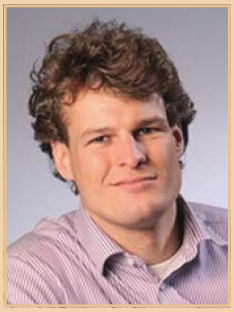
**Webapplicaties lijken steeds meer op Client-Server applicaties. Kijk naar voorbeelden als Gmail of Google-documents. In deze architectuur is de client volledig verantwoordelijk voor de user interface en de server verantwoordelijk voor data en business logica. Er is nog geen overeenstemming over de naam van deze aanpak. Een aantal gebruikte namen zijn RIA, SOFEA, en SOUI. Voorlopig hou ik het bij Client-Server 2.0, omdat dit redelijk goed duidelijk maakt wat we aan het doen zijn.**

# Client Server 2.0 met jQuery en Grails



**Andrej Koelewijn**
is als IT-architect werkzaam bij IT-eye. Hij is gespecialiseerd in JEE, Oracle en Open Source. Andrej is bereikbaar op het volgende email adres: andrej.koelewijn@it-eye.nl

De eerste generatie Client-Server applicaties was gebaseerd op proprietary ontwikkeltools en technologieën als Oracle Forms en de Oracle database of Visual Basic en MS-SQLserver. Tegenwoordig is het mogelijk en eenvoudig om client server applicaties te realiseren op basis van open standaarden als HTML, Javascript en -services.
Browsers bieden tegenwoordig een volwassen applicatiecontainer. Met open standaarden als Javascript, HTML en CSS kun je niet alleen webpagina's schrijven, maar ook complete applicaties. HTML is een krachtige en eenvoudige layout engine, skinning is eenvoudig met behulp van CSS, en Javascript is een krachtige, performante en complete programmeertaal geworden. Feitelijk een complete set tools om applicaties mee te bouwen en niet slechts webpagina's.
SOA wordt veel toegepast om herbruikbaarheid van software te vergroten. Door software in servicevorm aan te bieden is hergebruik eenvoudiger. Nieuwe software kan sneller gerealiseerd worden, bestaande software kan eenvoudiger aangepast worden.
Dit artikel illustreert hoe Client Server 2.0 applicaties eenvoudig en productief gerealiseerd kunnen worden met jQuery en Grails. jQuery gebruiken we als belangrijkste framework voor de clientkant van de applicatie. Grails gebruiken we om snel services te realiseren.

## jQuery

jQuery is een javascript library. De belangrijkste kenmerken daarvan zijn:

- eenvoudig manipuleren van DOM object, met behulp van een syntax die erg op die van CSS lijkt,
- statement chaining, met andere woorden het aan elkaar plakken van functie aanroepen, waardoor je compacte code krijgt,
- verbergen van browser verschillen. Jquery ondersteunt de belangrijkste browsers.

Het volgende voorbeeld komt van de homepage van jQuery. Deze ene regel code voegt aan alle paragrafen met de class neat een extra class ohmy toe, en vervolgens worden ze met een trage animatie getoond.

```
$("p.neat").addClass("ohmy").show("slow");
```

Op deze manier is met jQuery veel te bereiken met weinig.

### Scheiden inhoud, opmaak en gedrag

Veel ontwikkelaars zijn echter bezorgd dat je met Javascript geen onderhoudbare applicaties kunt bouwen. Een belangrijke stap om de onderhoudbaarheid te verbeteren is om inhoud, opmaak en gedrag goed te scheiden. Een HTML-bestand hoort slechts inhoud te bevatten, een CSS stylesheet slechts opmaak, en een Javascript bestand slechts gedrag.
Het volgende voorbeeld toont naast het scheiden van opmaak, inhoud en gedrag ook hoe je een single page applicatie kunt maken, waarbij dynamische gedeelten van de pagina toegevoegd worden met behulp van de jQuery load functie.
Eerst het statische gedeelte van de HTML pagina. Dit is het gedeelte dat de gebruiker niet verlaat. In de header worden de benodigde stylesheets en javascript bestanden gelinked. Verder bevat de pagina een button. De bijbehorende code voor deze bewerking is te vinden op www.javamagazine.nl/site/Hetblad/Extra.html onder codevoorbeeld 1.

Het volgende HTML bestand, *form.html*, zal dynamisch in de pagina geplaatst worden, nadat

# We laten uw IT en bedrijfsprocessen perfect bij elkaar aansluiten

Yenlo is als geen ander in staat de verschillende IT-componenten binnen een organisatie tot een perfect werkend geheel te smeden. We helpen onze klanten met advies, architectuur en softwareontwikkeling. Service gerichte architecturen ontwikkelen wij met de Oracle SOA Suite en met de SUN GlassFish Suite. We hebben de kennis, we hebben de competenties.

Yenlo

De SOA, BPEL en ESB specialist met Oracle SOA en SUN GlassFish

www.yenlo.nl

de gebruiker op de *Show Form!* button gedrukt heeft.

```
<form name="form1">
<label for="nameField">Name</label>
<input id="nameField" value="" />
<button type="button" id="saveButton">Save</button>
</form>
```

Voor de opmaak hebben we een stylesheet, *example1.css*, dat ervoor zorgt dat het dynamische formulier een grijze achtergrond krijgt.

```
#formPanel { background-color: #ddd; }
```

Tenslotte nog een Javascript document, *example1.js*, dat het gedrag achter de pagina implementeert. Het eerste gedeelte maakt gebruik van een jQuery functie, *ready*, die ervoor zorgt dat de code uitgevoerd wordt nadat de pagina is ingeladen. In de ready functie wordt de *Show Form!* button voorzien van een onClick event handler.

```
$(document).ready(function(){
$("#showFormButton").click(  function(){  showForm();
});
});
```

De tweede functie wordt aangeroepen indien de gebruiker de knop indrukt. Eerst wordt de knop uitgeschakeld, vervolgens wordt er een *div* in het document toegevoegd achter de knop, en wordt deze *div* gevuld met het document *form.html*. Zodra dit document geladen is, wordt de *div* met een animatie getoond, en wordt er een event handler aan de saveknop gehangen.

```
function showForm(){
$("#showFormButton").attr("disabled","true")
$("#showFormButton").after("<div   id='formPanel'></
div>");
$("#formPanel").hide().load("form.html",  function()
{
$("#formPanel").fadeIn("slow");
$("#saveButton").click( function(){ hideForm(); });
});
}
```

De functie hideForm zorgt ervoor dat de *div* weer verwijderd wordt, en dat de *Show Form!* button weer bruikbaar wordt.

```
function hideForm(){
$("#formPanel").remove();
$("#showFormButton").removeAttr("disabled");
}
```

Bovenstaand voorbeeld toont dat je inhoud en gedrag volledig kunt scheiden. Er staat geen regel Javascript code in de HTML-documenten. jQuery maakt het verder vrij eenvoudig om een single page applicatie te realiseren, waarbij dynamisch DOM-elementen worden toegevoegd of stukken HTML. Je kunt de HTML dus opdelen in verschillende bestanden. Het is niet nodig om alle HTML dynamisch met Javascript te genereren. Hierdoor blijft de HTML beter onderhoudbaar.

**jQuery componenten**

Het succes van jQuery heeft er voor gezorgd dat er ondertussen honderden jQuery componenten beschikbaar zijn, van formuliervalidatiefuncties, tot autocomplete componenten en hele complexe tabelcomponenten. Een complete lijst is te vinden op de jQuery plugin pagina: http://plugins.jquery.com/. De meeste jQuery componenten zijn onafhankelijk van de gebruikte technologie op de server. Dit zorgt ervoor dat er meer componenten zijn dan in server technologie specifieke componenten, zoals bijvoorbeeld JSF. JQuery componenten zijn ook een stuk eenvoudiger te realiseren dan server componenten, omdat je slechts met één implementatielaag te maken hebt: de view laag in de browser.

**jQuery accordion**

Het is vrij eenvoudig om nieuwe jQuery componenten te implementeren. Onderstaand voorbeeld toont hoe je een accordion menu kunt realiseren. We willen een component maken waarmee we een geneste unorderd list kunnen veranderen in een accordion menu. De eerste lijst bevat menusecties, daarbinnen wordt weer een unordered list gebruikt om de menuopties weer te geven, in de vorm van url links. Een semantisch correct geneste list dus.
De bijbehorende code voor deze bewerking is te vinden op www.javamagazine.nl/site/Hetblad/Extra.html onder codevoorbeeld 2.

Het accordion menucomponent, *example2-accordion.js*, bestaat uit twee onclick event handlers, één om de secties op en dicht te klappen (toggle), en één om ervoor te zorgen dat onClick events op geneste list items, niet door de parent list items gedetecteerd worden. Het component zelf wordt gedefinieerd met de expressie *$.fn.accordion*. De bijbehorende code voor deze bewerking is te vinden op www.javamagazine.nl/site/Hetblad/Extra.html onder codevoorbeeld 2a.

Nu hebben slechts één regel code nodig om van de unordered lists een accordion menu te maken:

```
$("#mainMenu").accordion();
```

**jQuery form componenten**

Er zijn een aantal componenten die van pas kunnen komen voor het initialiseren, valideren en

vereenvoudigen van formulieren.
Een nuttige is de jQuery validation plugin (http://bassistance.de/jquery-plugins/jquery-plugin-validation/). Hiermee kun je eenvoudig de door de gebruiker ingevoerde data controleren. Je kunt validaties op twee manieren specificeren, in de HTML-code en in Javascript. Enerzijds kun je de gewenste validatie toevoegen als CSS class:

```
<input type="text" id="bsnVeld" name="bsn"
class="required" />
```

Je dient dan nog wel de validation plugin op het gewenste formulier te activeren. Dit doe je als volgt:

```
$("#formPanel form").validate();
```

Invoer wordt nu automatisch gevalideerd: indien de gebruiker het veld verlaat zonder invoer, wordt er automatisch een label achter het veld geplaatst met een error melding. Deze wordt ook weer automatisch verwijderd als de gebruiker iets invoert in het veld. De toegevoegde error melding ziet er als volgt uit:

```
<label class="error" for="bsnVeld" generated="true"
  style="display: inline-block;">This field is
required.</label>
```

Validaties kunnen ook in javascript gedefinieerd worden. De eerder genoemde validatie wordt dan als volgt gedefinieerd:

```
$("#formPanel form").validate({
rules: { voornaam: "required" },
messages: { voornaam: { required: "Voornaam is een
verplicht veld" }}
});
```

Complexere validaties kunnen op een herbruikbare manier gedefinieerd worden:

```
jQuery.validator.addMethod("dateNL",function(value,
element){
return this.optional(element) || /^[0-9]{2}-[0-9]
{2}-[0-9]{4}/.test(value);
},"Voer een geldige datum in (dd-mm-yyyy)");
```

Datum velden zijn nu eenvoudig van validatie te voorzien:

```
<input id="geboortedatumField" name="geboortedatum"
value="" class="dateNL"/>
```

Deze voorbeelden tonen slechts een klein gedeelte van de mogelijkheden van de validation plugin. Een andere nuttige plugin is de *jQuery masked input* plugin (http://digitalbush.com/projects/masked-input-plugin/). Hiermee kun je eenvoudig aangeven wat het formaat van de invoer moet zijn, bijvoorbeeld voor een postcode of datum veld:

```
$("#postcodeField").mask("9999 aa");
$("#geboorteDatumField").mask("99-99-9999");
</code>
```

Meestal zul je datums via een datepicker willen invoeren. Ook hiervoor zijn meerdere plugins beschikbaar. Het volgende voorbeeld maakt gebruik van de jQuery UI datepicker plugin. Met dit component kun je een HTML-input tag met één regel Javascript code veranderen in een datepicker veld:

```
$('#geboorteDatumField').datepicker({ dateFormat:
"dd-mm-yy" });
```

## Grails

Grails is een compleet webframework gebaseerd op de ideeën van Ruby on Rails, maar geïmplementeerd op bewezen Java-frameworks. Grails maakt namelijk gebruik van Spring en Hibernate. Daarnaast is er voor de programmeertaal Groovy gekozen. Deze dynamische programmeertaal draait op de JVM, maar is wat compacter dan Java en biedt ondersteuning voor bijvoorbeeld Closures. Ook kun je eenvoudig op runtime bestaande code uitbreiden met nieuwe functionaliteit.
In dit artikel wordt slechts een klein gedeelte van de functionaliteit van Grails gebruikt. Grails blijkt namelijk één van de meest eenvoudige en productieve om REST services te realiseren.
REST wordt meestal gezien als de simpele variant van Web Services. Services die via HTTP aangeroepen worden, zondere complexe SOAP XML toestanden, waarbij slechts de relevante data in XML of JSON formaat wordt gecommuniceerd. REST is echter een compleet andere manier van het beschikbaar stellen van services.

Huidige SOA implementaties zijn meestal gebasseerd op een van de volgende concepten:

- Message Queueing – de verschillende onderdelen van een gedistribueerde applicatie communiceren met elkaar door middel van het versturen van berichten.
- Remote procedure calls – de verschillende onderdelen van een gedistribueerde applicatie communiceren met elkaar door middel van het aanroepen van functies via het netwerk.

Dat REST echt anders is, kan misschien het best verduidelijkt worden door het te zien als een datamodel. Resources zijn vergelijkbaar met entiteiten, of, beter nog, views. Deze views maken data beschikbaar, maar verbergen de daadwerkelijke implementatie. Het grote verschil tussen de twee eerder genoemde concepten en REST is dat

resources in REST adresseerbaar zijn met behulp van een unieke URL. Iedere resource heeft een URL. Belangrijk is ook dat resource met behulp van deze URL's linken naar andere resource, niet alleen binnen een organisatie, maar ook naar externe resources. URL's zijn dus feitelijk de primary keys van dit datamodel. Je kunt krijgt daarmee een gedistribueerd datamodel.

**Domein Objecten**

Grails heeft een command line tool waarmee je snel een project kunt opzetten. Je kunt hiermee ook de basis van de benodigde domein classes genereren. Daarna is het slechts een kwestie van het invullen van de benodigde attributen. Grails domein classes zijn in feite POJO's die met Hibernate gepersisteerd worden. Tabellen worden, indien gewenst, automatisch tijdens het starten van de applicatie gegenereerd. Voor dit voorbeeld voldoet de standaard tabel structuur, maar indien gewenst is deze wel volledig te configureren in de domein classe code. Ook kunnen validaties worden toegevoegd.

```
class Patient {
String voornaam
String achternaam
String straat
String huisnummer
String plaats
String postcode
GregorianCalendar geboorteDatum
}
```

Groovy zorgt zelf voor de benodigde getters en setters, en ook punt-komma's zijn niet nodig. Kort en bondig dus.

**Services**

Vervolgens hebben we een controller nodig. Deze handelt de binnenkomende http-verzoeken af. In de class UrlMappings kunnen we configureren welke controller welke url afhandeld. In het volgende voorbeeld is gespecificeerd dat de controller patient, geimplementeerd met de class PatientController, alle GET verzoeken voor de url */patient*, uitvoert.

```
class UrlMappings {
static mappings = {
"/patient"(controller:"patient"){
action = [GET:"show"]
}
}
}
```

De patient controller zelf kunnen we ook weer met Grails genereren, zodat we slechts de functie show hoeven toe te voegen. Grails voegt zelf aan domeinobjecten een aantal functies toe die je normaal in DAO objecten zou coderen. De functie list queried alle objecten. Vervolgens kan het resultaat eenvoudig in JSON geformatteerd worden mbv *render as JSON*.

```
import grails.converters.JSON

class PatientController {

def show = {
def all = Patient.list()
render all as JSON
}
}
```

Bovenstaande code is feitelijk alles wat we nodig hebben aan de serverkant om een service te implementeren waarmee alle patienten opgehaald kunnen worden.

**Overzichtstabel**

De volgende stap is om een HTML tabel te voorzien van data uit bovenstaande patient service. De bijbehorende code voor deze bewerking is te vinden op www.javamagazine.nl/site/Hetblad/Extra.html onder codevoorbeeld 3.

Onder de tabel is een button geplaatst. De bedoeling is dat hiermee de tabel gerefreshed kan worden. Eerst definiëren we een *onClick* event handler voor de button.

```
$(document).ready(function(){
$("#refreshPatientsButton").click( function(){
refreshPatientsTable();
});
});
```

De functie die vervolgens wordt aangeroepen maakt gebruik van de jQuery ajax functie. De url die we aanroepen is *patient/*, deze geeft alle bekende patienten terug in JSON formaat. Vervolgens specificeren we ook wat er moet gebeuren nadat de data succesvol is opgehaald: de functie *fillPatientTableData* wordt aangeroepen met de patient data, door jQuery al vertaald naar Javascript objecten.

```
function refreshPatientsTable(){
$.ajax({
url:"patient/",
dataType:"json",
success: function(json){ fillPatientTableData(json);
}
});
}
```

De functie *fillPatientTableData* loopt door de patienten objecten, en genereert de benodigde HTML voor in de tabel body. Vervolgens wordt de body van de tabel vervangen met de nieuwe HTML tags. De bijbehorende code voor deze bewerking is te vinden op www.javamagazine.nl/site/Hetblad/Extra.html onder codevoorbeeld 4.

Onderstaande screenshot toont het resultaat in Firefox. Onder de HTML-pagina zie je de Firebug plugin. Deze plugin biedt onder andere net-

werk data debug mogelijkheden. Zo zie je in de screenshot het resultaat van de serviceaanroep, een aantal Javascript objecten in een array, in JSON formaat.
De Firebug plugin is een onmisbaar tool bij het ontwikkelen van Javascript applicaties. Je kunt onder andere alle HTML wijzigingen live volgen, en CSS en Javascript debuggen. Daarnaast kun je ook al het netwerkverkeer debuggen, inclusief timing gegevens.

**Patient details**
Volgende stap is om ervoor te zorgen dat de gebruiker een patient kan wijzigen. Eerst zorgen we ervoor dat de gebruiker de patient kan selecteren. Dit doen we door de volgende regel toe te voegen in de functie *fillPatientTableData.* Iedere regel in de tabel bevat nu een *Edit* knop.
tbody += "<td><button value='" + data[i].id + "' type='button'>Edit</button></td>";
De onclick event handler voor deze button wordt in dezelfde functie gedefinieerd:

```
$("#patienten tbody td button").click(function(){
showEditPatientForm($(this).val());
});
```

De functie *showEditPatientForm* roept de eerder gedefinieerde service aan, maar nu specificeren we in de URL ook de id van de gezochte patient.

```
function showEditPatientForm(patientId){
$.ajax({
url:"patient/" + patientId,
dataType:"json",
success:function(json){initEditPatientForm(json);}
});
}
```

De functie *initEditPatientForm* wordt aangeroepen met de patient data van de service aanroep. Eerst wordt een div toegevoegd achter de patient tabel. Hierin wordt het HTML-bestand met het edit formulier geladen. Met behulp van de populate plugin worden de velden van het formulier geïnitialiseerd met de patient gegevens. De bijbehorende code voor deze bewerking is te vinden op www.javamagazine.nl/site/Hetblad/Extra.html onder codevoorbeeld 5.

In de controller zorgen we ervoor dat indien er een id is gespecificeerd, alleen de bijbehorende patient wordt gequeried en geretourneerd. De bijbehorende code voor deze bewerking is te vinden op www.javamagazine.nl/site/Hetblad/Extra.html onder codevoorbeeld 6.

In de url mapping voegen we toe dat in de URL optioneel een id parameter meegegeven kan worden. De bijbehorende code voor deze bewerking is te vinden op www.javamagazine.nl/site/Hetblad/Extra.html onder codevoorbeeld 7.

Tenslotte het edit formulier. Dit is een standaard, recht toe recht aan HTML formulier. De bijbehorende code voor deze bewerking is te vinden op www.javamagazine.nl/site/Hetblad/Extra.html onder codevoorbeeld 8.

In onderstaande screenshot het resultaat. In de Firebug console zie je onder andere het resultaat van de service aanroep, de patient data in JSON formaat.

**Opslaan wijzigingen**
Tenslotte nog het opslaan van de wijzigingen. De functie *serializeArray* plaatst alle form elementen in een JSON data structuur. Deze kan vervolgens in de ajax aanroep van de service meegegeven worden. Voor het wijzigen van resources wordt HTTP Post gebruikt. De bijbehorende code voor deze bewerking is te vinden op www.javamagazine.nl/site/Hetblad/Extra.html onder codevoorbeeld 9.

In de URL mappings definieeren we dat in het geval van een HTTP Post de *save* functie aangeroepen dient te worden in de controller.

```
class UrlMappings {
static mappings = {
"/patient/$id?"(controller:"patient"){
action = [GET:"show",POST:"save"]
}
}
}
```

De save functie bepaalt eerst of het om een nieuwe of reeds bestaande patient object gaat. Indien het object reeds bestaat wordt het gequeried. Daarna worden alle properties van het object voorzien van de waarden zoals meegegeven in de aanroep van de service. Datum waarden dienen bij de huidige aanpak nog handmatig van een string waarde naar een datum waarde omgezet te worden. Volgende stap is om het patient object te persisteren. Indien dit succesvol verloopt wordt het result weer als JSON teruggegeven. In het geval van een error wordt er gebruik gemaakt van de standaard faciliteiten van HTTP voor het communiceren van errors. De bijbehorende code voor deze bewerking is te vinden op www.javamagazine.nl/site/Hetblad/Extra.html onder codevoorbeeld 10.

## Conclusie

jQuery en Grails maken het mogelijk om productief en relatief eenvoudig complexe webapplicaties te bouwen waarbij de viewlaag volledig in de browser is geïmplementeerd. Een webframework op de server wordt hiermee overbodig. De clientapplicatie is volledig stateful, wat ontwikkelen een stuk eenvoudiger maakt. «

Behalve de codevoorbeelden kan ook het complete artikel worden gedownload op www.javamagazine.nl/site/Hetblad/Extra.html