Webservices kunnen ook zonder SOAP

REST/POX services maken met WCF

Michiel van Otegem

Wie denkt aan webservices denkt al snel aan SOAP. SOAP is het basisprotocol voor veel webservices en voor Windows Communication Foundation (WCF) de standaard voor de meeste bindings. Er zijn echter ook situaties waarin SOAP niet werkt of alleen maar overhead veroorzaakt. Gelukkig werkt WCF in .NET 3.5 ook goed zonder SOAP en kun je met WCF REST Starter Kit geavanceerde scenario's implementeren.

De keuze voor SOAP als standaardprotocol voor webservices is op zich goed. Een standaard maakt webservices communicatie wel zo eenvoudig. SOAP en de standaarden die daarop voortbouwen, zoals WS-Addressing, WS-Security en WS-ReliableMessaging, leggen de fundering voor flexibele communicatie, beveiliging, betrouwbaarheid, transacties enzovoorts. SOAP biedt echter niet altijd de (juiste) oplossing. Hier zijn twee belangrijke oorzaken voor. Ten eerste is SOAP gestoeld op het principe van een Remote Procedure Call (RPC). Een aspect hiervan is dat iedere request in principe uniek is en de de response daarop ook. Dit betekent bijvoorbeeld dat de response eigenlijk niet in een cache opgeslagen kan worden en iedere keer opnieuw opgebouwd moet worden. Verder staat in een SOAP Envelope allerlei informatie om de RPC call te kunnen maken, aangezien SOAP onafhankelijk is van het communicatieprotocol en dus niet kan rekenen op bijvoorbeeld HTTP headers om die informatie te versturen. Je hebt dus te maken met enige overhead. Daarnaast moeten beide kanten van de lijn met deze informatie overweg kunnen, wat me bij het tweede punt brengt: niet alle systemen ondersteunen SOAP en sommige systemen zullen dat ook nooit doen. Met SOAP werken betekent namelijk niet alleen het omzetten van data naar XML. Wil je met SOAP werken, dan moet je op z'n minst het basisprotocol volledig implementeren, inclusief foutafhandeling en dergelijke. Op een mainframe, waar CPU cycles duur zijn, ga je dit niet doen, tenzij je daar hele goede redenen voor hebt. SOAP is voor communicatie met legacy systemen daarom vaak geen goede oplossing. In dat geval wil 'gewone' XML, zonder dat er een standaard aan vast zit, nog wel eens de beste oplossing zijn. We noemen dit ook wel POX, wat staat voor Plain Old XML (simpele oude XML), omdat er geen voorgedefinieerde XML structuur is waar je aan moet voldoen.

POX is maar de helft van het verhaal

POX is slechts een afspraak over hoe data eruit ziet bij uitwisseling tussen twee systemen. Die afspraak is vaak een afspraak tussen twee partijen, want een discoverymechanisme zoals WSDL (Web Service Description Language) is er niet als je geen standaard gebruikt. Je kunt uiteraard een XML Schema uitwisselen, maar hoe weet een buitenstaander waar de specificatie te vinden is? Dat is in de SOAP wereld allemaal netjes vastgelegd, maar dat is nou net het protocol dat je laat varen.

Uitgaande van het idee dat POX puur data is, moet je op een of andere manier nog kunnen aangeven wat je met die data wilt doen. In de SOAP wereld is die informatie onderdeel van je SOAP Envelope en dat stoelt op het principe van een Remote Procedure Call (RPC). Je zou iets dergelijks kunnen doen met POX, maar dat is niet gebruikelijk. Een veel vaker gebruikte oplossing is het gebruik van REST (Representational State Transfer). In tegenstelling tot SOAP is het centrale principe van REST niet een service met functies, maar een resource (tekstbestand, afbeelding etcetera) waarop je acties kunt uitvoeren. Zo kun je een resource opvragen, aanmaken, wijzigen en verwijderen, met andere woorden je hebt CRUD operaties op een resource. REST wordt meestal toegepast met HTTP en de canonieke POST, GET, PUT en DELETE methodes zijn dan ook min of meer synoniem voor Create, Read, Update en Delete. Er is dus geen sprake van een RPC-achtig mechanisme waarmee je allerlei functies kunt aanroepen… of toch wel?

De crux bij REST is dat een resource aangewezen moet worden door een adres. Aangezien REST over het algemeen werkt op basis van HTTP is dit dus een URL. Nu kun je een URL wijzen naar een 'echte' resource, maar je kunt ook de URL in de server opvangen en aan de hand van de URL routeren naar een functie, bijvoorbeeld http://voorbeeld.nl/BerekenXplusY?x=1&y=2. Met het voorgaande voorbeeld zit de actie verwerkt in de URL. Wat daarbij wel belangrijk is, is herhaalbaarheid. Het opvragen van gegevens met GET dient in principe herhaalbaar te zijn. Een GET mag de resource strikt genomen niet aanpassen, aangezien

het met GET niet meer gedaan wordt dan het opvragen van een resource. In dit geval is de resource een stukje XML, maar dat zou net zo goed een afbeelding kunnen zijn. In feite kun je een zogenaamde RESTful service zien als een (statische) webpagina. Een groot voordeel van deze aanpak is dat het resultaat van een aanroep met GET opgeslagen kan worden in een cache. Het resultaat van een GET operatie mag alleen wijzigen als de resource gewijzigd is. Daar zit nog wel wat ruimte voor interpretatie, want moet een POST (create) of een PUT (update) precies de XML bevatten die je met een GET ophaalt of mag er verwerking plaatsvinden die een ander resultaat tot gevolg heeft. De REST puristen zullen zeggen dat er geen verwerking plaats mag vinden, maar het ironische is dat REST geen echt protocol is, dus dat je in feite vrij bent om dit wel te doen... je maakt je eigen protocol en zolang je maar goed documenteert wat er gebeurt als je een bepaalde actie uitvoert, is er niets aan de hand.

REST in WCF 3.5

In WCF 3.0 was er geen expliciete ondersteuning voor REST. Met enig kunst- en vliegwerk was het wel mogelijk, maar mooi was het niet. WCF 3.5 biedt echter goede ondersteuning voor REST. Hiervoor is ten eerste een nieuwe binding in het leven geroepen, de WebHttpBinding die niet veel meer doet dan het serialiseren van objecten naar XML. Dat doet de WsHttpBinding bijvoorbeeld ook, maar die zet er een SOAP Envelope omheen. Er zijn geen andere bindings beschikbaar waarmee je RESTful services kunt bouwen en dat impliceert dat REST (momenteel) alleen werkt via HTTP. Aangezien de operaties binnen REST in principe gekoppeld zijn aan het communicatieprotocol en REST vooral bedacht is binnen de context van een HTTP verbinding is dit niet zo gek. Deze koppeling met het HTTP protocol zie je ook terug in het Service Contract, waarin je met de nieuwe WebGet en WebInvoke attributen dient aan te geven hoe operaties aangeroepen kunnen worden met REST, zoals je kunt zien in codevoorbeeld 1. Het voordeel van het Service Contract in codevoorbeeld 1 is dat je onmiddellijk kunt zien dat het contract zal werken met REST en dat een service ook eenvoudig RESTful te maken is door deze attributen toe te voegen. Het nadeel is dat hiermee de strikte scheiding tussen contract en binding (protocol) die kenmerkend was voor WCF niet meer gevolgd wordt en daarmee de koppeling met HTTP wel redelijk hard is.

```
[ServiceContract(Namespace="http://Batavia-
Labs.AirlineSample")]
public interface IAirlineService
{
// Retourneert lijst met beschikbare
vluchten.

[OperationContract]
[WebGet(BodyStyle=WebMessageBodyStyle.
Bare,
RequestFormat=WebMessageFormat.Xml,
ResponseFormat=WebMessageFormat.Xml,
UriTemplate="GetAvailableFlights/{date}")]

List<Flight> GetAvailableFlights(String
date);

// Boekt een vlucht.
    [OperationContract]
    [WebInvoke(Method="POST",
        UriTemplate="BookFlight")]
    BookingConfirmation BookFlight(Booking
booking);
}
```

CODEVOORBEELD 1, SERVICE CONTRACT VOOR RESTFUL SERVICE.

Gegevens opvragen

Het voorbeeld in codevoorbeeld 1 laat een contract zien voor het weergeven van beschikbare vluchten en het boeken van een van die vluchten. De eerste is gericht op het opvragen van gegevens, hetgeen je in de REST wereld doet met een HTTP GET. In WCF geef je dit aan met het attribuut WebGet, waaraan je vier parameters mee kunt geven. Van deze parameters is alleen de parameter UriTemplate verplicht. Met deze parameter geef je aan welke URL achter de service URL geplaatst moet worden om de betreffende operatie aan te roepen. Omdat het een template is, kun je hierin ook wildcards en variabelen opnemen. De variabelen, die gemarkeerd worden door accolades dienen te corresponderen met de parameters van de service methode. Merk op dat de parameter op de service methode een String is, ondanks dat een datum benodigd is. De parameters kunnen alleen van het type String zijn en daarom dien je in de methode die het Service Contract implementeert zo nodig zelf moeten controleren of de waarde het juiste type heeft en de conversie uitvoeren. De URL voor de het opvragen van de vluchten van 10 oktober 2009 zou er op basis van de UriTemplate in codevoorbeeld 1 als volgt uit zien: http://mijnserver/AirlineService.svc/GetAvailableFlights/2009-10-10. Je bent echter vrij om te bepalen wat er achter de URL van de service komt, dus je kunt de parameter(s) ook als QueryString behandelen door de UriTemplate GetAvailableFlights?date={date} te gebruiken, of de parameter(s) in het pad op te nemen door de UriTemplate {date}/AvailableFlights.

De overige parameters van het attribuut WebGet zijn optioneel en bepalen het formaat van de data die verstuurd wordt. Met RequestFormat en ResponseFormat geef je aan of de data als XML (standaard) of als JSON (JavaScript Object Notation) verstuurd wordt. JSON is met name handig wanneer de services aangeroepen worden vanuit een webpagina. De parameter BodyStyle kan twee waardes hebben, namelijk Bare en Wrapped. Bare betekent dat data in feite alleen geserialiseerd wordt. Op basis van het Data Contract in codevoorbeeld 2 betekent dit dat GetAvailableFlights als resultaat de XML in codevoorbeeld 3 retourneert. Bij Wrapped krijgt de data een eigen namespace en wordt hier wat XML omheen gezet waardoor je ziet welke service methode van toepassing is. Dit kan bijvoorbeeld handig zijn als je een centraal afhandelingmechanisme hebt dat niet weet welke methode aangeroepen is. In veel gevallen is Bare echter genoeg, maar omdat WCF de (de)serialisatie voor je regelt is het meestal ook niet belangrijk welk formaat je gebruikt. Het is vooral van belang dat er goede afspraken zijn tussen de client en de server over het formaat, niet in de laatste plaats omdat je niet zoals bij SOAP gebruik kunt maken van WSDL (Web Service Description Language) om het service contract van de service te publiceren. Dit is overigens wel iets waar de WCF REST Starter Kit bij kan helpen. De WCF REST Starter Kit bestaat uit features die in WCF 4.0 terecht zullen komen, waaronder de mogelijkheid om RESTful services beter van documentatie te voorzien. Belangrijk in deze is wel om je te realiseren dat dit (vooralsnog) niet gebaseerd is op een standaard als WSDL. De WCF REST Starter Kit is beschikbaar via http://www.asp.net/downloads/starterkits/wcf-rest/.

```
[DataContract]
public class Flight
{

    [DataMember]
    public int FlightId { get; set; }

    [DataMember]
    public string FlightNumber { get; set;}

    [DataMember]
    public string Origin { get; set; }

    [DataMember]
    public string Destination { get; set;}
```

```
    [DataMember]
    public DateTime Departure { get; set;}

    [DataMember]
    public int AvailableSeats { get; set;}
}
```

CODEVOORBEELD 2, DATA CONTRACT VOOR EEN VLUCHT.

```
<ArrayOfFlight
    xmlns="http://schemas.datacontract.
org/2004/07/WCFPoxRest.Services.DataCon-
tracts"
    xmlns:i="http://www.w3.org/2001/XML-
Schema-instance">
    <Flight>
        <AvailableSeats>38</Availa-
bleSeats>
        <Departure>2009-10-07T13:40:00</
Departure>
        <Destination>IAH</Destination>
        <FlightId>2</FlightId>
        <FlightNumber>CO47</FlightNumber>
        <Origin>AMS</Origin>
    </Flight>
    <Flight>
        <AvailableSeats>29</Availa-
bleSeats>
        <Departure>2009-10-07T14:00:00</
Departure>
        <Destination>SEA</Destination>
        <FlightId>3</FlightId>
        <FlightNumber>KL94</FlightNumber>
        <Origin>AMS</Origin>
    </Flight>
</ArrayOfFlight>
```

CODEVOORBEELD 3, RESULTAAT VAN GETAVAILABLE-FLIGHTS MET WEBMESSAGEBODYSTYLE.BARE.

Gegevens wijzigen

Een service methode aanbieden die gegevens kan wijzigen is eigenlijk niet ingewikkelder dan het aanbieden van gegevens via WebGet. Sterker nog, je zou dit prima via WebGet kunnen doen, ware het niet dat een HTTP GET eigenlijk herhaalbaar zou moeten zijn. Gegevens invoegen op wijzigen doe je zoals als eerder gezegd in de regel met POST of PUT en daarvoor het je het attribuut WebInvoke. Deze heeft ten opzichte van het attribuut WebGet een extra parameter, te weten Method. Hierin geeft je aan welke HTTP method gebruikt dient te worden bij het aanroepen van de service methode. In het voorbeeld in Codevoorbeeld 1 zie je de service methode BookFlight om een vlucht te boeken. Conform de gedachtegang dat dit het aanmaken van nieuwe gegevens betreft, is aangegeven dat deze methode met POST werkt. Merk op dat de methode een parameter Booking heeft waarvoor geen overeenkomstige variabele opgenomen is in de UriTemplate. Dat is omdat de gegevens hiervan niet verstuurd worden in de URL, maar als content van de POST opdracht.

Een service aanroepen met REST

Het aanroepen van een service die aangeboden wordt via REST is bijna net zo eenvoudig als het maken van een dergelijke service. Wat je nodig hebt is het Service Contract, die je kunt delen via een assembly als zowel client als service geïmplementeerd worden in WCF 3.5. Als dit niet het geval is, kun je het Service Contract zelf maken, precies corresponderend met hoe deze zou zijn als je deze voor de service zou maken. Als de service aangeboden wordt vanuit een ander platform, is het wel mogelijk dat je het Data Contract moet aanpassen in de attributen DataContract en DataMember om de gegevens goed te kunnen deserialiseren. Gewapend met het Service Contract kun je een zogenaamde WebChannel aanmaken waarmee je de methodes die gespecificeerd zijn in het Service Contract kunt aanroepen, zoals in het voorbeeld hieronder.

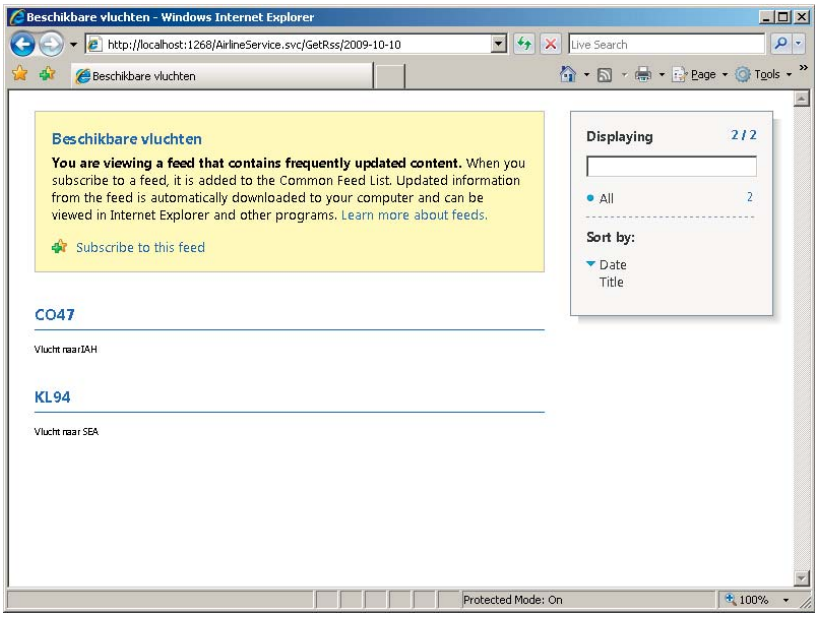
REST is zonder meer interessant en wordt al erg veel gebruikt

```
WebChannelFactory<IAirlineService> cf =
new
  WebChannelFactory<IAirlineService>(new
Uri(SERVICE_URI));
IAirlineService service = cf.CreateChan-
nel();
List<Flight> flights = service.
GetAvailableFlights(date);
```

Voor een methode die gegevens wijzigt geldt dit precies hetzelfde, dus voor jou als ontwikkelaar is het eigenlijk transparent dat je de service aanroept als een RESTful service.

RSS en ATOM

REST is zeker interessant, maar je vraagt je misschien af of het veel gebruikt wordt. Dit is zonder meer het geval, aangezien RSS en ATOM feeds waarmee de meeste blogs de nieuwste artikelen publiceren zeer goede voorbeelden van RESTful services zijn. Een aardige les die we hieruit kunnen trekken is dat RESTful niet per definitie protocolloos betekent. RSS en ATOM zijn geaccepteerde standaarden voor gegevensuitwisseling. Dit kun je in je voordeel gebruiken door gegevens die je wilt publiceren in het RSS of ATOM formaat aan te bieden. Het enige dat je dan hoeft te publiceren is welke velden uit de standaard je gebruikt om welke gegevens aan te bieden. De code om dit voor elkaar te krijgen is in WCF 3.5 een fluitje van een cent, door de ondersteuning voor RSS en ATOM feeds. Alles wat je hoeft te doen is een feed aanmaken en deze met een Rss20FeedFormatter of een Atom10FeedFormatter te retourneren uit een methode met het WebGet (met BodyStyle=Bare). Codevoorbeeld 5, waarin de methode GetAvailableFlights hergebruikt wordt voor het verkrijgen van de juiste gegevens laat zien hoe dat eruit ziet. Omdat het hier met name om het concept gaat, is het aantal feed eigenschappen dat gevuld wordt beperkt. Als je de feed opvraagt in Internet Explorer, dan resulteert dit in afbeelding 1.



AFBEELDING 1, BESCHIKBARE VLUCHTEN ALS RSS FEED.

```
<GetAvailableFlightsResponse
xmlns="http://BataviaLabs.AirlineSample">
    <GetAvailableFlightsResult
xmlns:a="http://schemas.datacontract.
org/2004/07/WCFPoxRest.Services.DataCon-
tracts"
        xmlns:i="http://www.w3.org/2001/
XMLSchema-instance">
        <a:Flight>

<a:AvailableSeats>38</a:AvailableSeats>

<a:Departure>2009-10-07T13:40:00</
a:Departure>

<a:Destination>IAH</a:Destination>
            <a:FlightId>2</a:FlightId>

<a:FlightNumber>CO47</a:FlightNumber>
            <a:Origin>AMS</a:Origin>
        </a:Flight>
        <a:Flight>

<a:AvailableSeats>29</a:AvailableSeats>

<a:Departure>2009-10-07T14:00:00</
a:Departure>

<a:Destination>SEA</a:Destination>
            <a:FlightId>3</a:FlightId>

<a:FlightNumber>KL94</a:FlightNumber>
            <a:Origin>AMS</a:Origin>
        </a:Flight>
    </GetAvailableFlightsResult>
</GetAvailableFlightsResponse>
```

CODEVOORBEELD 4, RESULTAAT VAN GETAVAILABLEFLIGHTS MET WEBMESSAGEBODYSTYLE.WRAPPED.

```
public Rss20FeedFormatter GetRss(String
date)
{

// Een feed is onafhankelijk van RSS of
ATOM.
    // Welk formaat je gebruikt geef je
later aan.
    SyndicationFeed feed = new Syndicati-
onFeed();
    feed.Title = new TextSyndicationConten
t("Beschikbare vluchten");
    feed.Description = new TextSyndication
Content("Vluchten op " + date);
    feed.Authors.Add(new
SyndicationPerson("M. van Otegem"));
    // Hier alleen feed maken. Logica voor
gegevens hergebruiken.
    List<Flight> flights =
GetAvailableFlights(date);

    List<SyndicationItem> items = new
List<SyndicationItem>();
    foreach(Flight flight in flights)
    {
        items.Add(new SyndicationItem()
        {
            Title = new
TextSyndicationContent(flight.FlightNum-
ber),
            Content = new TextSyndica-
tionContent(

String.Format("Vlucht naar {0}", flight.
Destination)),
            BaseUri = new Uri(

String.Format(SERVICE_URI + "{0}", flight.
FlightId)
        });
    }
    feed.Items = items;

    // Hier formaat bepalen en retourne-
ren.
    return new Rss20FeedFormatter(feed);
}
```

CODEVOORBEELD 5, GEGEVENS DELEN VIA EEN RSS FEED.

Michiel van Otegem, is Chief Software Architect bij BataviaLabs, schrijver en spreker. Hij is per email te bereiken via michiel@aspnl.com.