REST – Versprechen und Wirklichkeit

Thomas Bayer, predic8

REST verspricht eine einfache und leistungsfähige Kommunikation zwischen Anwendungen. Es kommt bei zahlreichen öffentlichen und zunehmend organisationsinternen Schnittstellen zum Einsatz. Ein Grund für die weite Verbreitung sind die geringen Hürden für die Nutzung eines REST-API. Für das Testen einfacher Aufrufe genügt ein Browser und Clients lassen sich in jeder Programmiersprache mit wenigen Zeilen Code realisieren; ein Compiler oder Build-Werkzeug ist nicht notwendig. Auf den zweiten Blick hat REST Nachteile für nicht öffentliche Schnittstellen. In der Projekt-Wirklichkeit verursacht es Mehraufwände, wirft Fragen auf und bringt Komplexität mit sich. Dieser Artikel macht auf die Nachteile von REST aufmerksam und ermutigt, Alternativen wie GraphQL oder JSON RPC auszuprobieren.

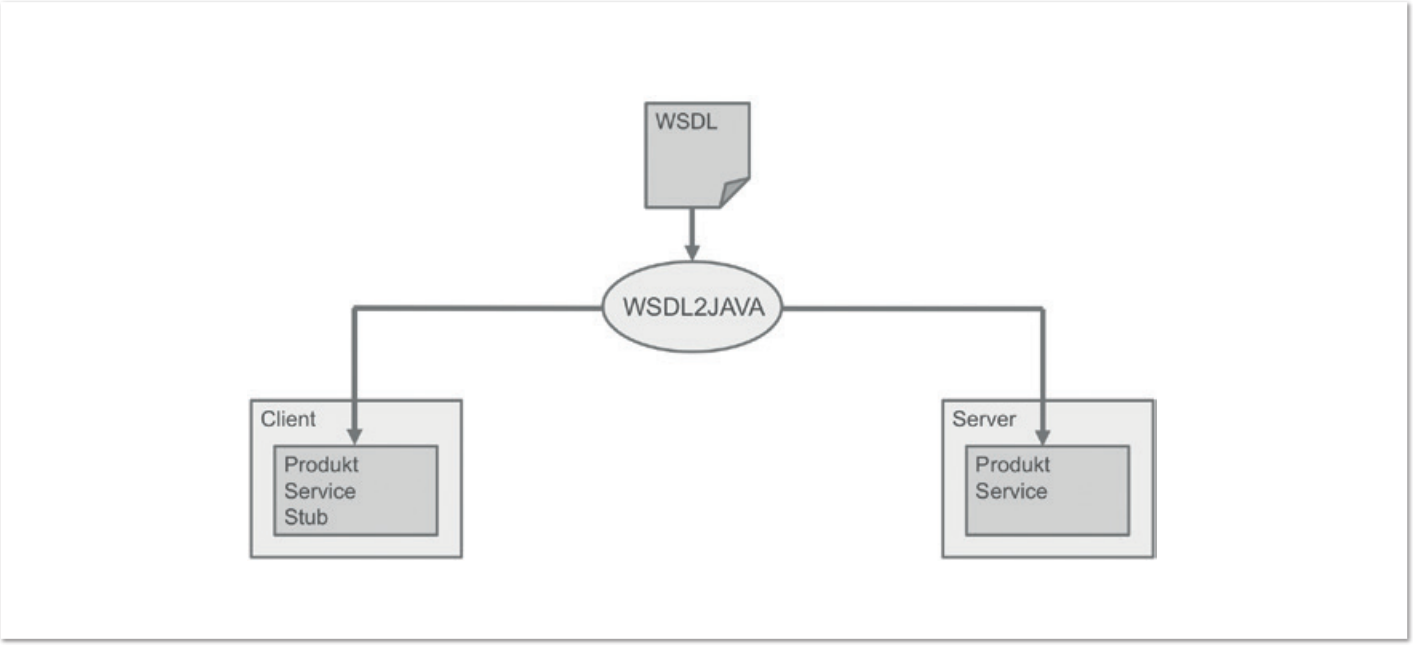

Abbildung 1: Ein Generator erzeugt Code für Client und Server

REST ist sehr technisch, was anhand eines Vergleichs mit Remote Procedure Calls ersichtlich ist. Als Beispiel für den Vergleich dienen die SOAP-basierten Web-Services. SOAP ist veraltet und die „WS-*" -Standards sind ein Paradebeispiel für unnötig komplexe Abläufe. SOAP soll hier nur als Beispiel dienen, da es von den RPC-Technologien derzeit noch am bekanntesten ist; man könnte für den Vergleich genauso gut Google RPC verwenden.

Zentral bei vielen RPC-Technologien ist eine Schnittstellen-Beschreibung. Im Beispiel in *Abbildung 1* ist dies ein WSDL-Dokument, aus dem ein Generator den Code für Client und Server erzeugt. Die für den Client erzeugte Bibliothek kümmert sich um alles Technische wie die Serialisierung zwischen Java und XML sowie die Kommunikation über HTTP.

Funktionen des Servers können mithilfe der Client-Bibliothek einfach aufgerufen werden. Der Code des Web-Service-Aufrufs in *Abbildung 2* unterscheidet sich nicht von einem lokalen Aufruf einer Funktion.

Das Erzeugen des Codes sowie das Kompilieren und Einbinden sind meist mithilfe eines Build-Werkzeugs automatisiert. Das Aufsetzen des Builds beispielsweise mit Maven ist der komplizierteste Arbeitsschritt bei RPC. Das eigentliche Programmieren hingegen ist einfach. Der Entwickler benötigt für die Arbeit mit SOAP weder XML- noch HTTP-Kenntnisse.

Die Tatsache, dass die Kommunikation über das Netz erfolgt, wird bei RPC in der Client-Bibliothek „weggekapselt"; so werden beispielsweise Netzwerk- und Anwendungsfehler in Exceptions der jeweiligen Programmiersprache überführt. Im Gegensatz zu RPC werden bei REST die Eigenschaften von HTTP ganz bewusst für die Entwicklung auf der Anwendungsebene genutzt:

- Die Adresse eines Objekts ist gleichzeitig seine ID (URI)
- Über HTTP-Header werden die Formate der Payload ausgehandelt
- Der Client interpretiert HTTP-Status-Codes und fängt keine Exceptions ab

Listing 1 zeigt typischen REST-Client-Code. Im Gegensatz zum RPC-Einzeiler ist der Aufruf aufwendiger und technischer. Der Client-Entwickler benötigt selbst bei einfacheren Client-Bibliotheken wie Unirest folgende Kenntnisse:

- Das HTTP-Protokoll
- Das Format der Repräsentation wie JSON
- Die REST-Prinzipien

REST ist die Technologie der Wahl für einfache öffentliche Schnittstellen. Der Client-Entwickler muss weder Compiler noch Code-Generator verwenden. Für komplexere Schnittstellen innerhalb eines Unternehmens spielt der Build-Aufwand einer RPC-Technologie eine untergeordnete Rolle. Dafür kann diese eine einfachere Um-

```
GetMethod method = new GetMethod("http://api.predic8.de/shop/products/65");

int sc = client.executeMethod(method);

if (sc != HttpStatus.SC_OK) {
    …
}

byte[] body = method.getResponseBody();
```

Listing 1: Client mit HTTP-Bibliothek

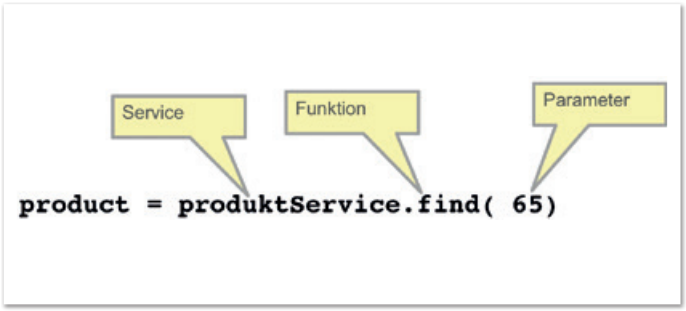

Abbildung 2: RPC-Aufruf am Beispiel von SOAP

setzung von Client und Server besonders bei Services mit vielen Parametern oder komplexen Datenstrukturen ermöglichen. Wird REST verwendet, muss jeder Anwendungsentwickler und Business-Analyst zusätzlich HTTP- und REST-Experte sein.

Fehlende Standards

Für REST gibt es keinen Standard. Wer wissen möchte, wie ein API zu gestalten ist, der schaut in die Dissertation von Fielding, in die HTTP-Spezifikation und auf Stack-Overflow nach. Diese Quellen werfen oft mehr Fragen auf, als sie beantworten. Daher gibt es unzählige API-Style-Guides von Adidas oder dem Weißen Haus, die diese Lücken füllen.

Roy Fielding beschreibt in seiner Arbeit „Architectural Styles and the Design of Network-based Software Architectures" aus dem Jahr 2000 in Kapitel 5 einen Architektur-Stil und nennt diesen „Representational State Transfer". Konkrete Hinweise darauf, wie ein REST-API zu gestalten ist, fehlen dort. Fieldings Arbeit ist nicht als Standard oder Handbuch geeignet, vielmehr als Hintergrund für ein Anwendungsprotokoll auf Basis des HTTP-Standards.

APIs sollten den HTTP-Standard einhalten. Zum Beispiel beschreibt die HTTP-Spezifikation, dass ein Server auf eine POST-Anfrage mit dem Status-Code „201 Created" antworten muss, falls eine neue Ressource angelegt wurde. HTTP beschreibt die Kommunikation zwischen Browser, Proxy und Web Server. Die Besonderheiten einer Kommunikation zwischen einem Anwendungsclient und einem Web-API sind dort nicht beschrieben. Verfolgt man auf Stack-Overflow Diskussionen zum REST-Design, kommen Fragen wie:

- Soll POST oder PUT für das Anlegen von neuen Ressourcen verwendet werden?
- PUT oder PATCH für das Ändern von Ressourcen?
- Darf ein GET-Request einen Body enthalten?
- Welchen Status-Code verwendet man für …

Es wird deutlich, dass es bei REST viel Spielraum für Interpretationen gibt. Häufig werden Fielding oder die HTTP-Spezifikation zitiert und die Bedeutungen ausgelegt:

- Laut Fielding …
- Ich verstehe die RFC 2762 anders …
- Dann ignorierst du die Empfehlung in HTTP 1/1 Sektion 4.3 …

Abbildung 3 zeigt eine typische Diskussion auf Stack-Overflow, die mehr als eine Million Aufrufe erzielen kann. Die Auslegung der HTTP-Standards und das Philosophieren über REST machen bisweilen Spaß. Aber bringt das ein Projekt wirklich weiter? Die Zeit zum Finden einer REST-konformen Lösung könnte zielführender für die Arbeit am Problem des Kunden eingesetzt werden.

Für REST-Puristen relativiert der Autor seine Aussage: Verwendet der Client Hypermedia und stellen die Repräsentationen ausschließlich den Vertrag zwischen Client und Server dar, so genügt Fieldings Arbeit in Verbindung mit der HTTP-Spezifikation. Würden REST-Prinzipien besser umgesetzt, gäbe es diese Kritik nicht. Bisher wurde so argumentiert: REST ist perfekt, es wird nur nicht richtig umgesetzt. Man kann auch anders argumentieren: REST ist so abgehoben und komplex, dass selbst 17 Jahre nach Fieldings Arbeit der Anteil an REST-konformen APIs verschwindend gering ist.

Das API-Design ist kompliziert

Ein API zu entwerfen, ist um einiges schwieriger, als eines aufzurufen. Beim Entwurf einer Schnittstelle sind zahlreiche Entscheidungen zu treffen: Werden Daten mit POST, PUT oder PATCH geändert? Kommt an das Ende von Container-Ressourcen ein Schrägstrich oder nicht, also „/produkte/" oder „/produkte"? Wie werden Relationen abgebildet? Nehmen wir an, ein Hersteller kann mehrere Produkte liefern. Wie wird dann eine Produkt-Ressource erzeugt? Etwa mit einem Post an die Container-Ressource *(siehe Listing 2)*? Oder mit einem POST an die Liste der Produkte des Herstellers durch „POST /hersteller/32/produkte/"?

Viel Spielraum gibt es auch bei der Abbildung von Relationen. Als Referenzen auf andere Geschäftsobjekte werden in den Repräsen-

Roy Fielding's comment about including a body with a GET request.

Yes. In other words, any HTTP request message is allowed to contain a message body, and thus must parse messages with that in mind. Server semantics for GET, however, are restricted such that a body, if any, has no semantic meaning to the request. The requirements on parsing are separate from the requirements on method semantics.

So, yes, you can send a body with GET, and no, it is never useful to do so.

This is part of the layered design of HTTP/1.1 that will become clear again once the spec is partitioned (work in progress).

....Roy

Yes, you can send a request body with GET but it should not have any meaning. If you give it meaning by parsing it on the server and *changing your response based on its contents*, then you are ignoring this recommendation in the HTTP/1.1 spec, section 4.3:

[...] if the request method does not include defined semantics for an entity-body, then the message-body SHOULD be ignored when handling the request.

And the description of the GET method in the HTTP/1.1 spec, section 9.3:

The GET method means retrieve whatever information ([...]) is identified by the Request-URI.

which states that the request-body is not part of the identification of the resource in a GET request, only the request URI.

share improve this answer

edited Apr 18 '16 at 18:18 Jarl 2,443 • 3 • 16 • 29

answered Jun 11 '09 at 20:27 Paul Morgan 16.5k • 3 • 18 • 26

Abbildung 3: Beitrag auf Stack-Overflow zu „HTTP GET mit Request Body"

```
POST /produkte/

{
  "hersteller": 342
  ...
}
```

Listing 2

```
{
  “vendor“: 342
  ...
}
```

Listing 3

```
{
  “vendor_url“: “https://api.predic8.de/shop/vendors/32“
}
```

Listing 4

```
PATCH /bestellung/74

{
  “status“: “storniert“
}
```

Listing 5

```
/rechnungen/
/rechnungen/{rid}
/positionen/
/positionen/{pid}
```

Listing 6

```
/rechnungen/
/rechnungen/{rid}
/rechnungen/{rid}/positionen/
/rechnungen/{rid}/positionen/{pid}
```

Listing 7

tationen oft Schlüssel verwendet *(siehe Listing 3)*. REST basiert auf Hypermedia. Anstatt eines Primärschlüssels in der Repräsentation sollte eine URI verwendet werden *(siehe Listing 4)*.

Bei der Verwendung von Verweisen stellen sich weitere Fragen:

- Sollten relative oder absolute URIs verwendet werden?
- Soll man ein eigenes Format für die Repräsentation einsetzen oder einen Standard wie die Hypertext Application Language?

In diesem Absatz wurden nur einige Fragestellungen exemplarisch aufgezählt. Je tiefer man sich mit dem API-Design beschäftigt, desto mehr tauchen davon auf.

REST begünstigt das CRUD-Antipattern

Die Eigenschaften der REST-Architektur führen zwangsläufig zu datengetriebenen Schnittstellen. Die wenigen HTTP-Methoden müssen für die Abbildung aller Funktionalitäten einer Schnittstelle ausreichen. Meist werden nur die CRUD-Methoden POST, GET, PUT und DELETE verwendet, die den Datenbankoperationen CREATE, READ, UPDATE und DELETE entsprechen. Die Kunst besteht darin, aussagekräftige Hauptwörter, also URIs, zu finden, auf die die wenigen HTTP-Methoden angewendet werden. Dazu ein Beispiel: Der Funktionsaufruf einer Nicht-REST-Schnittstelle dient zum Stornieren von Bestellungen: „storniereBestellung(74)“. Eine Möglichkeit, diesen Funktionsaufruf mit REST abzubilden, wäre ein DELETE-Aufruf an die zu stornierende Ressource mit „DELETE /bestellung/74“.

Für einfache Anwendungen ist das eine praktikable Lösung. Bei einem Storno wird einfach die Ressource gelöscht. Was aber, wenn die Daten später noch benötigt werden? Anstatt eine Bestellung zu löschen, könnte alternativ ein Storno angelegt werden: „PUT /bestellung/74/stornos“. Eine dritte Variante wäre das Überschreiben

Abbildung 4: Verlinkte Repräsentationen

der Bestellung mit einer neuen Repräsentation, bei der der Status aktualisiert würde *(siehe Listing 5)*.

Alle drei Varianten bilden den Aufruf „storniereBestellung()“ über eine Manipulation der Daten ab. REST-Schnittstellen bilden oft die Datenschicht ab, ohne zu abstrahieren und Geschäftslogik zu kapseln. Die Logik wandert dann in den Client und es entsteht eine Zweischichten-Architektur. Ein API ist keine Abstraktion für den Datenbank-Zugriff. Für die Manipulation von Daten gibt es SQL.

Eine weitere Variante des Antipattern ist eine flache CRUD-Schnittstelle, bei der jede Tabelle auf zwei URI-Templates abgebildet ist: eines für einzelne Objekte und eines für Listen. Das Beispiel in *Listing 6* zeigt die URIs für Rechnungen und die zugehörigen Positionen.

Ein solches Design ist „CRUD über HTTP“! Mit Unter-Ressourcen lässt sich die Fachlichkeit passender abbilden. Eine Position ist immer einer Rechnung zugeordnet. Eine Position existiert nie ohne eine Rechnung. Mit Sub-Ressourcen könnten abhängige Objekte zuordnet werden *(siehe Listing 7)*.

Eine neue Position ließe sich dann über die zugehörige Rechnung mit „POST /rechnungen/{rid}/positionen/“ erzeugen. Beim Löschen einer Rechnung sollten alle zugehörigen Positionen gelöscht werden. Das Löschen der Positionen könnte die Implementierung übernehmen. Das API übernimmt Verantwortung und kapselt die Geschäftslogik. Diese Realisierung ist besser, aber immer noch datenorientiert.

Wie das Beispiel zeigt, können Schnittstellen mit REST modelliert werden, die Abstraktionen bieten und Verhalten kapseln. Die Verwendung von Hypermedia verringert die Gefahr des CRUD-Antipattern. Dennoch neigt das REST-API-Design zu datenorientierten Schnittstellen mit der Gefahr der Verlagerung von Geschäftslogik vom Server zum Client.

Noch ein weiteres Beispiel: Wie startet man mit REST eine virtuelle Maschine oder einen Container? Ein Aufruf wie „POST /container/73/start“ verwendet ein Verb in der URI und ist somit nicht REST-konform. Wie sieht eine REST-konforme Lösung aus, die nicht datengetrieben ist?

Hypermedia wird nicht genutzt

Hypermedia ist bei REST nicht optional. *Abbildung 4* zeigt ein Beispiel mit drei verlinkten Repräsentationen. Der Aufbau der URIs spielt keine Rolle. Anstatt „/shop/vendors/672“ könnte eine URI auch „/foo“ oder „/3141“ heißen. Die Adresse für den Einstieg in das API ist die einzige URI, die dem Client bekannt sein muss. Danach sollte ein RESTful-Client die Repräsentationen interpretieren und die enthaltenen Links verfolgen. Im Client sollte keine weitere URI oder ein Template für URIs wie „/vendor/{vid}“ hinterlegt sein.

Bestimmte Fragen würden sich bei einem RESTful-Design erst gar nicht stellen; beispielsweise wäre das Design von URIs unnötig. Über Hypermedia wird viel geredet, es wird jedoch leider nur selten eingesetzt. Einige APIs verwenden Links, aber kaum ein Client nutzt diese zur Navigation zwischen Ressourcen. *Listing 8* zeigt einen REST-Client, der mit der Traverson-JavaScript-Bibliothek einen Link von einer Artikelbeschreibung zum Hersteller verfolgt.

```
const traverson = require('traverson');

traverson
  .from('https://api.predic8.de/shop/products/33')
  .json()
  .follow('vendor_url')
  .getResource((err, json) => {
     console.log(json.name)
});
```

Listing 8: Client mit HATEOAS-Unterstützung

```
{
  "products": [
        {
            "name": "Bananas",
            "product_url": "/shop/products/3"
        },
        {
            "name": "Oranges",
            "product_url": "/shop/products/10"
        },
        {
            "name": "Pineapples",
            "product_url": "/shop/products/33"
        }
    ]
}
```

Listing 9: Repräsentation mit Produktname und Link

```
ProductsApi api = new ProductsApi();
  for(ProductEntry pe : api.getShopProducts().getProd-
ucts()) {
    System.out.println(pe.getName());
  }
}
```

Listing 10

Vielleicht werden zukünftige Client-Programme mit Hypermedia-basierten Bibliotheken realisiert. Momentan wird Hypermedia im Client meist ignoriert. Möglicherweise sind Hypermedia und HATEOAS zu komplex oder deren Nutzen wird unterschätzt? Ohne Hypermedia fehlt REST ein wesentliches Konzept und die Vorteile wie lose Kopplung und State Transfer sind nicht nutzbar.

Swagger tötet Hypermedia

Mit Swagger beziehungsweise Open-API lassen sich REST-APIs beschreiben. Aus einer Swagger-Beschreibung kann über einen Code-Generator eine Client-Bibliothek erzeugt werden. Wird aus einem Swagger-Dokument eine Bibliothek erzeugt, dann folgt der Client, der diese verwendet, dem RPC-Paradigma. Ein Beispiel soll dies erläutern. Angenommen, ein GET-Aufruf gegen eine Ressource für Produkte liefert die Repräsentation in *Listing 9*.

Listing 10 zeigt einen auf einer mit Swagger erzeugten Bibliothek basierenden Java-Client, der diese Liste abruft und eine Liste mit den Produktnamen ausgibt.

Die Bibliothek schirmt den Entwickler vollkommen von HTTP ab; im Code ist nichts vom HTTP-Protokoll oder von JSON erkennbar.

```
ProductsApi api = new ProductsApi();
for(ProductEntry pe : api.getShopProducts().getProducts()) {
  Matcher matcher = Pattern.compile(".*\\/(.*)$").matcher(pe.getProductUrl());
  if(matcher.matches()) {
    int id = Integer.valueOf(matcher.group(1));
    System.out.printf("URL: %s Id: %d\n", pe.getProductUrl(), id);
    Product product = api.getShopProductsId(id);
    System.out.printf("%s : %s\n\n", product.getName(), product.getPrice());
  }
}
```

Listing 11: Parsen von URIs mit regulären Ausdrücken

Angenommen, der Client soll zusätzlich zum Produktnamen auch den Preis anzeigen. Um an den Preis zu gelangen, sind die „product_url"-Verweise im *Listing 11* zu verfolgen. Der Code zeigt den erweiterten Client.

Um die Repräsentation eines einzelnen Produkts abzurufen, ist ein Schlüssel im Integer-Format mit „Product product = api.getShopProductsId(id);" an die Methode „getShopProductsId" zu übergeben. Die Repräsentation der Produktliste enthält jedoch keine Integer-Schlüssel, sondern REST-konforme URIs, die auf die Produkt-Ressourcen verweisen. Dem Client-Entwickler bleibt nichts anderes übrig, als eine URI beispielsweise mit einem regulären Ausdruck zu parsen, um die ID zu erhalten.

Über URI-Templates und Swagger wird im RPC-Stil mit einer REST-konformen Schnittstelle kommuniziert. Warum wird für das REST-Design so viel Aufwand betrieben, wenn dies auf dem Client nicht genutzt wird?

API-Beschreibungen wie Swagger sind nützlich und erleichtern die Arbeit. Schnittstellen-Beschreibungen sind allerdings nicht im Sinne von REST. Ist der Einsatz einer Schnittstellen-Beschreibungssprache gewünscht, sollte über den Einsatz einer RPC-Technologie mit Beschreibungssprache etwa über Google RPC nachgedacht werden. Wird REST richtig eingesetzt, ergeben sich diese Vorteile:

- Lose Kopplung zwischen Client und Server: URIs sind nicht Teil des Vertrags und können umbenannt werden
- Es ist keine Dokumentation notwendig
- Unterstützung für dynamische Clients
- Ohne eine Anpassung der Clients können Schnittstellen evolutionär verändert werden
- Caching kann ohne Änderungen von Client und Server nachträglich eingeführt werden

Mit URI-Design und starren URIs sind diese Vorteile nicht nutzbar. Wenn die REST-Vorteile nur mit Hypermedia zu erzielen sind, aber niemand Hypermedia nutzt, ist es dann sinnvoll, REST zu verwenden?

Fazit

Für einfache öffentliche APIs ist REST die Technologie der Wahl. Bei nicht öffentlichen Schnittstellen lohnt es sich, die Verwendung von REST zu hinterfragen und dem Hype nicht blind zu folgen. Vielleicht ist GRPC, GraphQL oder JSON RPC für bestimmte Aufgaben die passendere Technologie.

Quellen

- Roy Thomas Fielding; Architectural Styles and the Design of Network-based Software Architectures: *https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm*
- Representational state transfer (REST) and Simple Object Access Protocol (SOAP): *https://stackoverflow.com/questions/209905/representational-state-transfer-rest-and-simple-object-access-protocol-soap*
- HTTP GET with request body: *https://stackoverflow.com/questions/978061/http-get-with-request-body*
- White House Web API Standards: *https://github.com/WhiteHouse/api-standards*
- Adidas-API-Guidelines: *https://github.com/adidas-group/api-guidelines*

Thomas Bayer
bayer@predic8.de

Thomas Bayer ist Mitgründer der predic8 in Bonn und der OIO in Mannheim. Als Berater und Trainer ist er mit den Themen „Microservices" und „REST" unterwegs. In seiner Freizeit praktiziert er Yoga, fotografiert und baut Quadkopter.