Er voltrekt zich op dit moment een revolutie op het gebied van webapplicatie-ontwikkeling. Deze revolutie betreft het vervangen van het vermaledijde page-reload paradigma door het page-update paradigma. De term die hiervoor in zwang is geraakt is Ajax (Asynchronious JavaScript and XML), naar aanleiding van een artikel van Jesse James Garret.

Webapplicaties worden volwassen met Ajax

Nieuwe techniek is eenvoudig maar revolutionair

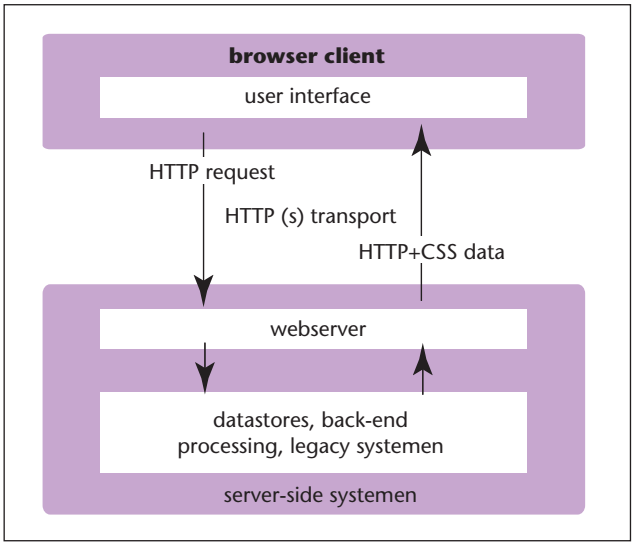
Dit artikel laat zien hoe met de Java servlet-technologie in combinatie met Ajax relatief eenvoudig een UI framework kan worden gebouwd om component based en event driven thin clients te bouwen.

De in dit artikel gepresenteerde architectuur heeft de elegantie van de eenvoud en brengt het ontwikkelen van rich thin clients binnen het bereik van elke Java-ontwikkelaar. *Gebruikers* kunnen straks webapplicaties enkel nog onderscheiden van desktopapplicaties doordat ze in een browser draaien (en in de regel minder saai ogen). *Ontwikkelaars* kunnen straks webapplicaties bouwen zoals ze desktop applicaties bouwen: component based en event driven.

'RICH CLIENT' De grote tekortkoming van de gangbare thin client-architectuur is de povere gebruikerservaring. Dit leidde enkele jaren geleden nog tot de opkomst van de rich 'thin client'-architecturen. Applicaties die gebouwd worden volgens deze architecturen zouden zowel thin als rich zijn: het beste van twee werelden. Toch hebben deze 'rich client'-architecturen geen grote vlucht genomen: de meeste waren of niet echt 'thin' of niet echt 'rich'. Daarmee bleef de klassieke thin client de norm: een webapplicatie op basis van HTML.

EXIT: PAGE-RELOAD PARADIGMA Het zwakste punt van de klassieke thin client architectuur is de page reload cyclus. Dit betekent dat elke client-server interactie verloopt via een submit van een HTML FORM. Met deze submit verstuurd de browser een request met alle door de gebruiker gewijzigde gegevens naar de server. De server retourneert vervolgens een compleet nieuwe HTML-pagina. De browser laadt deze en presenteert hem aan de gebruiker (figuur 1).

Deze cyclus is bijzonder processorintensief voor netwerk, server en client. Het vereist een grote mate van creativiteit om hiermee toch een enigszins rijke gebruikersinterface te bouwen. Dit geldt zowel voor de serverzijde als de client-zijde (met JavaScript en DHTML). Het sleutelwoord hierbij is caching. Om de inherente complexiteit op server en client beheersbaar te houden zijn er de afgelopen jaren duizenden UI-frameworks en toolkits ontwikkeld. De resultaten zijn vaak verbluffend, maar uiteindelijk staan de beschikbare processorkracht en netwerkbandbreedte een 'echt' rijke gebruikerservaring op basis van deze inefficiënte architectuur in de weg.

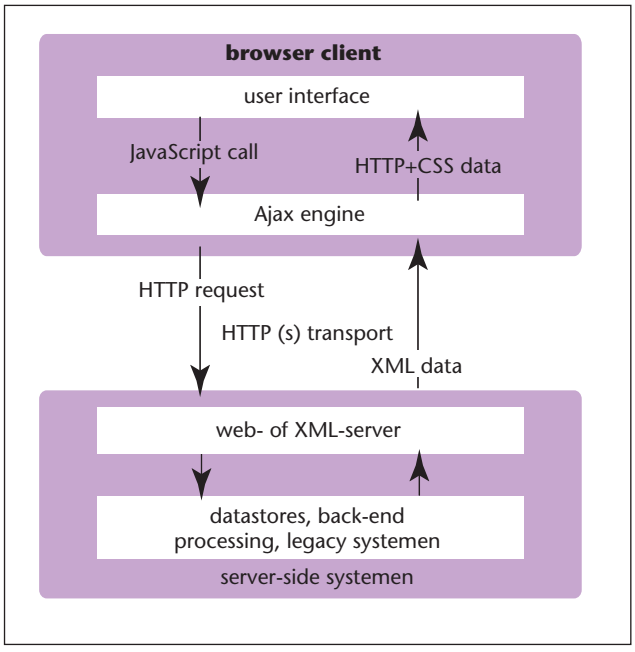


FIGUUR 1. Klassiek webapplicatiemodel.

ENTER: PAGE-UPDATE PARADIGMA De thin client-architectuur op basis van Ajax daarentegen is gebaseerd op de page update cyclus. In tegenstelling tot de klassieke webapplicaties laadt de browser nu nog maar één pagina. Deze wordt bij het opstarten van een webapplicatie geladen. Vervolgens worden selectief acties door de gebruikers gedelegeerd naar de Ajax engine(JavaScript). Deze verstuurd een XMLHttpRequest naar de server. De server verwerkt de request en retourneert een XML. Aan de hand van de data in deze XML past de Ajax-engine waar nodig de pagina in de browser aan.

De hoeveelheid data die over het netwerk wordt verstuurd en die door client en server moet worden verwerkt is dus minimaal in deze nieuwe generatie thin clients. Dit betekent dat deze nieuwe thin clients veel sneller kunnen reageren op de gebruiker. Omdat de achterliggende technologie ook mogelijk maakt dat er asynchroon gegevens worden uitgewisseld tussen client en server, kan de gebruiker in veel gevallen tijdens een page update-cyclus zelfs gewoon doorwerken en hoeft hij in het geheel niet te wachten.

Maar het vervangen van de page reload-cyclus zal niet alleen resulteren in een nieuwe generatie thin clients die efficiënter met hardware en brandbreedte omgaan. Het zal uiteindelijk ook leiden tot de opkomst van de 'rich thin client'. Dit zijn pure thin clients die een net zo rijke of geavanceerde UI bieden als desktopapplicaties doen. Deze rich thin client webapplicaties zullen net zoals desktopapplicaties component based en event driven worden opgebouwd. Dit zal gebeuren met behulp van component based event driven webapplication frameworks.



FIGUUR 2. Ajax webapplicatie-model.

EVENT DRIVEN EN COMPONENT BASED In een event driven component based webapplication framework manipuleren controllers op de server een geneste structuur van UI componenten op basis van UI events. Het webapplication-framework zorgt vervolgens op transparante wijze voor:

- de reproductie van de geneste structuur van server componenten op de client,
- de synchronisatie van beide structuren met elkaar,
- de notificatie van servercomponenten van events waarop zich controllers zich als listeners hebben geabonneerd, zodat de servercomponent op zijn beurt deze controllers kan notificeren.

Dergelijke frameworks bestaan er al geruime tijd in verschillende smaken. De eerste en grootste categorie vereist de installatie van additionele software op de client. Daarmee zijn de resulterende applicaties niet echt thin. Aanbieders vinden overigens vaak van wel, meestal met als belangrijkste argumenten dat de installatie van de additionele software verregaand is geautomatiseerd en dat de businesslogica op de server wordt uitgevoerd.

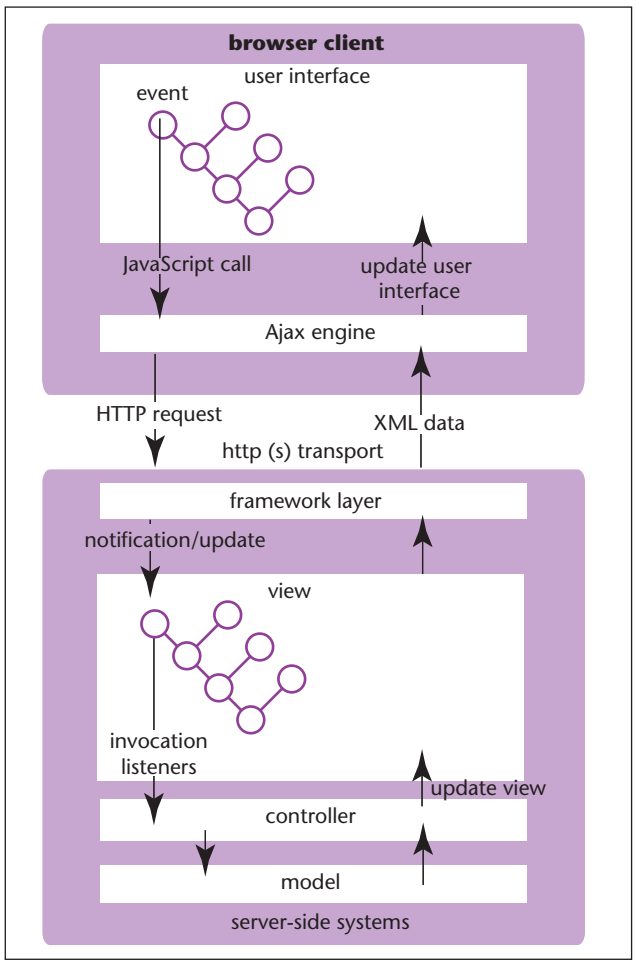
Daarnaast bestaan er ook event driven, component based UI Frameworks die gebaseerd zijn op de traditionele thin client-architectuur. Vanwege de achterliggende page reload cyclus zijn de hiermee gebouwde applicaties niet echt rich of niet echt responsive. Willen we werkelijk rich thin clients te kunnen bouwen met deze frameworks dan moet de page reload cyclus in deze frameworks vervangen worden door een page update cyclus. Ajax is de sleutel. Dit is ook exact wat we nu zien gebeuren. Een mooi voorbeeld hiervan is Echo: een volledig event driven, component based open source webapplication framework, dat nu herbouwd is op basis van de Ajax-technologie en kan worden gedownload als Echo2.

FRAMEWORK OP BASIS VAN AJAX De Ajax-technologie past dus uitstekend in de architectuur van een event driven component based webapplicatie-framework (zie figuur 3). In een webapplicatie die met een dergelijk framework gebouwd is, worden user events gedelegeerd aan de Ajax-engine, mits listeners zich op de server hiervoor hebben geabonneerd. Deze stuurt het event door naar de server. Het framework op de server zorgt er vervolgens voor dat het corresponderende server-side component wordt geïdentificeerd en genotificeerd. Deze component roept vervolgens alle controllers aan die zich als listeners op dit event hebben geabonneerd. De controllers kunnen op basis van dit event het model aanpassen of raadplegen en eventueel aanpassingen doorvoeren op de UI. Uiteindelijk krijgt het framework de controle terug en zal het alle UI-wij-

zigingen verzamelen en in de vorm van XML-data retourneren aan de client. Op de client tenslotte zal de Ajax-engine op basis van deze data de pagina aanpassen zodat deze weer in overeenstemming is met de server.

De hiervoor geschetste architectuur kan op verschillende manieren verder worden uitgewerkt. Zo kan het renderen van de HTML op de client of op de server plaatsvinden. Renderen op de server is de eenvoudigste en meest efficiënte oplossing omdat de componenten in dat geval op de server meteen naar het goede formaat kunnen worden omgezet. Het alternatief is dat ze op de server naar een tussenformaat worden omgezet en dat vervolgens dit tussenformaat op de client weer moet worden omgezet naar HTML. Kiezen voor renderen op de server betekent ook dat de Ajax-engine uiterst eenvoudig en de hoeveelheid JavaScript op de client tot het absolute minimum beperkt kan blijven. Hier wordt daarom gekozen voor renderen op de server. Dit betekent dat het webapplicatie framework een XML naar de server stuurt met een aantal HTML-fragmenten.

CLIENT: AJAX ENGINE De Ajax engine is geschreven in JavaScript. In het voorbeeld is de Ajax-engine



FIGUUR 3. Ajax-technologie in de een event driven, component based webapplicatie-framework.

opgenomen in de HTML. Dit is niet inefficiënt omdat er in beginsel maar één pagina naar de browser wordt gestuurd. De Ajax-engine wordt dus met de openingspagina meegestuurd. Voor het gemak veronderstellen we dat deze openingspagina tevens een invoerveld en een button bevat. De complete pagina ziet er dan als volgt uit:

```
<html>
<head>
…
<SCRIPT LANGUAGE="JavaScript">
<!--

var http = window.ActiveXObject ? new
ActiveXObject("Microsoft.XMLHTTP"): new
XMLHttpRequest();

function doGet(id,event) {…}

function doPost(id,event,value) {
    http.open('POST', '/vegax/examples'+'?eve
nt='+event+'&id='+id+'&value='+escape(value),t
rue);
    http.setRequestHeader('Content-
Type','application/x-www-form-urlencoded');
    http.onreadystatechange = handleResponse;
    http.send(null);
}

function handleResponse() {
   if(http.readyState == 4 && http.status ==
200){
        var xmlDocument = http.responseXML;
        if(xmlDocument){
          var items = xmlDocument.getElements
ByTagName("item");
          for(var i = 0 ; i < items.length;
i++){
            var elm = document.
getElementById(items[i].getAttribute("id"));
            if (elm){
              elm.innerHTML = items[i].first-
Child.data;
            }
          }
        }
   }
}

-->
</SCRIPT>
</head>
<body>
   <span id='0'>
      Your name:
      <span id='1'><input onBlur="doPost('1',
'onBlur',value);" type=”text”  value="" /></
span>
      <span id='2'><b></b></span><br/>
      <span id='3'><input onClick="doGet('3',
'onClick');" type="button" value="OK" /> </
span>
   </span>
</body>
</html>
```

In de header wordt de Ajax-engine gedeclareerd. In de body zien we het invoerveld en de button. Op het invoerveld wordt bij een onBlur event de functie doPost

aangeroepen van de Ajax-engine. Op de button wordt bij een onClick event de functie doGet aangeroepen op de Ajax engine. Dit betekent dat zich op de server tenminste één object als listener heeft geabonneerd op de het onBlur-event van het inputveld en tenminste één object op de het onClick-event van de button.

Als argumenten bij het aanroepen van de methoden op de Ajax engine wordt tenminste het id en het type event meegegeven. Het id stelt het framework op de server in staat de corresponderende component op de server te identificeren en te notificeren van het event. De genotificeerde component roept vervolgens de listeners aan die zich bij hem op dit event hebben geabonneerd. Het id correspondeert tevens met het id van het span element waarbinnen de betreffende component is opgenomen. Dit stelt de Ajax engine in staat om componenten snel te updaten.

XMLHttpRequest Het JavaScript object waar alles om draait in de Ajax-engine is het XMLHttpRequest. Dit object verzorgt de communicatie met de server. De eerste regel in de Ajax engine initieert dit object op een browser-onafhankelijke wijze:

```
var http = window.ActiveXObject ? new
ActiveXObject("Microsoft.XMLHTTP"): new
XMLHttpRequest();
```

De doGet en doPost zijn nagenoeg gelijk en we beperken ons tot de doPost functie.

De eerste regel van de doPost functie opent een HTTP POST-connectie naar de server waarbij een query string met de argumenten id, event en value zijn toegevoegd. Het laatste argument geeft aan dat de request asynchroon moet worden uitgevoerd.

```
http.open('POST', 'vegax/example?event='+even
t+'&id='+id+'&value='+escape(value),true);
```

De volgende regel spreekt voor zich.

```
http.setRequestHeader('Content-
Type','application/x-www-form-urlencoded');
```

Met de vierde regel zorgen we ervoor dat de 'call back'-notificaties wordt afgehandeld worden door de functie handleResponce van de Ajax-engine:

```
http.onreadystatechange = handleResponse;
```

De laatste regel verstuurt de data naar de server.

```
http.send(null);
```

Omdat we in de eerste regel hebben opgegeven dat de request asynchroon afgehandeld moet worden zal de aanroep van functie send() onmiddellijk retourneren. Vandaar ook dat de handleRepsonse aan de functie onreadystatechange is gekoppeld. Wanneer we er echter voor kiezen om de XMLHttpRequest synchroon uit te laten voeren dan is dit niet nodig en kan de handleResponce functie direct aangeroepen worden na aanroep van send():

```
function doPost(id,event,value) {
    http.open('POST', '/vegax/examples'+'?eve
nt='+event+'&id='+id+'&value=
'+escape(value),false);
    http.setRequestHeader('Content-
Type','application/x-www-form-urlencoded');
    http.send(null);
    handleResponse();
}
```

Het synchroon uitvoeren van de request lijkt een goed idee, maar heeft als nadeel dat bij netwerk problemen de client blijft 'hangen'.

Update van de pagina Op de server wordt aan de hand van het id de corresponderende server-component opgezocht en genotificeerd. Deze roept de op het event geabonneerde listeners aan. Nadat de listeners hun werk hebben gedaan, verzamelt het framework de wijzigingen in de view en verstuurt deze als verpakt in een XML-bericht terug naar de client. De geretourneerde XML zou er als volgt uit kunnen zien:

```
<?xml version=\"1.0\" standalone=\"yes\"?>
<update>
  <item id=”2” ><![CDATA[<b>Jan Jansen</b>]]><
/item>
</update>
```

Dit zou dan betekenen dat er één object met id 2 is gewijzigd op de server. De handeResponse functie moet dit element nu vervangen in de pagina. Dit gebeurt in de kern van de handeResponse-functie:

```
var xmlDocument = http.responseXML;
  if(xmlDocument){
     var items = xmlDocument.getElementsByTag
Name("item");
       for(var i = 0 ; i < items.length;
i++){
         var elm = document.
getElementById(items[i].getAttribute("id"));
          if (elm){
             elm.innerHTML = items[i].first-
Child.data;
          }
       }
 }
```

WEBAPPLICATIE FRAMEWORK Zo eenvoudig als de Ajax engine is het framework niet. Complex is het evenmin. In essentie bestaat het slechts uit drie classes: Application, ApplicationServlet en Component. De eerste twee classes, Application en ApplicationServlet, zijn abstracte classen die webapplicatiebouwers moeten implementeren. Application heeft niets om het lijf, zoals we zullen zien. De ApplicationServlet is een gewone servlet die alle requests afhandelt. Component tenslotte is de super-class van alle server componenten en is de class die ook het meeste werkt verricht.

APPLICATION CLASS De class Application stelt weinig voor:

```
public abstract class Application implements
Serializable{
  private Window window;

  protected abstract Window init() throws
Exception;

  public Window getWindow() throws Exception{
     return window==null?window=init():window;
  }
}
```

Er is slechts een methode die de applicatie bouwer hoeft te implementeren: de init() methode. Deze retourneert een Window (een subclass van Component) met de openingspagina. In het geval van de obligate HelloWorld-webapplicatie ziet de implementatie er als volgt uit:

```
public class HelloWorldApplication extends
Application {

  protected Window init() throws Exception{
     return new HelloWindow();
  }

}
```

Dit is de HelloWindow-class:

```
public class HelloWindow extends Window {

  public HelloWindow() throws Exception {
     setTitle("HelloWorld");
     HTML html = new HTML();
     html.insert("Hello World!");
     add(html);
  }

}
```

APPLICATIONSERVLET CLASS De ApplicationServlet is eveneens een abstracte class met een abstracte methode die geïmplementeerd moet worden: de methode newApplicationInstance():

```
public abstract Application newApplicationIn-
stance();
```

Deze methode geeft een implementatie van Application-class terug. In het geval van de HelloWorld-webapplicatie ziet de implementatie van de ApplicationServlet er dus als volgt uit:

```
public class HelloWorldServlet extends
ApplicationServlet{

  public Application newApplicationInstance()
{
     return new HelloWorldApplication();
  }

}
```

Uiteraard zal de implementatie van ApplicationServlet ook gedeclareerd moeten worden in de web.xml. De WebApplicationServlet zelf ziet er met weglating van de Exception afhandeling als volgt uit:

```
public abstract class ApplicationServlet
extends HttpServlet {
  private String APPLICATION =  getClass().
getName();

  public abstract Application newApplication-
Instance();

  protected void doPost(HttpServletRequest
req, HttpServletResponse resp) …{
     doGet(req, resp);
  }

  protected void doGet(HttpServletRequest req,
HttpServletResponse resp)…{
     HttpSession session = req.getSession();
     Application appl = (Application) sessi-
on.getAttribute(APPLICATION);
     String id = req.getParameter("id");
     if (appl == null || id == null ||
"".equals(id)) {
        if (appl == null) {
           appl = newApplicationInstance();
           session.setAttribute(APPLICATION,
appl);
        }
              //Page (re)load
        getPage(req, resp, appl.getWindow());
     } else {
        //Page update
        getUpdates(req, resp, appl.getWin-
dow(), id);

     }

  }

  private void getPage(HttpServletRequest req,
HttpServletResponse resp, Window window)…{
     PrintWriter out = resp.getWriter();
     out.println("<!DOCTYPE HTML …
><html><head> …");
     JavaScriptLibrary.main(out, req.getCon-
textPath()+ req.getServletPath());
```

Advertentie

```
    out.println("</head><body>");

    window.streamComponent(out);

    out.println("</body></html>");
  }

  private void getUpdates(HttpServletRequest
req, HttpServletResponse resp, Window window,
String id)…{

    Component container = window.
getComponentById(Long.parseLong(id));
       container.handleUIInvocation(request);

    resp.setContentType("text/xml");
    PrintWriter out = resp.getWriter();
    out.println("<?xml version=\"1.0\"
standalone=\"yes\"?>");
    out.print("<update>");
    window.streamUpdates(out);
    out.print("</update>");
  }



}
```

Aanroepen van doPost worden gedelegeerd naar de doGet methode. De doGet methode bestaat uit twee takken. De eerste tak roept de getPage() methode aan die een volledig nieuwe pagina op de response zet, de tweede tak roept de methode getUpdates() aan die een XML met updates voor de client op de response zet.

De methode getPage wordt aangeroepen wanneer een nieuwe client voor het eerst de applicatie aanroept of wanneer de refresh of backbutton wordt gebruikt. In alle andere gevallen wordt getUpdates() uitgevoerd.

GETPAGE() METHODE De kern van de methode getPage bestaat uit de aanroep van streamComponent op de Window van de applicatie. Deze methode is gedefinieerd op de Component en zet de complete component inclusief alle embedded componenten als HTML op de stream:

```
public abstract class Component implements
Serializable {
…
 final public void streamComponent(PrintWrite
r out) {
    out.print("<span id='");
    out.print(id);
    out.println("'>");

    streamContent(out);

    out.println("</span>");
    modified = false;
  }

  protected void streamContent(PrintWriter
out) {
    if (components != null) {
       for (int i = 0; i < components.
size(); i++) {
          ((Component) components.get(i)).
streamComponent(out);
       }
    }
  }
 …
}
```

De methode streamContent zal nagenoeg altijd door subclasses van Component worden overschreven. In het geval van een minimale implementatie van een List-component zou deze er als volgt kunnen uitzien:

```
public class List extends Component {
  …
  protected void streamContent(PrintWriter
out) {
    ArrayList list = getComponents();
    out.println(ordered?"<ol>":"<ul>");
    for (int i = 0; i < list.size(); i++) {
      out.println("<li>");
      ((Component)list.get(i)).
streamComponent(out);
      out.println("</li>");
    }
    out.println(ordered?"</ol>":"</ul>");
  }
  …
}
```

GETUPDATES() METHODE In de methode getUpdates() wordt eerst de Component opgehaald waarop het event heeft plaatsgevonden. Dit gebeurt door aanroep van de methode getComponentById op het Window-object. Ook deze methode is gedeclareerd op het Component-object:

```
public abstract class Component … {
 …

public final Component getComponentById(long
id)…{
    if (this.id == id) {
       return this;
    } else if (components != null) {
       for (int i = 0; i < components.
size(); i++) {
          Component component = ((Component)
components.get(i))
             .getComponentById(id);
          if (component != null)
             return component;
       }
    }
    return null;
  }
 …
}


Op de door de methode getComponentbyId()
geretourneerde Component wordt vervolgens de
methode handleUIInvocation aangeroepen:
```

```
public abstract class Component … {
 …
 public final void handleUIInvocation(HttpSer
vletRequest req){
     update(request);
     notifyListeners(req.getParameter(EVENT));
  }

 public void update(HttpServletRequest req){}
 …
}
```

De methode update kan door subclasses worden overschreven. Dit stelt hen in staat om zich te updaten met de door de client op de request gezette data. De methode notifyListeners doet wat de naam zegt. Nadat de listeners hun werk hebben kunnen doen zet de methode getUpdates de XML met updates op de response. Dit gebeurd in de volgende zes regels:

```
resp.setContentType("text/xml");
PrintWriter out = resp.getWriter();
out.println("<?xml version=\"1.0\"
standalone=\"yes\"?>");
out.print("<update>");
window.streamUpdates(out);
out.print("</update>");
```

Belangrijk is het zetten van het juiste response content type (text/xml). Alleen wanneer de juiste content-type is gezet én de XML valid is zet het XMLHttpRequest-object op de client de opgebouwde XML als DOM-object in de variabele responseXML. De updates zelf worden op de response geplaatst door de aanroep van de methode streamUpdates op het Window-object. Ook deze methode is weer als final methode gedeclareerd op de Component class:

```
public abstract class Component … {
 …
 final public void streamUpdates(PrintWriter
out) {
     if (modified) {
        out.print("<item id=\"");
        out.print(id);
        out.print("\"><![CDATA[");

        streamContent(out);

        out.print("]]></item>");
        resetModified();
     } else if (components != null) {
        for (int i = 0; i < components.
size(); i++) {
           ((Component) components.get(i)).
streamUpdates(out);
        }
     }
  }
 …
}
```

De streamUpdate methode roept recursief alle componenten aan. Een component die gewijzigd is zet zichzelf (inclusief alle onderliggende componenten) als HTML verpakt in een CDTA element op de XML met updates. De methode streamContent die hiervoor wordt aangeroepen wordt ook aangeroepen in de getPage() methode en is al besproken.

COMPONENT CLASS De belangrijkste methode van de class Component hebben we hiervoor al besproken. Daarnaast beheerst deze klasse de listeners, componenten, attributen en ten slotte de modified flag. De Component-class bezit een aantal eenvoudige methoden om deze variabelen te wijzigen of te raadplegen.

SAMENVATTEND Ajax is een eenvoudige, maar revolutionair nieuwe techniek op het gebied van webapplicatie ontwikkeling. Het revolutionaire karakter van deze techniek betreft het vervangen van de inefficiënte page reload cyclus van de klassieke thin client door de page update-cyclus. Dit brengt een nieuwe generatie thin clients binnen bereik die in niets onderdoen voor desktop applicaties. Net als desktop-applicaties zijn deze volledig component based en event driven.

In dit artikel hebben we laten zien hoe met Ajax op relatief eenvoudige wijze een webapplicatie-framework kan worden gebouwd waarmee applicatie-ontwikkelaars deze nieuwe generatie thin cliënts kunnen bouwen. Zelfbouw is echter niet noodzakelijk. De eerste op Ajax gebaseerde open source webapplicatieframeworks voor het ontwikkelen van de nieuwe generatie thin clients zijn al beschikbaar voor Java-ontwikkelaars: Echo2 en jWic. Voor de ontwikkeling van professionele applicaties zijn deze frameworks op dit moment nog wat mager. Daar staat tegenover dat ze wel volop in ontwikkeling zijn. Bovendien zijn ook al de eerste commerciele frameworks beschikbaar voor Java-ontwikkelaars. Voor wie dit geen optie is en niet wil of kan wachten, moet aansluiting zoeken bij de bestaande initiatieven of zelf de hand aan de ploeg slaan.

Het besproken voorbeeld van een webapplicatie-framework kan gedownload worden als Eclipse project op: http://www.release.nl.

drs. Huub Cleutjens werkt als Software Research Engineer Financial Solutions bij IBS Nederland (huub.cleutjens@ibs.nl).