12. Mrz 2013, Martin Helmich, Webentwicklung (https://www.mittwald.de/blog/webentwicklung-webdesign/webentwicklung), Webentwicklung/-design (https://www.mittwald.de/blog/webentwicklung-webdesign)

RESTful Webservices (1): Was ist das überhaupt?

Einer der Begriffe, die Entwickler in letzter Zeit immer wieder zu hören bekommen, ist der Begriff des **RESTful Webservices**. Zahlreiche große Internet-Unternehmen, wie z. B. **Twitter** (https://dev.twitter.com/docs/api), **Facebook** (http://developers.facebook.com/docs/reference/api/) oder **Google** (https://developers.google.com/custom-search/v1/using_rest) bieten eigene REST-Schnittstellen zu ihren Diensten an. Was hat es also mit den Begriffen **REST**, **RESTful HTTP** und **REST-Webservices** auf sich und wie können Entwickler solche Webservices mit den üblichen Hausmitteln (wie beispielsweise **TYPO3 Flow**) selbst implementieren?

Einleitung

Auf den ersten Blick besteht das WWW nur aus den **Komponenten**, die ein menschlicher Benutzer auch **wahrnehmen** kann – die Webseiten und -applikationen, die beispielsweise über den Browser aufgerufen werden können. Tatsächlich aber greifen über diese Infrastruktur **nicht nur menschliche Benutzer** auf Inhalte zu, sondern es gibt auch **„maschinelle" Benutzer**, also beispielsweise andere Anwendungen, die auf Daten aus dem Internet zugreifen. Ein Betreiber kann somit einen bestimmten Dienst im Internet anderen Anwendungen zur Verfügung stellen – diese Dienste werden **Webservices** genannt. Ein ganz einfaches Beispiel ist etwa **Twitter**: über einen entsprechenden Webservice können beliebige Anwendungen – vorherige Authorisierung vorausgesetzt – im Namen eines Benutzers Tweets auslesen oder schreiben).

Anfang des letzten Jahrzehnts war das Protokoll der Wahl für solche Webservices das **Simple Object Access Protocol** – kurz **SOAP**. Durch die Verwendung von SOAP konnte ein Client die Methoden eines serverseitigen Objekts aufrufen – das Ganze funktionierte dabei über recht komplexe **XML-Nachrichten**, die über HTTP ausgetauscht wurden. Die Kombination aus XML und HTTP machte das ganze Protokoll **programmiersprachen**– und **plattformunabhängig**; somit konnte beispielsweise ein in C++ geschriebener Client Methoden von serverseitigen Java-Objekten aufrufen – insgesamt also eigentlich eine recht praktische Sache.

Blog durchsuchen

Kategorien +

Lerne das Blog-Team kennen (https://www.mittwald.de/blog/das-blog-team)

Ähnliche Beiträge

1. Google Analytics Alternative:

Nach einiger Zeit fielen jedoch die Schwächen von SOAP auf: die komplizierten XML-Nachrichten hatten einen recht großen Overhead (viele Metadaten, wenig Nutzdaten), außerdem war das Zusammen- und Auseinanderbauen der XML-Nachrichten rechenintensiv. Hinzu kam, dass mittlerweile auch die großen Softwarehersteller SOAP für sich entdeckt hatten und den anfänglich (relativ) unkomplizierten Standard um zahlreiche weitere Sub-Standards ergänzt hatten (und natürlich unterstützte kaum ein Hersteller die Standards der anderen Hersteller, was die Sache noch weiter verkomplizierte). Mittlerweile gibt es mehrere hundert dieser sogenannten „WS-*"-Standards (http://de.wikipedia.org/wiki/WS-*), und einen richtigen Überblick darüber hat eigentlich niemand mehr.

Grundprinzipien von REST-Architekturen

„Das muss doch auch irgendwie einfacher gehen", fragten sich die Webservice-Entwickler irgendwann – und fanden die Antwort im REST-Architekturstil (kurz für Representational State Transfer). Der Begriff wurde im Jahr 2000 von Roy Fielding geprägt (http://jpkc.fudan.edu.cn/picture/article/216/35/4b/22598d594e3d93239700ce79bce1/7ed3ec2a-03c2-49cb-8bf8-5a90ea42f523.pdf). Prinzipiell ist der Architekturstil selbst unabhängig von irgendwelchen konkreten Protokollen – in der Regel wird zur Implementierung solcher Architekturen allerdings das altbekannte HTTP-Protokoll verwendet.

Grundsätzlich dreht sich in einer REST-Architektur alles um Ressourcen. Um beim Twitter-Beispiel zu bleiben, könnte beispielsweise jeder Benutzer und sogar jeder einzelne Tweet als eigene Ressource betrachtet werden). Diese Ressourcen sollten laut Fielding folgende Anforderungen erfüllen:

Adressierbarkeit: Jede Ressource muss über einen eindeutigen Unique Resource Identifier (kurz URI) identifiziert werden können. Ein Kunde mit der Kundennummer 123456 könnte also zum Beispiel über die URI http://ws.mydomain.tld/customers/123456 adressiert werden.

Zustandslosigkeit: Die Kommunikation der Teilnehmer untereinander ist zustandslos. Dies bedeutet, dass keine Benutzersitzungen (etwa in Form von Sessions und Cookies) existieren, sondern bei jeder Anfrage alle notwendigen Informationen wieder neu mitgeschickt werden müssen.

Durch die Zustandslosigkeit sind REST-Services sehr einfach skalierbar; da keine Sitzungen existieren, ist es im Grunde egal, wenn mehrere Anfragen eines Clients auf verschiedene Server verteilt werden.

Einheitliche Schnittstelle: Jede Ressource muss über einen einheitlichen Satz von Standardmethoden zugegriffen werden können. Beispiele für solche Methoden sind die Standard-HTTP-Methoden wie GET, POST, PUT, und mehr.

Entkopplung von Ressourcen und Repräsentation: Das bedeutet, dass verschiedene Repräsentationen einer Ressource existieren können. Ein Client kann somit etwa eine Ressource explizit beispielsweise im XML- oder JSON-Format anfordern.

Piwik 1.0 im Kundencenter (https://www.mittwald.de/blog/allgemein/google-analytics-alternative-piwik-1-0-im-kundencenter)

2. wec_discussion 2.1.1 schließt Sicherheitslücke (https://www.mittwald.de/blog/cms/typo3-cms/wec_discussions-2-1-1-schliest-sicherheitslucke)
3. WordPress Hooks: Updatesicher modifizieren (https://www.mittwald.de/blog/hosting/wordpress-updatesicher-modifizieren-mit-hooks)
4. Mitt-Links: t3n Web Award, TYPO3.org, Aufräumaktion im WordPress Plug-in-Verzeichnis und britisches PHP (https://www.mittwald.de/blog/cms/wordpress/mitt-links-t3n-web-award-typo3-org-aufraumaktion-im-wordpress-plug-in-verzeichnis-und-britisches-php)

REST und HTTP

Zur Umsetzung von solchen REST-Services wird in aller Regel das **HTTP-Protokoll** verwendet. Gründe dafür gibt es viele: Einerseits ist das Protokoll etabliert (schließlich basiert ja das gesamte WWW darauf), dennoch vergleichsweise einfach aufgebaut (der genaue Aufbau des Protokolls ist in **RFC 2616 (http://www.w3.org/Protocols/rfc2616/rfc2616.html)** definiert) und zu guter Letzt auch in so gut wie jeder Firewall offen.

Der HTTP-Standard definiert einen ganzen Satz an Operationen, mit denen auf Ressourcen zugegriffen werden kann. Die Methoden GET und POST werden auch von jedem Browser verwendet, wenn Internetseiten aufgerufen, bzw. Formulare abgeschickt werden. Daneben gibt es aber auch noch einen ganzen Satz weiterer Operationen, die von den meisten Browsern nicht genutzt werden:

- **GET** dient dazu, **lesend** auf Ressourcen zuzugreifen. **Per Definition (http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html#sec9.1.1)** darf eine GET-Anfrage nicht dazu führen, dass Daten auf dem Server verändert werden.
- Mit einem **POST**-Request können **neue Ressourcen erstellt** werden, deren URI noch nicht bekannt ist. Per POST-Request an http://ws.mydomain.tld/customers könnte also beispielsweise ein neuer Kunde mit automatisch vergebener Kundennummer (und URI) erstellt werden.
- Ein **PUT**-Request wird verwendet, um **Ressourcen zu erstellen oder zu bearbeiten**, deren URI bereits bekannt ist. Ein PUT-Request an http://ws.mydomain.tld/customers/123456 sollte demnach also einen Kunden mit der Kundennummer 123456 anlegen, falls er nicht existiert, oder ansonsten bearbeiten.
- Mit einem **DELETE**-Request können **Ressourcen gelöscht** werden.

Ein kleines Beispiel

Stellen wir uns einen fiktiven Webservice vor, der Zugriff auf eine Produkt-Datenbank bietet. Der Einstiegspunkt für den Webservice soll die URI **http://ws.mydomain.tld/products** sein. Ein GET-Request an diese URI sollte demnach also eine **Liste aller Produkte** zurückliefern:

Request	Response

`GET /products HTTP/1.0`
`Accept: application/json`
	`HTTP/1.0 200 OK`
	`Content-Type: application/json`
	`Content-Length: 1234`
	
	`[ { uri: "http://ws.mydomain.tld/products/1000",`
	`name: "Gartenstuhl",`
	`price: 24.99 },`
	`{ uri: "http://ws.mydomain.tld/products/1001",`
	`name: "Sonnenschirm",`
	`price: 49.99 } ]`

Im obigen Beispiel wird zudem noch deutlich, dass der Client über den

```
Accept
```

-Header mitteilen kann, in welchem Format er gerne eine Antwort erhalten würde (der Server muss diesen Header dann natürlich entsprechend auswerten und das angeforderte Format auch unterstützen). Alternativ könnte man zum Beispiel auch einen

```
Accept: text/xml
```

-Header mitschicken, und der Server würde dieselbe Produktliste als XML zurückschicken.

Durch einen POST-Request an dieselbe URL könnte dann ein neuer Artikel erstellt werden:

Request	Response

`POST /products HTTP/1.0`
`Content-Type: application/json`
`Content-Length: 38`
 	
`{ name: "Sandkasten",`
`price: 89.99 }`
	`HTTP/1.0 201 Created`
	`Location: http://ws.mydomain.tld/products/1002`

Interessant ist hier vor allem, in welchem Format die Daten zum Server geschickt werden. Schickt man ein Formular im Browser ab, so kodiert dieser die gesendeten Daten in URL-Codierung (der Sandkasten-Artikel aus dem letzten Beispiel sähe dann beispielsweise so aus:

```
&name=Sandkasten&price=89.99
```

). In einer REST-Architektur wäre dies genauso möglich, allerdings müsste der Client dann auch einen

```
Content-Type: application/x-www-form-urlencoded
```

-Header mitschicken.

Außerdem auffällig ist hier, dass der Server in diesem Fall nicht mit dem üblichen

```
200 OK
```

-Status antwortet, sondern mit

```
201 Created
```

. Außerdem enthält die Antwort einen Header, der die URI der neu erstellten Ressource enthält.

Weitere Funktionen von HTTP

Dadurch, dass für einen Webservice einfaches HTTP verwendet wird, ergeben sich noch reichlich weitere interessante Möglichkeiten. Da HTTP beispielsweise einen ganzen Satz an Caching-Headern (http://www.w3.org/Protocols/rfc2616/rfc2616-sec13.html#sec13) definiert, kann ein REST-Webservice beispielsweise ganz genau vorschreiben, unter welchen Umständen und für welchen Zeitraum ein Client die Antworten zwischenspeichern kann. Die Cache-Control-Header