentlichen von Anwendungseigenschaften abhängig und kann nicht allgemeingültig vorgegeben werden. In die Berechnung muss alles einbezogen werden, was nicht verändert werden darf. Das ist mindestens die Nachricht selbst, das HTTP-Verb und die URL der Ressource. Werden im HTTP-Header noch weitere relevante Daten übertragen, so müssen auch diese mit einbezogen werden.

Im nachfolgenden Beispiel werden neben Nachricht, Verb und URL noch der Benutzername und ein Zeitstempel im Header übertragen. Die Daten, für die der Hash-Wert errechnet werden soll, müssen übrigens vorher in eine kanonische Form gebracht werden, so dass auf beiden Seiten tatsächlich derselbe Wert entstehen kann. Sollten sich Client und Server gemeinsamen Utility-Code teilen können, ist das trivial. Ansonsten muss genau definiert werden, in welchem Format und in welcher Reihenfolge die einzelnen Informationsfragmente zusammengesetzt werden müssen. Eine derartige Definition findet man beispielsweise bei Amazon im Kontext der Amazon-Webservices[84]. Üblicherweise eine der Implementierungstätigkeiten, für die man sich einen üppigen Puffer einplanen sollte, wenn es keine fertige Bibliothek gibt.

Jeder eingehende Request wird in einem globalen Handler auf einen gültigen HMAC geprüft. Stimmen übertragener und berechneter HMAC überein, wird an den nächsten Handler in der Kette delegiert. Ansonsten wird die Verarbeitung mit einem entsprechenden HTTP-Fehlercode abgebrochen.

---

[82] http://docs.aws.amazon.com/AmazonS3/latest/dev/RESTAuthentication.html

[83] https://tools.ietf.org/html/rfc2104

[84] http://docs.aws.amazon.com/AmazonSimpleDB/latest/DeveloperGuide/HMACAuth.html#REST_RESTAuth

```
var hmac = require('./rest_hmac.js').createHMAC();

server.use(function(req, res, next) {
  if (hmac.verifyRequest(req)) {
    return next();
  } else {
    res.send(400, new Error('hmac invalid'));
    return next(false);
  }
});
```

Die eigentlichen Berechnungsfunktionen sind in diesem Beispiel in ein Modul (`rest_hmac.js`) ausgelagert und stehen über ein Objekt zur Verfügung, das mittels `createHMAC()` erzeugt wird. Optional können bei der Erzeugung des HMAC-Objekts noch Werte für den zu verwendenden Algorithmus und das gemeinsame Geheimnis übergeben werden.

```
HMAC.prototype.createHash =
  function createHash(method, url, timestamp) {
    var canonicalForm = method+"#"+url+"#"+timestamp;
    var hmac = crypto.createHmac(this.options.algorithm,
                                 this.options.secret);
    hmac.update(canonicalForm);
    var calculatedHash = hmac.digest("hex");

    return calculatedHash;
  }
```

```
HMAC.prototype.verifyRequest = function verifyRequest(req) {
  var clientsideHash = req.headers["hmac"];
  var clientsideTimestamp = req.headers["timestamp"];
  var currentTimestamp = Math.round(Math.round(new Date().getTime() / 1000));
  var requestAge = currentTimestamp-clientsideTimestamp;

  var calculatedHash = this.createHash(req.method,
                                       req.url,
                                       clientsideTimestamp);

  return ((clientsideHash === calculatedHash) &&
          (Math.abs(requestAge) < 10)
         );
}
```

Der im Code gezeigte Timestamp ist ein Unix-Timestamp – Millisekunden seit dem 1. 1. 1970 0:00 – und bezieht sich immer auf die Zeitzone GMT beziehungsweise UTC. Auch wenn Client und Server in verschiedenen Zeitzonen stehen, sollte die Prüfung also korrekt durchführbar sein. Allerdings muss darauf geachtet werden, dass die Uhren beider Systeme hinreichend synchron laufen und bei der Definition des „maximalen Request-Alters“ gegebenenfalls die Netzlaufzeit beachtet wird.

OAuth

OAuth[85] wird meistens benutzt, wenn eine Anwendung im Auftrag des Benutzers Funktionen einer anderen Anwendung verwenden möchte oder sogar der komplette Anmeldevorgang über eine andere Anwendung abgewickelt werden soll. Die erste Version von OAuth war noch sehr auf den Einsatz innerhalb von Browseranwendungen fokussiert, die Version 2 ist darüber hinaus auch für die Kommunikation zwischen Maschinen beziehungsweise nativen Anwendungen geeignet. Populäre Beispiele für solche Szenarien sind

- ein Blog, in welchem aktuelle Bilder aus dem Flickr-Fotostream des Benutzers ausgegeben werden,
- eine Anwendung, die Trainingsdaten erfasst und das Ergebnis auf Twitter zur Verfügung stellt,
- Anwendungen, die es zulassen, sich mit Twitter- oder Facebook-Accounts anzumelden.

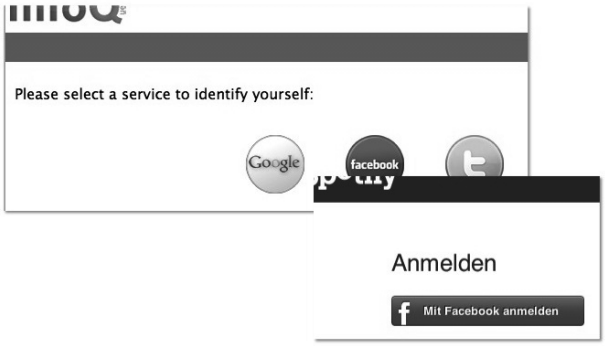


Bild 2.17 „Please login with …“

Um ein besseres Gefühl für mögliche Einsatzbereiche zu bekommen, wird an dieser Stelle das OAuth-Protokoll kurz erläutert. Nicht in jedem Detail, aber doch genau genug, um das Zusammenspiel der involvierten Partner zu erkennen. Wer sind nun die involvierten Partner bei OAuth?

- Der Anwender, der den Zugriff auf seine Daten erlauben möchte,
- die Anwendung, die auf die Daten zugreifen möchte,
- der Server, der die Daten des Anwenders hält.

Anwendung und Server müssen initial miteinander bekanntgemacht werden. Das geschieht jenseits einer konkreten Anfrage, an sich schon, bevor die Anwendung „live“ geht, und nicht im Kontext eines Userrequests. Noch zur Entwicklungszeit werden ein API Key und eine Client-ID angefragt und eine Redirect-URL auf dem Server hinterlegt – Letztere wird in wenigen Augenblicken noch erklärt.

[85] http://oauth.net/

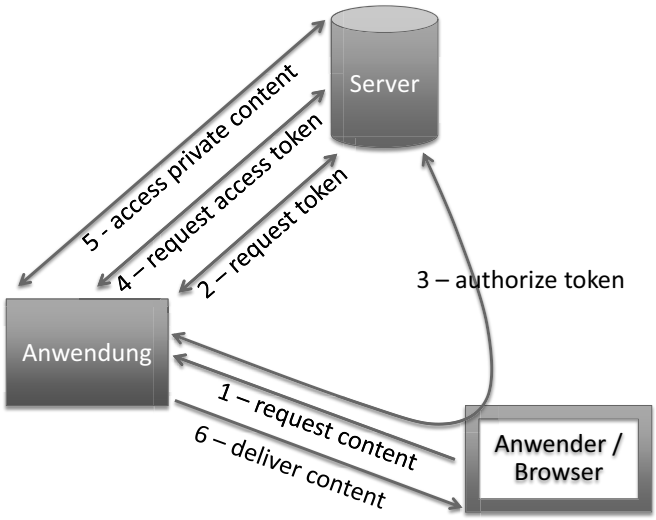


Bild 2.18 OAuth

Erfolgt nun der erste Request an die Anwendung, für den der Server angefragt werden muss (1), so lässt sich die Anwendung vom Server für den anfragenden Anwender ein Request Token ausstellen (2). Der Server stellt nicht nur das Token zur Verfügung, sondern auch eine Redirect-URL. Über diese Redirect-URL wird der Browser des Anwenders auf eine Login-Seite des Servers gelenkt, wobei das Token ebenfalls im Request mitgeschickt wird (3). Auf dem Server erwarten den Benutzer eine Login-Seite und der Hinweis, dass die Anwendung Datenzugriff erbeten hat. Dabei können natürlich beliebige Abstufungen eingeführt werden, eventuell soll ja nicht jede Anwendung den kompletten Funktionsumfang nutzen dürfen. Hat der Anwender auf dem Server seine Auswahl bestätigt, so antwortet der Server mit der eingangs erwähnten Redirect-URL und der Anwender wird wieder in die nunmehr autorisierte Anwendung geleitet. Mit dem Request Token ist es nun möglich, ein weiteres Token, das Access Token anzufordern (4). Mit dem Access Token ist es der Anwendung jetzt möglich, tatsächlich Anfragen im Namen des Anwenders an den Server zu stellen (5) und letztendlich den gewünschten Inhalt an den Client zu liefern beziehungsweise Funktionen des Servers zu nutzen (6).

Auf den ersten Blick erscheint die Anzahl der Tokens unnötig hoch, aber durch die unterschiedlichen Tokens ist es möglich, für verschiedene Funktionalitäten verschiedene Gültigkeitsdauern zu vereinbaren und die Autorisierung bei Bedarf auch wieder zu entziehen. Der gezeigte OAuth-Ablauf nennt sich „Authorization Code“ und stellt die vollständigste Variante dar, wie sie mit Webanwendungen verwendet wird. Für andere Arten von Anwendungen wie beispielsweise native Anwendungen stehen noch andere Profile bereit, in denen jedoch sensible Daten in den Anwendungen gespeichert werden müssen.

Aus technischer Sicht gibt es im OAuth-Umfeld somit zwei wesentliche Komponenten, die im Rahmen der Anwendungsentwicklung eine Rolle spielen können: Wird ein Server implementiert, so muss dieser in der Lage sein, Tokens auszustellen und eventuell Zugriffe einer Anwendung zuzulassen. Wird hingegen die Anwendung implementiert, so müssen die Tokens beantragt und der Zugriff im Namen des Anwenders durchgeführt werden. Da in

einem stark vernetzten Umfeld durchaus beides der Fall sein kann, müssen beide Seiten gleichermaßen betrachtet werden.

Für die OAuth-Unterstützung auf dem Server stehen einige wenige Node-Module zur Verfügung. Allerdings sind alle vielversprechenden Implementierungen in Form einer *Express*- oder *connect*-Middleware umgesetzt und werden deshalb nicht in allen technischen Details in diesem Abschnitt behandelt. Eine genauere Beschreibung, was eine Middleware ist und wie sie funktioniert, kann in „Middleware Framework Connect“ nachgelesen werden.

2.6.3 XML-Verarbeitung

Wird mit Webservices kommuniziert, so kommt man früher oder später auf jeden Fall mit XML-Strukturen in Kontakt. Bei SOAP-Services ergibt sich das automatisch durch das zugrunde liegende Datenformat, bei REST-Services hängt es davon ab, welche Art von Content der Service anbietet bzw. anbieten muss. Darüber hinaus ist XML-Verarbeitung natürlich auch an einer Vielzahl anderer Stellen gefragt, beispielsweise beim Einlesen von Konfigurationsdaten. Die nächsten Abschnitte werden sich deshalb den verschiedenen Möglichkeiten widmen, lesend und schreibend mit XML umzugehen.

2.6.3.1 XML-Parsing

Beim Parsen von XML-Strukturen unterscheidet man zwischen zwei grundsätzlichen Vorgehensweisen:

- *Dokumentenbasiert:* Das XML-Dokument wird eingelesen und steht in der Anwendung als Ganzes und in abstrahierter Form zur Verfügung. Navigiert wird ausgehend vom Wurzelknoten über eine API, die Zugriff auf Kindknoten und Attribute bietet. Diese Variante ist unter dem Namen DOM bekannt.
- *Ereignisbasiert:* In der ereignisbasierten Verarbeitung hat die Anwendung keinen Zugriff auf die komplette XML-Struktur, um frei zu navigieren. Beim sequenziellen Parsen des Dokuments werden jedoch Events erzeugt, auf die die Anwendung reagieren kann. Diese Art der Verarbeitung wird mit SAX bezeichnet, eine weitere Variation ist unter StAX bekannt.

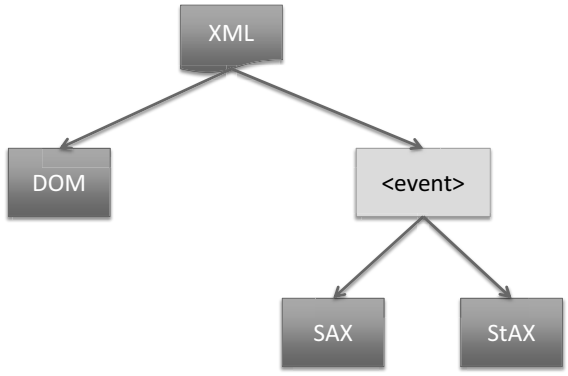


Bild 2.19 XML-Parsing

In unterschiedlichen Anwendungsszenarien spielen die beiden Herangehensweisen ihre Vor- und Nachteile aus und es ist nicht selten so, dass nur eine der beiden Varianten wirklich sinnvoll ist. Node.js bietet uns natürlich sowohl für DOM als auch für SAX Unterstützung.

2.6.3.1.1 Dokumentenbasierte Verarbeitung

Die dokumentenbasierte Verarbeitung ist eine sehr intuitive Art, mit XML-Dokumenten umzugehen. Im Grunde bildet sie im Code nach, was man normalerweise macht, wenn man als Mensch etwas in einer XML-Struktur sucht: Man hangelt sich über Eltern-Kind-Beziehungen von Knoten zu Knoten und wertet Inhalte und Attribute aus. Damit ergibt sich auch schon der erste Vorteil der API, nämlich dass sie sehr intuitiv zu benutzen ist. Eine einfache API alleine sollte aber natürlich nicht der ausschlaggebende Grund sein, sich für eine Technologie zu entscheiden.

Wann sollte man also dokumentenbasierte Verarbeitung verwenden? Ein wesentlicher Charakterzug des DOM-Ansatzes ist, dass die zu verarbeitenden Dokumente komplett in den Speicher geladen werden und die Logik anschließend den Parser steuert. Damit ergibt sich für die Anwendung die Möglichkeit, sich völlig frei im Dokument zu bewegen oder auch an verschiedenen, sich eventuell gegenseitig referenzierenden Stellen gleichzeitig zu arbeiten. Außerdem obliegt es natürlich der Anwendung, zu entscheiden, wann verschiedene Verarbeitungsschritte stattfinden, beziehungsweise man kann diese Schritte von beliebigen Ereignissen abhängig machen, wie beispielsweise eingehende Anfragen.

Nachdem man immer von einem Knoten aus navigiert, besteht zudem die Möglichkeit, innerhalb der Anwendung bei Bedarf einen Verarbeitungskontext aufzubauen. Gelangt man beispielsweise zu einem bestimmten Element, so kann man die weitere Verarbeitung davon abhängig machen, ob vorher eventuell ein bestimmtes umgebendes Element mit einem passenden Attribut gefunden wurde.

Auch bei der gleichzeitigen Verarbeitung von mehreren XML-Dokumenten ist DOM klar im Vorteil. Durch den wahlfreien Zugriff auf die Daten ist es problemlos möglich, passende Elemente in verschiedenen Dokumenten zu suchen, um anschließend eine gemeinsame Verarbeitung durchzuführen.

Was letztendlich nicht verschwiegen werden sollte, ist die Tatsache, dass DOM auch schreibenden Zugriff auf das XML-Dokument bietet. Beim Durchwandern der XML-Strukturen ist man nicht darauf beschränkt, Werte und Attribute auszulesen. Genauso einfach können wir bestehende Elemente verändern, neue Elemente einführen und letztendlich die Struktur wieder in eine Datei serialisieren oder beispielsweise als Antwort auf einen HTTP-Request zurückliefern. Aber das wird erst in Abschnitt 2.6.3.2 („XML-Erzeugung und -Veränderung“) gezeigt, nun geht es zunächst um das Einlesen von XML-Daten.

Aus der Vielzahl an verfügbaren Modulen wurde an dieser Stelle *libxmljs*[86] herausgegriffen, um die dokumentenbasierte Verarbeitung von XML-Strukturen zu zeigen. Zu Anfang wird das XML-Dokument als String an den Parser übergeben und via XPath-Ausdruck nach Tags mit dem Namen „author“ gesucht.

[86] https://www.npmjs.com/package/libxmljs

```
var xmlDoc = libxmljs.parseXml(xmlString);
var authorElems = xmlDoc.find(
  "//nb:author",
  {nb: "http://nodebook.de/samples/"}
);
```

Im Beispieldokument wird mit Namespaces gearbeitet. In der Suche wird deshalb der Tag-Name inklusive Namespace-Präfix angegeben und der Namespace selbst ebenfalls übergeben. Das Präfix muss übrigens nicht mit dem im Originaldokument identisch sein.

Anschließend iteriert man über die gefundenen Elemente – wenn es denn welche gab – und liest das XML-Attribut namens „id“ aus. Es wird als Schlüssel verwendet, um eine interne Map aufzubauen, in der die Autorennamen liegen. Anschließend werden über den Parser alle Kindelemente des aktuellen Tags ermittelt und geprüft, ob es sich um das Tag `<vorname/>` oder `<nachname/>` handelt. Falls ja, wird der entsprechende Wert übernommen.

```
for (i=0; i<authorElems.length; i++) {
  authorElem = authorElems[i];

  authorId = authorElem.attr("id").value();
  authors[authorId] = {"fn":"n/a", "ln":"n/a"};

  nameElems = authorElem.childNodes();

  for (j=0; j<nameElems.length; j++) {
    if (nameElems[j].name() == "firstname") {
      authors[authorId]["fn"] = nameElems[j].text();
    }
    if (nameElems[j].name() == "lastname") {
      authors[authorId]["ln"] = nameElems[j].text();
    }
  }
}
```

Im letzten Schritt wird das Dokument nach Kapiteleinträgen durchsucht, um letztendlich auszugeben, wer welches Kapitel geschrieben hat.

```
var chapterElems = xmlDoc.find(
  "//nb:chapter",
  {nb: "http://nodebook.de/samples/"}
);
for (i=0; i<chapterElems.length; i++) {
  chapterElem = chapterElems[i];

  chapterNum = chapterElem.attr("num").value();
  authorRef = chapterElem.attr("authRef").value();
  chapterTitle = chapterElem.text();

  console.log("Chapter #"+chapterNum+" is called '"
    +chapterTitle
    +"' and was written by "
    +authors[authorRef]["fn"]+" "
    +authors[authorRef]["ln"]);
  }
```

Da das Dokument nicht sequenziell durchsucht werden muss, sondern wahlfrei zugegriffen wird, ist es kein Problem, dass die Information im XML-Dokument in anderer Reihenfolge abgelegt war. Im folgenden Abschnitt ist zu sehen, dass das zwar bei der ereignisgesteuerten Verarbeitung natürlich auch möglich ist, aber ein wenig aufwendiger. Und je größer und komplexer die zu verarbeitenden Strukturen sind, desto mehr macht sich dieser Effekt bemerkbar.

Im obigen Beispiel wurde auf Fehlerbehandlung verzichtet. Jedes `get()` oder `find()` sollte natürlich auch dahingehend überprüft werden, ob tatsächlich entsprechende Elemente gefunden wurden. Steht für das zu verarbeitende Dokument eine Schemadefinition zur Verfügung, so kann man auf die Prüfungen wenigstens für im Schema als verpflichtend markierte Elemente tatsächlich verzichten.

```
var xmlString = fs.readFileSync("node-book.xml").toString();
var xmlDoc = libxmljs.parseXml(xmlString);

var xsdString = fs.readFileSync("node-book.xsd").toString();
var xsdDoc = libxmljs.parseXml(xsdString);

if (xmlDoc.validate(xsdDoc) == false) {
  throw new Error("invalid xml!");
}
```

Die XSD-Beschreibung des Dokuments wird hierzu analog zum eigentlichen XML-Dokument eingelesen und der Validierungsfunktion `validate()` übergeben. Erhält man `true` als Ergebnis zurück, so kann man im weiteren Verlauf davon ausgehen, dass alle als Pflichtfelder markierten Elemente auch gefunden werden.

Die Verarbeitung in der DOM-Variante bedingt, dass das Dokument komplett in den Speicher gelesen wird. Für große Dokumente kann das natürlich bezüglich der Ressourcenauslastung auf dem Server ein Problem darstellen und im B2B-Umfeld entstehen beim Abgleich von Systemen mitunter sehr große Datenstrukturen. In solchen Fällen ist es effizienter, nur auf Teilausschnitten der Daten zu arbeiten und Techniken wie Streaming und Chunking in Betracht zu ziehen. Das Arbeiten auf Teilausschnitten ist wiederum Grundprinzip der ereignisbasierten Verarbeitung von XML-Dokumenten, welches im nächsten Abschnitt vorgestellt wird.

2.6.3.1.2 Ereignisbasierte Verarbeitung

Aus Implementierungssicht ist die API für ereignisbasierte XML-Verarbeitung noch einfacher als die im letzten Abschnitt vorgestellte Schnittstelle. Die grundlegende Idee ist, dass dem Parser ein XML-Dokument übergeben wird und dieser dann während der Verarbeitung verschiedene Ereignisse erzeugt. Die Verwendung von Ereignissen – also Events – ist in JavaScript bzw. Node.js ein gängiger Mechanismus, dem man in diesem Buch ja ebenfalls ständig begegnet.

Konzeptionell ist diese Art der Verarbeitung aber etwas anspruchsvoller, weil man sich im Code nach dem Parser und damit nach der Dokumentenstruktur richten muss und nicht umgekehrt. Nachfolgend werden zwei Module betrachtet, die ereignisbasiertes Parsing unterstützen:

- *node-expat*[87], welches auf der nativen Bibliothek Expat basiert und
- *sax*[88] in einer reinen JavaScript-Implementierung.

Beide Module funktionieren ähnlich. Ereignisse werden erzeugt, wenn im XML-Dokument vom Parser bestimmte Konstrukte gefunden werden, wie beispielsweise der Start oder das Ende eines Tags, ein Attribut oder Text. Anschließend obliegt es der Anwendung, diese Ereignisse in den richtigen Kontext zu setzen und zu interpretieren.

Daraus ergeben sich schon erste Hinweise, wann diese Art des Parsens an sich geeignet ist. Da sich der Parser sequenziell durch das Dokument arbeitet, ohne sich Status oder Kontext zu merken, ist er sehr ressourcenfreundlich. Unter der Annahme, dass er ordentlich implementiert ist, werden Speicherverbrauch und Durchsatzgeschwindigkeit nicht von der Größe der XML-Dokumente beeinflusst. Wie bereits erwähnt, kann ja alleine schon die Größe der zu verarbeitenden Daten dazu führen, dass die DOM-Variante nicht verwendbar oder zumindest sehr ineffizient ist.

Spielt die Größe des Dokuments keine wesentliche Rolle, so sollte man prüfen, wie komplex die Strukturen im Dokument sind und wie viel zusätzliche Information man vorhalten muss, um die Bedeutung eines gefundenen Knotens zu interpretieren.

Beide Bibliotheken sind zudem in der Lage, sogar mit Streams zu arbeiten. Auch das kann – abhängig vom jeweiligen Use Case – ein wichtiges Feature sein.

```
var util = require('util');
var fs = require('fs');
var expat = require("node-expat");

var xmlString = fs.readFileSync("node-book.xml").toString();
var parser = new expat.Parser("UTF-8");

var currentAuthorId = null;
var insideFirstname = false;
var insideLastname = false;
var authors = {};

parser.addListener('startElement', function(name, attrs) {
  if (name == "nb:author") {
    currentAuthorId = attrs["id"];
    authors[currentAuthorId] = {"fn":"n/a", "ln":"n/a"};
  }
  if (name == "nb:firstname") {
    insideFirstname = true;
  }
  if (name == "nb:lastname") {
    insideLastname = true;
  }
});

parser.addListener('text', function(value) {
  if ((currentAuthorId != null) && insideFirstname) {
    authors[currentAuthorId].fn = value;
  }
  if ((currentAuthorId != null) && insideLastname) {
```

[87] https://www.npmjs.com/package/node-expat
[88] https://www.npmjs.com/package/sax

```
    authors[currentAuthorId].ln = value;
  }
});
```

Das angeführte kleine Beispiel zeigt bereits den Nachteil, den die ereignisbasierte Verarbeitung mit sich bringt. In dem Moment, in dem ein Textknoten gefunden wird, gibt uns der Parser keine weitere Information, wo dieser Knoten gefunden wurde, was er also bedeutet. Im Beispiel wird dieses Problem einfach mit Hilfe von globalen Variablen gelöst – wenig elegant und für komplexere Dokumente sehr schnell unübersichtlich. Was ebenfalls auffällt, ist die Reihenfolge, in der man die Daten aus dem Dokument extrahiert bekommt. Sie war nicht durch den Entwickler gesteuert, sondern durch den Parser vorgegeben und insofern eventuell dem eigentlichen Logikfluss genau entgegengesetzt. Das wiederum kann dann zusätzlich zu umständlichen und schwer wartbaren Konstrukten im Code führen.

Bild 2.20 Eventbasierter XML-Parser

Die Grafik verdeutlicht nochmals, welche Ereignisse in welcher Reihenfolge vom Parser gemeldet werden. Die Reihenfolge der Events entspricht unserem normalen Leseverhalten – von links nach rechts, zeilenweise von oben nach unten. Ein grüner Stern steht für das Ereignis „startElement“, ein gelber für „text“ und ein roter für „endElement“. Ob nun ein „text“-Ereignis für einen Vornamen oder eine Kapitelüberschrift steht, muss die Anwendung aber selbst herausfinden.

Das im Beispiel gezeigte *node-expat* ist vergleichsweise schnell, da es auf einer nativen Implementierung eines XML-Parsers beruht. Das bedeutet aber auch, dass auf dem Zielsystem diese Bibliothek vorhanden sein muss oder wenigstens im Zuge der Auflösung von Abhängigkeiten gebaut werden können muss. Es bietet auch nur die drei genannten Events und keinerlei weitere Unterstützung für Namespaces oder Schemavalidierung.

Eine weitere prominente Bibliothek ist *sax* – ein Name, den man sicherlich auch aus anderen Programmiersprachen kennen könnte. Es handelt sich um eine reine JavaScript-Implementierung, die entsprechend ohne weitere Vorkehrungen in jeder JavaScript-Laufzeitumgebung funktioniert. Sie erzeugt deutlich mehr Ereignisse, da zusätzlich zu den einfachen

XML-Konstrukten auch Namespaces, Kommentare oder CDATA-Elemente behandelt werden. Die Verwendung ist aber der von *node-expat* sehr ähnlich und wird deshalb an dieser Stelle nicht weiter dargestellt.

Welcher der beiden Bibliotheken man nun den Vorzug geben möchte, hängt im Wesentlichen davon ab, auf welche XML-Elemente man reagieren möchte und wie wichtig im konkreten Fall die Geschwindigkeit des Parsers ist. Gegebenenfalls sollte man sich – vor allem im Falle eines PaaS-Deployments – Gedanken dazu machen, ob die für *node-expat* benötigte Bibliothek auf dem Zielsystem überhaupt verfügbar ist.

2.6.3.2 XML-Erzeugung und -Veränderung

Natürlich möchte man XML-Dokumente nicht nur lesen, sondern manchmal auch erzeugen oder verändern. Bedingt durch die zugrunde liegende Technik, können XML-Daten nur mit Bibliotheken erzeugt werden, die auf DOM basieren. Diese halten ein komplettes Modell des Dokuments im Speicher und man kann über die vorhandenen Methoden an jede beliebige Stelle des Dokuments navigieren. An der gewünschten Stelle angekommen, können bestehende Werte einfach geändert oder auch neue Knoten eingefügt werden.

Um den schreibenden Zugriff auf ein XML-Dokument zu zeigen, wird einfach das Beispiel aus dem Abschnitt 2.6.3.1.1 „Dokumentenbasierte Verarbeitung“ aufgegriffen und weitere Daten werden hinzugefügt, geändert und gelöscht.

Im ersten Schritt wird der Knoten „book“ um ein neues Attribut namens „isbn“ ergänzt:

```
bookElem = xmlDoc.get(
  "//nb:book",
  {nb: "http://nodebook.de/samples/"}
);
bookElem.attr("isbn", "978-3-00000-000-0");
```

Das Attribut ist in der Schemadefinition vorgesehen, also ist das XML-Dokument immer noch gültig.

Als Nächstes wird ein neuer Kindknoten von „book“ eingefügt, auf gleicher Ebene mit den Kapitel- und Autorenknoten:

```
bookElem.node("nb:pages", "1234");
```

Es ist auch möglich, statt eines Kindknotens Geschwisterknoten – sogenannte Siblings – in das Dokument einzusetzen. Diese tauchen dann auf der Ebene des Ausgangsknotens vor oder nach diesem auf. Die XML-Validierung sollte jetzt fehlschlagen, da dieser Knoten nicht in der XSD definiert ist. Also sollte er wieder entfernt werden:

```
xmlDoc = libxmljs.parseXml(xmlDoc.toString());
pagesElem = xmlDoc.get(
  "//nb:pages",
  {nb: "http://nodebook.de/samples/"}
);
pagesElem.remove();
```

Hier gibt es eine kleine, nicht dokumentierte Besonderheit in *libxmljs* – wahrscheinlich einfach ein Bug: Ein neu eingefügter Knoten wird zwar beispielsweise ausgegeben, wenn

das Dokument mit der Methode `toString()` geschrieben wird, er ist aber nicht über die Methoden `get()` oder `find()` auffindbar. Erst wenn das Dokument neu geparst wird, funktionieren diese Methoden wieder erwartungsgemäß. Vermutlich wird im Hintergrund ein internes Modell des Dokuments gehalten, welches leider nicht aktualisiert wurde.

Neben dem Einfügen und Entfernen von Knoten beziehungsweise Attributen können bestehende Elemente natürlich auch verändert werden. Dies ist sowohl für die Namen der Elemente als auch für deren Inhalt möglich.

Abschließend bleibt zu erwähnen, dass es tatsächlich auch SAX-basierte Bibliotheken gibt, die in der Lage sind, XML-Dokumente zu schreiben oder zu verändern. Dies ist dann aber eine spezifische Erweiterung der eigentlichen Parser-Funktionalität und eher die Ausnahme. *libxmljs* bietet zum Beispiel nicht nur den gezeigten DOM-Parser, sondern gleichzeitig auch einen SAX-Parser.

2.6.3.2.1 (Nicht nur) Bei Webservice-Clients auf Performance achten

An dieser Stelle soll noch ein kleiner Hinweis mit auf den Weg gegeben werden: Es lohnt sich durchaus, hin und wieder an Performance zu denken – auch wenn man noch nicht direkt in Engpässe läuft.

An sich ist das natürlich keine Weisheit, die sich nur auf Webservices reduziert, aber SOAP-Clients sind eine Möglichkeit, die Problematik schön zu veranschaulichen. Sobald man sich in einer serverseitigen Anwendung befindet, macht auch Kleinvieh spürbar Mist. Ruft man aus einer Standalone-Anwendung einen Webservice, so geschieht das in der Regel aufgrund einer Nutzeraktion. Der Anwender füllt beispielsweise ein Formular aus und bestätigt seine Eingaben. Anschließend werden seine Daten über einen Webservice an einen Server geschickt. Wie geschickt oder ungeschickt ein Webservice-Client an dieser Stelle implementiert wurde, dürfte niemandem auffallen – das Ergebnis wird aller Wahrscheinlichkeit nach innerhalb einer Zeitspanne vorliegen, die dem Anwender angemessen erscheint.

Ganz anders verhält sich das auf der Server-Seite. Wenn wir uns den Code des Webservice-Aufrufs nochmals vor Augen halten, dann erkennen wir hier zwei Phasen:

- Erzeugung des Clients inklusive WSDL-Abruf und Parsing,
- Aufruf der Operation inklusive Marshalling und Unmarshalling von Datenstrukturen.

Je nachdem, welcher Webservice-Stack verwendet wird, kann die erste Phase sehr „teuer" sein und darf nicht für jeden Aufruf erneut durchgeführt werden, weil sich der Performanceverlust serverseitig vervielfacht. Statt eines einzigen, vom Anwender ausgelösten Aufrufs behandelt man unter Umständen Hunderte oder Tausende von Aufrufen pro Stunde und verbraucht entsprechend CPU-Zeit.

Es ist keinesfalls empfehlenswert, jede Zeile Code im Hinblick auf ihre CPU-Zyklen zu optimieren, weil man so in der Regel sehr viel Zeit für sehr wenig Ergebnis verschwendet – „*premature optimization is the root of all evil*"[89]. Aber man sollte sich durchaus ein Gespür für verdächtige Codepassagen aneignen und eine Anwendung entsprechend designen.

[89] Donald Knuth, http://pplab.snu.ac.kr/courses/adv_pl05/papers/p261-knuth.pdf, Seite 268

Die oben gezeigte Verwendung von SOAP-Clients ist übrigens unkritisch und muss innerhalb der eigenen Anwendung nicht weiter optimiert werden. Das Modul *soap* verwendet intern bereits einen Cache und reduziert die Ausführung der ersten Phase auf ein Minimum.

2.6.3.3 Exkurs: Ein (selbst unterschriebenes) Zertifikat erstellen

Nachdem Kommunikation über HTTP sowohl in diesem Abschnitt als auch bereits in den Abschnitten zur Erstellung einer Webanwendung beschrieben wurde, soll hier kurz auf das Thema Sicherheit und Zertifikate eingegangen werden.

Im Rahmen von Entwicklungsarbeiten ist es hin und wieder auch notwendig, Zertifikate einzusetzen, um SSL-gesicherte Kommunikation zu testen – in diesem Buch beispielsweise in den Abschnitten zu SOAP und REST, aber auch im Zusammenhang mit Webanwendungen basierend auf *express*. Da es zur Entwicklungszeit weniger um tatsächliche Sicherheit geht, werden in der Regel keine „echten" Zertifikate, sondern sogenannte „self-signed" Zertifikate erzeugt. Diese werden nicht von offiziellen Stellen wie beispielsweise VeriSign, Symantec oder Thawte[90] ausgestellt, sondern vom Entwickler selbst erzeugt. Das kann innerhalb weniger Minuten erledigt werden und ist zudem kostenlos. Um den Sprung nach Google überflüssig zu machen, werden hier die einzelnen Schritte vorgestellt und erklärt.

Zuerst zum Tooling: Zum Erstellen von Zertifikaten (und für vieles mehr aus diesem Umfeld) kann man das Tool openssl[91] verwenden. In der Regel ist es auf allen Unix-basierten Systemen bereits installiert, eine Windows-Version steht ebenfalls zum Download bereit. Unter Umständen ist es auf Unix-Systemen notwendig, Root-Rechte zu besitzen, um openssl benutzen zu können.

Kommunizieren Client und Server über SSL miteinander, so geschieht das aus Gründen der Performance mit einem symmetrischen Schlüssel. Es wird also derselbe Schlüssel verwendet, um auf der einen Seite Nachrichten zu kodieren und auf der Gegenseite zu dekodieren. Wenn auf beiden Seiten derselbe Schlüssel verwendet wird, stellt sich natürlich die Frage, wie dieser sicher ausgetauscht werden kann. Letztendlich hat vermutlich jeder Leser schon einmal das Online-Angebot seiner Bank wahrgenommen, aber noch nie in der Bankfiliale einen digitalen Schlüssel abgeholt und auf dem heimischen Rechner installiert. An dieser Stelle kommen asymmetrische Verschlüsselungsverfahren und Zertifikate ins Spiel. Bei asymmetrischen Verfahren basiert die Verschlüsselung auf einem Schlüsselpaar, einem privaten Schlüssel und einem öffentlichen Schlüssel. Der private Schlüssel verbleibt beim Eigentümer, also beispielsweise bei der Bank, wohingegen der öffentliche Schlüssel frei an alle Kunden verteilt werden kann. Ist der öffentliche Schlüssel beim Kunden angelangt, kann dieser Nachrichten entschlüsseln, die von der Bank verschlüsselt gesendet werden. Er kann allerdings auch – und das ist viel wichtiger – mit dem öffentlichen Schlüssel Nachrichten kodieren, die nur von der Bank mit dem privaten Schlüssel gelesen werden können. Eine SSL-Verbindung wird nun dadurch aufgebaut, dass der Browser auf Kundenseite den öffentlichen Schlüssel vom Server auf Bankseite anfordert. Anschließend erzeugt er für die Dauer der Sitzung einen symmetrischen Schlüssel, den er mit dem öffentlichen Schlüssel kodiert an den Server sendet. Nur die Bank ist in der Lage, den symmetrischen Schlüssel

[90] https://www.secure128.com/website-security-services/ssl-certificates/ssl-certificate-product-list.aspx
[91] http://www.openssl.org/

wieder zu lesen. Sie dekodiert ihn und verwendet ihn zur Sicherung der nachfolgenden Kommunikation.

Um eine Man-in-the-Middle-Attacke auszuschließen, muss der Kunde jedoch sicher sein, dass der öffentliche Schlüssel, den er von der Bank angefordert hat, auch tatsächlich von der Bank stammt. Hierfür wird im Rahmen der SSL-Kommunikation das Zertifikat verwendet. Das Zertifikat enthält allgemeine Angaben zum Eigentümer, unter anderem auch den Namen des Servers (`CN=www.my-online-banking.com`) und den öffentlichen Schlüssel des Servers. Außerdem ist es von einer vertrauenswürden Stelle (Root Certificate Authority, Root CA) unterzeichnet, um die Korrektheit der Daten zuzusichern.

Diesen letzten Schritt – die Unterzeichnung durch eine Root CA – möchte man sich zu Testzwecken normalerweise ersparen, weil er in der Regel langwierig und kostspielig ist. Deshalb verwendet man in solchen Fällen self-signed Zertifikate.

Im ersten Schritt wird ein RSA-Schlüssel, hier mit 1024 Bit, erzeugt. Für produktive Schlüssel sollte allerdings bereits auf 2048 Bit zurückgegriffen werden. Normalerweise wird das Schlüsselpaar noch durch ein Passwort geschützt. Für das vorliegende Beispiel ist das aber nicht notwendig und im Serverbetrieb sogar manchmal hinderlich, da das Passwort zur Verwendung des Schlüssels eingegeben werden muss. Der Schlüssel ist damit aber ungeschützt und darf nicht in fremde Hände gelangen.

```
$ openssl genrsa -out server.key 1024
```

Der RSA-Schlüssel wird nun verwendet, um einen Certificate Request zu erzeugen. In diesem Zuge müssen auch die oben erwähnten allgemeinen Angaben gemacht werden. Normalerweise würde der Request nun an eine Root CA geschickt, die daraus ein signiertes Zertifikat macht.

```
$ openssl req -new -key server.key -out server.csr
```

Im letzten Schritt wird aus dem Certificate Request ein Zertifikat erzeugt. In diesem Fall wird zudem eine Gültigkeit von 365 Tagen hinterlegt, was für Testzwecke ausreichen sollte.

```
$ openssl x509 -req -days 365 -in server.csr -signkey server.key -out server.crt
```

Zertifikat (`server.crt`) und Schlüssel (`server.key`) können nun einem HTTPS-Server als Startoptionen übergeben werden. Natürlich wird ein Client das Zertifikat nicht ohne Warnung akzeptieren, da er ja die Authentizität nicht prüfen kann.

Um eine SSL-geschützte URL mit dem Tool cURL abzurufen, müssen weitere Parameter in den Aufruf aufgenommen werden. Wäre die Ressource aus Abschnitt 2.6.2.1.1 mit dem oben erzeugten Zertifikat geschützt, dann lautete der richtige cURL-Aufruf wie folgt:

```
$ curl -k -s -H 'accept-version: 2.0.*' -w "\nHTTP: %{http_code}\n" https://
localhost:8080/versioned/service
```

Wichtig ist die Angabe des HTTPS-Protokolls, da ansonsten überhaupt kein SSL-Handshake durchgeführt wird. Zudem ist auch noch der Parameter „`-k`“ wichtig. Er sorgt dafür, dass cURL auch selbst signierte Zertifikate akzeptiert. Ansonsten würde cURL nur eine leere Antwort zurückmelden. Wird auch noch „`-v`“ übergeben, so werden Details zum SSL-Handshake ausgegeben.

Bild 2.21 Selbst signierte Zertifikate

Sollte man noch mehr Informationen zum SSL-Handshake und zur Verschlüsselung benötigen, so kann man sich alternativ auch über openssl verbinden. Der Befehl `openssl s_client -connect localhost:8080` baut eine Verbindung auf, über die anschließend das HTTP-Protokoll wie in einer Telnet-Sitzung manuell ausgeführt werden muss: `GET /versioned/service HTTP/1.1`.

2.7 Clustering

Node.js ist sehr schlank und performant. Damit ist es zwar in der Lage – bei passender Problemstellung –, eine deutlich höhere Last abzuarbeiten als beispielsweise eine gewöhnliche JEE-Anwendung, aber trotzdem gerät man in größeren Systemen schon auch mal an die Grenzen dessen, was Node.js leisten kann. Mehr als eine Instanz zu betreiben, erscheint also naheliegend. Ferner arbeitet Node.js ganz intensiv mit einem einzigen Thread, auch wenn die Bezeichnung „single-threaded" eventuell etwas zu plakativ ist. Da jedoch Server in der Regel eine Vielzahl von CPU-Kernen besitzen, wäre es tatsächlich Ressourcenverschwendung, nur eine Node.js-Instanz pro Server zu betreiben.

Mit dem Modul *cluster*[92] ist bereits seit Version 0.6 beziehungsweise in überarbeiteter Form seit Version 0.8 ein Clustering-Modul direkt in Node.js enthalten und kann somit out-of-the-box verwendet werden. Hierdurch wird ein Multiinstanzbetrieb immens vereinfacht.

Das folgende simple Beispiel zeigt einen Webserver, der alle verfügbaren CPUs beziehungsweise Kerne des Systems nutzt:

[92] http://nodejs.org/api/cluster.html

Index

N

O

P

Q

R

S

T

V

W

X

Z