Interface

Het afgelopen jaar is er veel te doen geweest over REST. Iedereen lijkt het erover te hebben, maar velen weten eigenlijk niet goed wat REST precies is. REST begint erg populair te worden als alternatief voor SOAP om webservices mee te implementeren. In Java EE 6 zal er zelfs een nieuw framework worden toegevoegd, speciaal om RESTful-services mee te ontwikkelen. In dit artikel bespreekt de auteur wat REST is, hoe dit gebruikt kan worden om webservices mee te ontwikkelen en wat de voordelen zijn ten opzichte van SOAP-webservices.

RESTful Webservices

serieus alternatief voor SOAP?

Paul Bakker
is ontwikkelaar en trainer bij Info Support B.V.
Blog: http://blogs.infosupport.com/blogs/paul_bakker
Email: paulb@infosupport.com

Achtergrond

REST is in 2000 als eerste beschreven door Roy Fielding in zijn promotiewerk over architectuurstijlen. REST is een architectuurstijl die beschrijft hoe resources geadresseerd en gebruikt kunnen worden. Opvallend aan REST is dat het heel erg dicht op het http-protocol aansluit en hier niet een laag bovenop legt, zoals met bijvoorbeeld SOAP gebeurt. Waar REST nu veel voor gebruikt wordt, is het bouwen van webservices. Webservices, gebouwd volgens de principes van REST, noemen we RESTful Webservices.

De REST-hype

Ondanks dat REST dus eigenlijk niets nieuws is, lijkt REST toch *de* nieuwe manier voor het ontwikkelen van webservices te zijn. Een opvallende ontwikkeling, aangezien SOAP toch al enkele jaren de gevestigde orde is op webservicegebied. Een belangrijke reden hiervoor is dat bedrijven zoals Google, Yahoo en Amazon hun diensten steeds meer beschikbaar stellen om geconsumeerd te worden door applicaties, naast de al bestaande webinterface. Natuurlijk zou dit kunnen via SOAP-services, maar het is nog veel eenvoudiger als de services geconsumeerd kunnen worden net zoals dat op de webinterface wordt gedaan door de browser. En dat is precies wat een RESTful-webservice is. Ook de opkomst van lichtgewicht webframeworks zoals Ruby on Rails dragen bij aan de populariteit van REST, omdat er vanuit deze frameworks goede ondersteuning is voor een RESTful manier van werken.

Services op het web

Voordat we in-depth kunnen gaan kijken naar RESTful-webservices, is het goed als we eerst iets meer kijken naar wat een webservice nu eigenlijk is. De term webservice wordt vaak direct gekoppeld aan SOAP. SOAP is één manier om webservices te ontwikkelen, maar niet de enige manier. Kort door de bocht is een webservice niet meer dan een webapplicatie die bedoeld is om gebruikt te worden door andere applicaties, die geïmplementeerd kunnen zijn met heel andere technologie. Hier is weinig anders aan dan een normale webapplicatie, behalve dat hij gebruikt wordt door applicaties in plaats van door mensen (door middel van een browser). Als je op die manier tegen een webservice aankijkt, is het eigenlijk raar dat we webservices zo anders implementeren dan 'normale' webapplicaties.

RPC-style webservices

De reden daarvoor is dat we gewend zijn om remoting, oftewel communicatie tussen twee applicaties, te doen volgens het Remote Procedure Call (RPC) principe. Denk bijvoorbeeld aan RMI en CORBA. Ditzelfde principe wordt gebruikt bij SOAP-webservices. Ondanks dat er bij SOAP-webservices XML-berichten tussen client en server worden verstuurd, bevatten deze XML-berichten eigenlijk niet meer dan een RPC-aanroep.

RESTful-webservices

De belangrijkste eigenschap van een RESTful-webservice is dat er niet gewerkt wordt volgens het RPC-paradigma. Een RESTful-webservice maakt direct gebruik van de methodes van http. Voordat we daar naar gaan kijken, hebben we nog een andere term nodig die centraal staat bij RESTful-webservices: *resources*. Alles wat je via een RESTful-webservice beschikbaar wilt maken, wordt gezien als een resource. Voorbeelden van

resources zijn bijvoorbeeld 'een lijst met producten', 'een product', 'een order' of 'een klant'. Verderop in het artikel bekijken we hoe je resources kiest en hoe deze mappen op de RPC-wereld waar we mee gewend zijn te werken.

HTTPmethodes

HTTP is een heel simpel protocol. Een client stuurt een http-request naar de server, en de server stuurt een http-response terug. Zowel een request- als een response-bericht bestaan uit een header en een body. In de request-header staat bijvoorbeeld welke URI wordt opgevraagd en in de response-header staat een statuscode. In de body wordt eigenlijk de resource-representatie geplaatst. Voor een webapplicatie is dit vaak html, maar dit kan ook bijvoorbeeld XML, plain text, een bestand, of een plaatje zijn.Zoals ongetwijfeld bekend is, heeft http verschillende methodes. De belangrijkste methodes zijn:

Methode	Beschrijving
GET	Verkrijg een resource
PUT	Insert of update een resource
DELETE	Verwijder een resource
POST	Overloaded, gedrag bepaald door de server
HEAD	Controleert het bestaan van een resource (GET zonder body)

Een RESTful-webservice gebruikt deze methodes direct om acties uit te voeren op resources. Dit in tegenstelling tot het bedenken van een eigen API bovenop een resource. Werken volgens deze methodes is het aanhouden van de *Uniform Interface*. Misschien dat het nog moeilijk voor te stellen is hoe je een echte service met slechts deze methodes kunt implementeren. Hier kijken we verderop naar.

REST-eigenschappen

Een RESTful service moet aan vier criteria voldoen.

- Uniform Interface
- Addressability
- Statelessness
- Connectedness

Uniform Interface

Het gebruik van de Uniform Interface hebben we zojuist al besproken. We maken gebruik van de methodes die http ons al biedt en bedenken niet een eigen API. Dit heeft als groot voordeel dat elke resource dezelfde interface heeft en dat het altijd duidelijk is wat elke methode doet. Belangrijk hierbij is dat het ook direct duidelijk is welke effecten elke methode heeft. Een GET-request mag bijvoorbeeld nooit het effect hebben dat een resource wordt aangepast. Nu zul je op het web misschien services gezien hebben die wel op een RESTful-service lijken maar alleen GET-methodes gebruiken. Ondanks dat deze services nog steeds prima bruikbaar kunnen zijn, zijn het geen echte RESTful-services. Services die wel de RESTful-stijl aanhouden, maar niet alle RESTful-eigenschappen hebben, worden ook wel RPC-Hybrid genoemd; een mix tussen REST en RPC.

Addressability

Zoals eerder gezegd, draait een RESTful-service om resources. Elke resource moet adresseerbaar zijn, oftewel een eigen URI hebben. Dit in tegenstelling tot een SOAP-webservice, waarbij er slechts één URI voor de hele service is. Uiteraard kunnen URI's dynamisch gemaakt worden en wijst niet elke URI naar een statische resource. Hieronder zijn een aantal voorbeelden opgenomen van resources met bijbehorende URI's.

Resource	URI
Lijst van producten	/products
Product met productId 55	/product/55
Gebruiker met gebruikersnaam paulb	/user/paulb
Recente berichten	/posts/recent
Zoeken naar producten	/products/search

Let er ook hier weer op dat we dit eigenlijk altijd doen bij het maken van een 'normale' webapplicatie. Een aantal URI's in bovenstaand voorbeeld mapt op min of meer statische resources (zoals een product), terwijl andere URI's mappen naar een dynamisch gegenereerde resource (zoals een lijst met recente berichten).

Statelessness

Een RESTful-service is altijd stateless. Hier hebben we het natuurlijk over sessie-state van een gebruiker. Elke request die gedaan wordt staat geheel los van eerdere requests. Dat betekent dat de client zelf verantwoordelijk is voor het vasthouden van state, en verantwoordelijk is om relevante state mee te sturen als onderdeel van een request. Dit is de enige eigenschap die REST en SOAP met elkaar delen, een SOAP-webservice is van nature namelijk ook stateless. Er zijn overigens wel frameworks om stateful SOAP-services mee te maken, in de meeste gevallen heb je dit echter niet nodig. De statelessness-eigenschap zorgt er voor dat de services schaalbaar zijn, en er eenvoudigweg nieuwe servers bijgeplaatst kunnen worden om meer requests af te kunnen handelen.

Connectedness

Deze laatste eigenschap van RESTful-services vormt bijna de basis van het web, namelijk het geven van relevante andere URI's bij een resource. Op een website zie je dat in de vorm van links waarop je kunt klikken. In een RESTful-

webservice kunnen er URI's worden gegeven naar andere resources die de service (of een andere service) aanbiedt. Een eenvoudig voorbeeld hierbij is een resource "lijst van producten" die voor elk product niet alleen de relevante gegevens toont, maar ook een URI geeft waar meer details van een specifiek product gevonden kan worden. Als deze eigenschap goed geïmplementeerd wordt, is er slechts een zeer beperkte servicebeschrijving nodig om een client gebruik te laten maken van een service. Dit is echter ook de eigenschap die het vaakst niet wordt geïmplementeerd. De reden hiervoor is dat de service ook prima werkt zonder deze eigenschap, zolang de service maar goed is beschreven.

Het mappen van resources

Voor veel mensen die net met RESTful-webservices beginnen, is de verschuiving van methodes naar resources het lastigst. Het vraagt om een andere kijk naar functionele problemen om daar een antwoord in de vorm van resources voor te vinden. Dit betekent echter niet dat het erg lastig is, want bijna alles is goed uit te drukken in resources. Vergeet vooral niet dat resources dynamisch gecreëerd kunnen worden, en dus zeker niet iets fysieks binnen een applicatie hoeven te zijn (zoals een rij in een database bijvoorbeeld wel is). Als voorbeeld kijken we naar een eenvoudige webwinkelwebservice. Via de service kunnen producten bekeken worden en kan er een bestelling worden gedaan. De voor de hand liggende resources zijn als volgt:

Resource	URI
Lijst van producten	/products
Product met $id	/product/$id
Bestelling met $id	/order/$id
Lijst van bestellingen	/orders
Zoeken naar producten op naam	/products/search ?name=$name

De 'order' Resource bevat bijvoorbeeld de gegevens van de klant en een lijst met bestelde producten. Omdat de resources de Uniform Interface gebruiken, spreekt het al voor zich wat de acties zijn op elke resource. Het is echter wel zo dat niet elke methode op elke resource uitgevoerd kan of mag worden. De lijst producten bijvoorbeeld wordt dynamisch gemaakt uit alle beschikbare producten. Het heeft dus weinig zin om een PUT of DELETE te doen op deze lijst. Let er op dat een DELETE hier zou betekenen dat de lijst wordt verwijderd, niet een individueel product. Op een product kunnen wel alle methodes worden uitgevoerd. Hier is het echter zo dat alleen geautoriseerde gebruikers de resource mogen wijzigen. Autorisatie kan bijvoorbeeld worden geïmplementeerd door een key mee te sturen in de http-header. Een order wordt juist wel weer door een client aangemaakt. Eigenlijk ligt de set van resources erg voor de hand. Een uitgebreide servicebeschrijving is dan ook niet nodig. Dit is dan ook één van de pluspunten van RESTful-webservices; het maakt services eenvoudig.

Representations

Waar we het nog niet over hebben gehad is in welk formaat een resource aan een client wordt gegeven; oftewel de representatie van een resource. Een voor de hand liggende keuze is XML. Dit zijn we gewend van SOAP-webservices en is eenvoudig te gebruiken en te genereren. Ook bij veel RESTful-webservices wordt vaak XML gebruikt als representatie van een resource. Niets houdt ons echter tegen om meerdere representaties van dezelfde resource aan te bieden. Voor een JavaScript/Ajax-client zou bijvoorbeeld een JSON-representatie weleens handig kunnen zijn. HTTP biedt standaard al functionaliteit om met verschillende representaties om te gaan. In de header van een http-request kan worden aangegeven welke contenttypes er geaccepteerd worden en welke contenttypes de voorkeur hebben. Dit mechanisme heet 'Content Negotiation'. We zouden een service dus zo kunnen maken dat hij standaard XML teruggeeft als representatie, maar als de client om JSON vraagt, kan dit ook. Soms kan het zelfs handig zijn om ook (X)HTML als representatie aan te bieden, zodat je ook vanuit een browser kunt bekijken wat de verschillende resources van een service zijn.

Request-parameters

Voor sommige dynamische resources is het lastig om een goede URI-mapping te bedenken. Denk bijvoorbeeld aan een resource 'zoekresultaten', die het resultaat toont van een zoek-query. Deze query moet op een of andere manier in de URI worden verwerkt. Een mogelijkheid is de volgende URI: 'products/search/name/productA/color/red'. Naast dat dit soort URI's lastig te parsen zijn aan de servicekant, zit er ook geen logische, hiërarchische opbouw meer in de URI. Is bijvoorbeeld color een eigenschap van product, of een eigenschap van naam? Dit is een situatie waarbij goed gebruik gemaakt kan worden van request-parameters. Request-parameters worden gebruikt voor scoping-informatie. Een betere URI zou in dit geval zijn: '/products/search?name=product A&color=red'.

Implementeren van een service

Eigenlijk hebben we nu de hele basis van RESTful-webservices besproken. Natuurlijk moeten we ze alleen nog wel gaan implementeren. Aangezien een RESTful-webservice eigenlijk niets anders

is dan een normale webapplicatie zonder html-voorkant, ligt het voor de hand om te kijken naar webframeworks. De eerste stap hierin is natuurlijk servlets. Een service implementeren met servlets is prima mogelijk, het betekent echter wel dat je op jezelf bent aangewezen voor zaken als Content Negotiation en het parsen van URI's.
Uiteraard willen we voorkomen dat we zelf teveel boilerplate-code moeten schrijven en daarom zou een geavanceerder framework handig zijn. Er zijn in grote lijnen twee verschillende manieren die gebruikt kunnen worden om een RESTful-webservice te implementeren. Om te beginnen zijn er frameworks die er geheel op gericht zijn om RESTful-webservices te ontwikkelen. De meest in het oog springende is hierbij JAX-RS, dat onderdeel is Java EE 6. Deze API is echter nog in ontwikkeling, maar gelukkig kunnen we ook zonder JAX-RS aan de slag.
Een heel natuurlijke manier om RESTful-webservices te ontwikkelen is dan ook om dit te doen op basis van een webframework. Bijna elk webframework kan hiervoor gebruikt worden, al zijn sommige webframeworks wel beter geschikt dan anderen. Eigenlijk heb je maar twee dingen nodig in een framework om een service te kunnen implementeren; een eenvoudige manier om URI's te mappen en een eenvoudige manier om verschillende contenttypes te ontvangen en te generen. Componentgebaseerde frameworks die een hoge abstractie van het request/response-model hebben, zijn daarom minder geschikt. Zeer geschikte voorbeelden zijn bijvoorbeeld Grails en Spring MVC, terwijl JSF een voorbeeld is van een minder geschikt framework. JSF heeft geen eenvoudige manier om logische URI's te mappen, en het componentgebaseerde en event-driven model van JSF past niet goed op het eenvoudig genereren van bijvoorbeeld XML.

Voorbeeld Grails

Om een indruk te geven van hoe een RESTful-service eenvoudig kan worden geïmplementeerd met behulp van een webframework, volgt hier een kort voorbeeld met Grails. Het voorbeeld bestaat uit twee files, een standaard configuratiefile van Grails waarin url's aan controllers worden gekoppeld en een voorbeeld van een controller.

```
class UrlMappings {
 static mappings = {
   "/posts"(controller:"post", action:"list")

   "/post/$title"(controller:"post",
          action:"showPost")

   "/posts/recent"(controller:"post",
          action:"recent")
 }
}
```

Codevoorbeeld 1. URLMappings.groovy

```
class PostController {
  def showPost = {
    Post p = Post.findByTitle(params.title)
    render(contentType:"text/xml") {

       post(title:p.title, text:p.text,
           data:p.date)
      }
    }
}
```

Codevoorbeeld 2. PostController.groovy

JAX-RS

In Java EE 6 wordt een framework opgenomen voor het eenvoudig implementeren van RESTful-webservices, genaamd JAX-RS. Met JAX-RS kun je een POJO voorzien van annotaties en hiermee aangeven welke methode bijvoorbeeld een bepaalde request kan afhandelen. De specificatie is nog in ontwikkeling, maar er wordt al hard gewerkt aan een referentie-implementatie: Jersey. Bovendien heeft ook JBoss al een implementatie beschikbaar. Deze implementaties zijn nog niet aan te raden in een productieomgeving, aangezien de specificaties zelfs nog niet definitief zijn, maar geven al wel een goed beeld van wat we met Java EE 6 kunnen verwachten. Dat beeld is erg positief. Met slechts enkele annotaties configureer je een RESTful-service met alle benodigde eigenschappen. Daarnaast werk je natuurlijk gewoon in een Java EE-omgeving en kun je dus eenvoudig gebruik maken van bijvoorbeeld JPA. Hiermee lijkt JAX-RS een zeer goede keuze te zijn zodra Java EE 6 beschikbaar is.

Voorbeeld JAX-RS

Met JAX-RS worden RESTful-webservices geïmplementeerd in POJOs, geannoteerd met JAX-RS-annotaties. De belangrijkste annotaties zijn hieronder opgenomen.

Annotatie	Beschrijving
@Path	De URI van een resource, plus eventuele parameters
@HttpMethod	De HTTP methode die een methode afhandelt
@ProduceMime	Het formaat dat wordt teruggegeven door de method, bruikbaar voor content negotiation
@ConsumeMime	Als boven, maar dan voor inkomende berichten
@UriParam	Verwacht een variabele in een URI

Je kunt op deze manier bijvoorbeeld een POJO maken die verantwoordelijk is voor de resource 'Product'.

```
@Path("/products")
public class ProductResource {
   @GET
```

```
    @ProduceMime("application/xml")
    public String getProducts() {
        //Return list of Products as XML
    }

    @Path("{id}")
    @GET
    @ProduceMime("application/JSON")
    public String getProduct(@PathParam("id")
        //Return a single product as JSON
    }
    @PUT
    @ConsumeMime("application/xml")
    public void saveProduct(String body) {
        //Save or update the product
    }
}
```

Codevoorbeeld 3. Een eenvoudige JAX-RS Resource

Zoals te zien is in dit voorbeeld, is het bijzonder eenvoudig om met JAX-RS een service te implementeren. Uiteraard bevat JAX-RS nog veel meer mogelijkheden dan in het voorbeeld wordt weergegeven. Zo zijn er bijvoorbeeld Entity Providers die resources naar verschillende representaties kunnen converteren.

Implementeren van een client

Uiteraard heeft een webservice weinig nut als erg geen clients zijn. Het implementeren van een client is eenvoudig, maar wordt ook wel als lastiger gezien dan een SOAP-client. Bij een SOAP-client genereer je meestal Stub-code vanuit de WSDL. Aangezien je bij REST geen WSDL of vergelijkbare interface hebt, valt er ook weinig uit te genereren. Dit betekent dat we zelf aan de slag moeten. Het aanroepen van een service bestaat eigenlijk uit twee stappen:

- Het doen van een eenvoudige http-request
- Het consumeren van de representatie

Een http-request uitvoeren kan eenvoudig met vrijwel elke taal. In Java is hier bijvoorbeeld de Commons HTTPClient-library handig. De tweede stap is het verwerken van de representatie die de service teruggeeft. Dit kan bijvoorbeeld XML of JSON zijn. Dit kan verwerkt worden zoals elk ander XML-document verwerkt kan worden. De eenvoud van het concept is zowel een sterk als zwak punt van REST. Omdat het concept en de technologie eenvoudig zijn, kan er vanuit elk type applicatie en vanuit elke taal van de service gebruik worden gemaakt zonder dat er een taalspecifieke library noodzakelijk is. Dit is vooral een voordeel voor grote, publieke webservices die vele verschillende clients hebben. Het betekent echter ook dat er meer code nodig is aan de client-kant. Natuurlijk is dit wel op te lossen door zelf een client-library te schrijven en dit bij de service te leveren, zodat clients deze library simpelweg kunnen aanroepen. Amazon doet dit bijvoorbeeld voor hun S3-service.

RESTful-webservices en Ajax

Vaak worden RESTful-webservices en Ajax samen genoemd. De reden hiervoor is dat het bijzonder eenvoudig is om een Ajax-client bovenop een RESTful-webservice te ontwikkelen. Met Ajax is het erg eenvoudig om requests te doen naar een RESTful-webservice, en het JSON- of XML-resultaat te parsen. Voor veel eenvoudige webapplicaties geldt zelfs dat er geen webframework nodig is om pagina's te genereren, maar dat er gekozen kan worden voor een volledig op Ajax gebaseerde oplossing. Via Ajax-aanroepen kunnen de data, die normaal gesproken door een controller worden klaargezet, direct van een RESTful-webservice worden opgehaald. Hieronder is een voorbeeld opgenomen van een Ajax-client die tegen de eerder besproken product RESTful-webservice aanspreekt. In dit voorbeeld is Prototype gebruikt als JavaScript/Ajax-framework.

```
var header = new Object();
header.Accept = "application/json";
new Ajax.Request('products',
   {
     method:'get',
     requestHeaders:header,
     onSuccess: function(transport) {
     var response = transport.responseText;
     var jsonResult = response.evalJSON();
     for(var i = 0; i < jsonResult.length; i++) {
         name = jsonResult[i].name
         //Write to the DOM
     }
   }
```

Codevoorbeeld 4. Prototype voorbeeld van een GET-request

Bij het implementeren van een Ajax-client moet je natuurlijk wel rekening houden met het feit dat de browser alleen requests toestaat binnen het domein waar de pagina is geladen.

Webservices zonder WSDL

Een SOAP-webservice is altijd beschreven in een WSDL. Dit is de interfacebeschrijving van de service en bevat onder andere welke methodes er zijn, welke data heen en weer gestuurd kunnen worden en waar de service zich bevindt. Dit klinkt als een voordeel, want op deze manier kunnen client en server op elkaar aangesloten worden door middel van deze interface. Een WSDL is echter dusdanig complex dat het overzicht snel verloren is en zijn waarde verliest als servicebeschrijving. De client-code wordt echter wel op basis van de WSDL gegenereerd en voldoet hierdoor altijd aan de interface.
Bij een RESTful-webservice heb je geen WSDL, of vergelijkbare servicebeschrijving. Dit is ook niet direct nodig, want door het gebruik van de Uniform Interface is het in elk geval al duidelijk welke methodes er beschikbaar zijn. Wat echter niet vast gespecificeerd is, is welke data er precies

Een RESTful-webservice maakt direct gebruik van de methodes van http

nodig zijn om bijvoorbeeld een nieuwe resource te creëren. Hoe weet een client bijvoorbeeld welke data een product bevat? In de praktijk is dit eenvoudig op te lossen door bijvoorbeeld te kijken naar het resultaat van een GET-request, dit aan te passen en te gebruiken voor een PUT-request. Ook kunnen bijvoorbeeld XML-schema's prima gebruikt worden in combinatie met REST. Uiteraard moet er wel een functionele beschrijving zijn van de service, waarin beschreven is welke resources er zijn onder welke URIs. Dit is echter niet anders dan bij een SOAP-service. Elke SOAP-service hoort namelijk ook een functionele beschrijving te hebben. Een WSDL is dit niet, dat is een technische interface.

HTTPvoordelen bij REST

Naast het feit dat RESTful-webservices eenvoudiger zijn dan webservices op basis van SOAP, kan er ook beter gebruik worden gemaakt van de voordelen die http standaard biedt. HTTP biedt ondersteuning voor zaken zoals caching en compressie. Een SOAP-webservice kan hier niet optimaal gebruik van maken, omdat http alleen als transportmechanisme wordt gebruikt en het gebruik van slechts één URI caching bijvoorbeeld onmogelijk maakt. Omdat RESTful-webservices wel volledig gebruik maken van de mogelijkheden van http, kunnen ook deze voordelen zonder enig probleem gebruikt worden. Het valt buiten het bereik van dit artikel om uitgebreid te kijken naar hoe dit soort zaken geconfigureerd kunnen worden, maar het is zeker iets waar naar gekeken kan worden om bijvoorbeeld betere performance voor een service te verkrijgen.

Serieus alternatief

De andere aanpak van REST om webservices te implementeren biedt in veel gevallen voordelen ten opzichte van een SOAP-webservice. Vooral conceptueel passen RESTful-webservices beter bij het web. Dit wil niet zeggen dat SOAP nooit meer gebruikt zou mogen worden. In sommige gevallen past een RPC-aanpak wel beter binnen een architectuur, en sommige services hebben de aanvullende infrastructuur van de WS-*-stack nodig. In elk geval is duidelijk dat RESTful-webservices een serieus alternatief is geworden voor SOAP en ze met steeds meer framework-ondersteuning niet meer weg te denken zijn uit de wereld van webservices.

Referenties

Dissertation Roy T. Fielding: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
JSR 311 JAX-RS: https://jsr311.dev.java.net/
Jersey: https://jersey.dev.java.net/