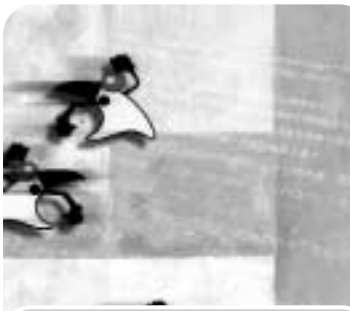
De queeste naar het vlot ontwikkelen van webgebaseerde applicaties gaat nog steeds onverminderd door. Marc Portier, werkzaam bij Outerthought en actieve Apache Cocoon committer, brengt ons in dit artikel verslag van de nieuwste ontwikkelingen op dat gebied binnen het Apache Cocoon project.



thema

Flow-Controle in Java gebaseerde webapplicaties

Apache Cocoon en de MVC-architectuur

De webgerelateerde API's (Servlet, JSP, Taglibs) gelden als de volwassen grote broer van het J2EE gezin. Ze staan al een tijdje bekend voor hun stabiliteit en betrouwbaarheid en dat wordt ook uitgedragen door de kwaliteit van de container implementaties. De laatste specificatie updates wijzigen wat punten en komma's, en brengen wat verduidelijkingen in grijze zones, maar brengen duidelijk niet de conceptuele wijzigingen of essentiële nieuwe features die bijvoorbeeld de EJB-tak van de familie de laatste jaren nog heeft meegemaakt. Echter, zoals het een goede familiekroniek betaamt gaat het verhaal natuurlijk wel (eindeloos?) verder. De queeste naar het snel ontwikkelen van webgebaseerde applicaties gaat nog steeds onverminderd door, zoals ook in het Apache Cocoon project.

De Java web-API's hebben aanleiding gegeven tot een grote schare aan 'webapplicatie raamwerken', die nauwelijks te overzien is. Deze (met enige goede wil) 'afstammelingen' zijn niet zelden open source projecten die telkens weer vertrekken van een aantal breed herkenbare noden in de bouw van web applicaties. Noden die door de bestaande API's eerder '*oplosbaar*' dan wel '*al opgelost*' zijn. Het raamwerk in deze sector dat u minstens qua naam behoort te kennen is 'Apache Struts'. De product- en serviceondersteuning van een aantal grote commerciële entiteiten zal aan die bekendheid niet vreemd zijn. Velen gebruiken dit of een ander raamwerk tot hun voldoening. En het moet gezegd: de meeste van deze projecten slagen er ook in voldoende snel de essentiële nieuwe features van elkaar op te pikken. Een convergentie in de ene of andere richting valt uiteindelijk dus wel te verwachten. Maar voor we daar zijn blijft er ruimte voor een soort arbitrage die verondersteld dat enige feature-scouting bij de verschillende projecten gebeurt.

Wat dat betreft verdient het Apache Cocoon[1] (http://cocoon.apache.org/) project eigenlijk meer dan wat terloopse aandacht: het brengt een bijzonder verfrissende kijk op het web voor zowel publicatie- als applicatie-toepassingen en maximaliseert zo ook de oplevering van de grote XML beloftes.

TERUG NAAR DE BRON Het uitgangspunt bij elke verwerking binnen Cocoon is de HTTP Request URI. Het zal de ervaren web-programmeur niet echt verbazen: de web-container steunt op het Inversion of Control (IoC) principe[2], wat voordraagt dat een Servlet pas door zijn container wordt aangeroepen als er effectief een HTTP request te verwerken is. Die request vertrekt pas bij een browser (algemeen user-agent) als de bestemming gekend is, en dat is precies die befaamde

1 Recentste release van dit raamwerk draagt versienummer 2.1.1. Het leefde sinds maart 1999 onder de koepel van XML gerelateerde Apache projecten op xml.apache.org maar heeft bij de laatste projectreorganisatie daar zijn eigen top-level status afgedwongen waardoor het naast projecten als Apache Jakarta en Apache Ant komt te staan.

2 Als memotechnisch middeltje: Het IoC principe wordt ook wel eens het 'Hollywood' principe genoemd, waarna de befaamde quote "don't call us, we'll call you" wordt aangehaald. Het dient aan te geven dat de servlet-componenten voor hun lifecycle en verwerkings-triggers totaal afhankelijk zijn van hun container.

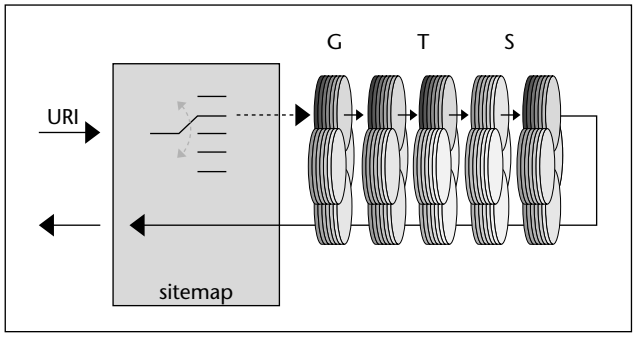
URI. Er is evenwel meer dan dit louter technische gegeven. De universele adressering en naamgeving die de URI aan de Resources op het web heeft verleend is door het effectieve gebruik ervan voor dynamische applicaties eigenlijk verworden tot een universele adressering van "*Services*". Het effect hiervan is dat de URI in essentie de rol opneemt van de applicatie-interface: ze wijzen (bijna net als methodenamen) specifieke aan te roepen diensten aan. Het gekende belang van een gedegen ontwerp en beheer van de applicatie interface manifesteert zich dan ook in webapplicaties zoals na te lezen is in verschillende publicaties van webarchitectuur goeroes:

- Tim Berners Lee, "Cool URI's don't change" (http://www.w3.org/Provider/Style/URI.html)
- Jacob Nielsen, "URL as UI" (http://www.useit.com/alertbox/990321.html)
- Paul Prescod, "Location, Location, Location: Web Services Need Jury's" (http://www.prescod.net/rest/importance_of_uris.html)

De conclusie dringt zich op: de URI's die door je server behandeld worden kies je niet licht. Net zoals je Java namen voor packages, classes, interfaces en public methods laat je die niet aan het toeval over. Evenmin ben je gediend van een platform dat je daarin het een of het ander opdringt. Helaas zijn de gevolgen van deze gedachtengang niet even makkelijk te verwerken: de verantwoordelijkheid kaatsen we nu wel terug naar de webontwikkelaar: "Doe er iets aan en beheer je URI namespace!"

Dat nieuwe gedeelte van de taak krijgt in Cocoon meteen een centrale plaats: door je URI-namespacedesign neer te schrijven in zijn 'sitemap' kun je dat extra werk meteen verzilveren. Cocoon zet de file namelijk om in een run-time dispatcher die binnenkomende URI's op hun patroon weet te herkennen, er parameters uit destilleert en die in één adem weet door te geven aan XML gebaseerde publicatie-pijplijnen (zie figuur 1 en codevoorbeeld 1:

De XML Loodgieter Het voorbeeld toont meteen ook de publicatie-pijplijnen aan het werk. Daarin zijn drie soorten componenten actief: Generators, Transformers en Serializers. Deze drie soorten bouwstenen hebben elk hun eigen functie: de Generators produceren een XML datastroom in de vorm van SAX-events[3] die ze doorsturen naar de hun door de sitemap aangeleverde XMLConsumer. Deze



Figuur 1.

laatste is een interface die zowel de Transformers als de Serializers implementeren. De Transformers zijn er om de binnenkomende XML datastroom om te zetten naar een uitgaande XML datastroom die wordt doorgestuurd naar de volgende XMLConsumer in de rij. Dit omzetten bestaat typisch in het herkennen van specifieke XML elementen in de binnenkomende stroom en die uit te filteren of te vervangen door nieuwe datastructuren. Op

```
<!– OPMERKING: de aangehaalde URI's binnen
deze context
     zijn relatief ten opzichte van de server
     waarop ze draaien.
     De effectieve browser-URI's zullen dus
     nog het
     http://servername.domain.nl/cocoon/ pre-
     fix dragen. –>

<map:pipelines>
  <map:pipeline>

     <!– pijplijn getriggered door lege URI:
     "" –>
     <map:match pattern="">
        <map:generate  type="file" src="welco-
        me.xml"/>
        <map:transform type="trax" src="samp-
        les2html.xsl" />
        <map:serialize type="html" />
     </map:match>

     <!– pijplijn getriggered door alle URI's
     van
        een bepaald type: "*/*.jx" –>
     <map:match pattern="calc/*">
        <map:generate  type="serverpages"
        src="calc/{1}.xsp"/>
        <map:transform type="trax" src="simp-
        le-page2html.xsl"/>
        <map:serialize type="html" />
     </map:match>

  </map:pipeline>
</map:pipelines>
```

Codevoorbeeld 1

3 SAX staat voor Simple API for XML en is een event-gebaseerde API voor het parsen, valideren en communiceren van XML structuren. Ze is in het public domein ontstaan en is na zijn oorspronkelijke Java roots ook naar een aantal andere programmeeromgevingen geporteerd. SAX geldt als een efficiënter alternatief voor DOM dat met name in dit soort 'streaming' omgevingen uitblinkt.

Naam	Beschrijving
Generators	
FileGenerator (default)	Parst een XML file van een URL (meestal vanaf schijf) als input voor de pijplijn.
HTMLGenerator	Gebruikt Jtidy om de html file vanaf een bepaalde URL (vaak niet op schijf) naar geldige XML om te zetten alvorens die als input voor de pijp te gebruiken.
DirectoryGenerator	Codeert de inhoud van een directory als een XML structuur die als input voor de pijp wordt gebruikt.
JSPGenerator	Gebruikt de output van een klassieke JSP page als input voor de pijplijn.
JXGenerator VelocityGenerator XSPGenerator	Verschillende template-language alternatieven (net zoals JSP) die toelaten property's van klaargezette Java objecten op aangeduide plaatsen in een XML document in te vullen en het resultaat als input van de pijplijn aan te wenden
RequestGenerator	Codeert de verschillende property's van het HTTPRequest object in een XMLstructuur die de input van de pijplijn vormt.
WebServiceProxyGenerator	Integreert de reply van een geconsulteerde WebService in de pijplijn.
Aggregator	Strikt genomen een alternatief voor de Generator die de output van verschillende pijplijnen samenbrengt.
Transformers	
XSLTTransformer (default)	Transformeert de SAX events volgens de beschrijvingen van een XSLT stylesheet.
I18NTransformer	Herkent elementen en attributen in een specifieke 'i18n' namespace en gebruikt hun waarden als look-up keys in vertaaltabellen om verschillende talen te ondersteunen.
LogTransformer	Transformeert de SAXevents niet, maar schrijft ze naar een logfile voor debugging.
SQLTransformer	Herkent elementen in een specifieke 'sql' namespace, filtert ze uit en vervangt de sql-statements die ze meedragen door de resulterende data uit de database.
XIncludeTransformer	Implementatie van de W3C XInclude recommendation (inclusief xpointer en xml base)
Serializers	
HTMLSerializer (default)	Modificeert de XHTML output van de pijp zodat ze HTML compliant wordt (HTML kent geen <empty /> syntax en verkiest dat sommige attributen geen waarde hebben.)
XMLSerializer	Stuurt het pure XML resultaat van de pijplijn naar de browser.
HSSFSerializer	Schrijft het numerieke XML formaat op het einde van de pijplijn weg in het binaire MS Excel formaat. (Op basis van Apache POI)
RTFSerializer	Rendert XSL-FO als RTF wat vaak als input voor MS Word wordt gebruikt.
PDFSerializer	Rendert XSL-FO als PDF. (op basis van Apache FOP)
SVG/JPEGSerializer SVG/PNGSerializer SVG/TiffSerializer	Rastert het SVG resultaat van de laatste transformer in de pijp tot een van de gekende binaire bitmap formaten. (op basis van Apache Batik)
ZipArchiveSerializer	Genereert een zip archief op basis van een XML structuur die de inhoud van de zip beschrijft. Inhoud van de gearchiveerde files kan gemixed inline of via url-referenties worden aangegeven.

TABEL 1. De Cocoon implementatie levert een grote set kant-en-klare Generators, Transformers en Serializers

het einde van een dergelijke keten die dus meerdere Transformers kan bevatten dient dan een Serializer om de resulterende logische XML structuur te 'serialiseren' naar een specifiek fysisch formaat dat over de HTTP connectie teruggestuurd wordt naar de browser. Het pijplijn-designpatroon dat hier wordt aangewend is overduidelijk geïnspireerd door wat de SAX API aanlevert, de waarde die de Cocoon implementatie eraan toevoegt is tweevoudig. Enerzijds hebben ze naast de logische data-verwerking-API (zeg maar: "de pure SAX van ContentHandler, XMLReader en XMLFilter") ook de nodige componentbeheer[4] methodes toegevoegd. Anderzijds leveren ze meteen een grote set aan kant en klare Generators, Transformers en Serializers (zie de tabel 1 voor een zeer onvolledige lijst).

In de goede OO traditie zijn elk van deze componenten ontworpen om een zo klein mogelijke atomaire taak zo goed mogelijk uit te voeren. De kracht zit dan in het gecombineerd gebruik ervan. Het ontwerp van elke deelcomponent houdt zich strikt aan het Separation of Concerns[5] (SoC) principe dat voorschrijft dat gescheiden verantwoordelijkheden door gescheiden objecten moeten worden opgenomen. Het eindresultaat is dat je eenvoudig en declaratief in de sitemap deze herbruikbare componenten slim aan elkaar kunt koppelen en zo

4 Zoals het bij open source projecten gebruikelijk is wordt het wiel niet graag opnieuw uitgevonden. De component-structuur die bij het ontwerp van Cocoon is gebruikt is het product van een ander Apache project: Apache Avalon (http://jakarta.apache.org/avalon).

dezelfde XML formaten aan de bron naar zeer uiteenlopende doel-formaten kunt laten omzetten en daarvoor onderweg mogelijks gelijklopende trajecten van dezelfde deel-componenten laat gebruik maken. Het produceren van deze of gene output blijft dan netjes getriggerd door zijn eenduidige URI. En de cirkel is rond: die URI's hadden we elk hun plaats gegeven in de beschikbare URI-request-ruimte van onze server. Bovenstaande architectuur maakt Cocoon meteen inzetbaar en is behoorlijk vernieuwend. Sinds de jongste release is het echter ook de onderbouw waarop interactieve webapplicaties worden gebouwd.

Het tweede model Voor de creatie van interactieve webapplicaties in Java geldt het zogenaamde 'Model-2' als het 'nec plus ultra'. De gouden formule komt erop neer dat men onderkent dat een webapplicatie zijn klassieke twee verantwoordelijkheden (1) HTTP request parameters verwerken door ze te mappen op uit te voeren back-end (trans-)acties en (2) mooie schermen met de resultaten op genereren het best kan realiseren door een combinatie van Servlets resp. JSP's eerder dan een monocultuur van de ene of de andere. Ze hebben dan ook hun eigen specialisme: de eerste laten vooral een eenvoudig uitschrijven van de logische verwerking toe, terwijl de laatste vooral uitblinken om als een soort template-taal de schermen te coderen. Binnen Cocoon vinden we aan de basis een identieke opdeling. Arbitraire logica wordt afgehandeld door de FlowProcessor en de presentatie is de verantwoordelijkheid van de XML publicatie pijplijnen. (zie hierboven) De lessen van Model-2 blijven alvast gelden: voor de publicatie en presentatie blijkt de declaratieve aanpak van de sitemap (XML formaat) een schot in de roos, voor het uitschrijven van de arbitraire logica grijpen we het liefst terug naar een algemene programmeertaal (Java-componenten dus).

Sinds de jongste release is het echter ook de onderbouw waarop interactieve webapplicaties worden gebouwd

Toch niet weer MVC? Met die arbitraire logica is in de modale webapplicatie helaas nog wel wat meer mis. Voor het modelleren ervan blijft men heel graag teruggrijpen naar de (ogenschijnlijk) beter gekende wereld van de GUI applicaties. Getuige hiervan zijn de ontelbare webapplicatie raamwerken die stellig beloven de ultieme WebMVC te hebben uitgedacht, daarmee gretig refererend naar het welhaast mythisch succespattern voor GUI applicaties. Marketing-term-sceptici van het eerste uur plegen nogal eens uit te halen naar dit soort krachttermen. De losse koppeling van het web mist namelijk de gesloten event-lus waar het MVC patroon wel van profiteert: je controller krijgt namelijk geen request (geen event) voor de 'BackButtonPressed', 'UserAgentClosed', of 'NetwerkConnectionDropped'. Toch blijft vanuit het oogpunt van de applicatiemodellering de MVC filosofie overeind: het Model krijgen we van de back-end, de View wordt gerealiseerd door de JSP of publicatiepijplijn en we wensen de flow van de applicatie te kunnen uitschrijven in een Controller. Dat laatste liefst zo dat ze rechtstreeks mapt op de use-case die in het analyse-model staat uitgeschreven.

Op dat laatste punt scoren de webapplicaties doorgaans slecht: de klassieke controller (bijvoorbeeld een servlet) in een webcontext heeft geen toestandsvariabelen en dient daarom bij elke te verwerken request eerst zijn context op te bouwen uit de aanwezige omgeving: typisch op basis van variabelen die in de Session worden opgeborgen. De technische limitatie dwingt zo de ontwikkelaar om de natuurlijke 'flow' van zijn applicatie uiteen te halen in enerzijds een datastructuur die de toestand van de use-case bijhoudt (bemerk dat dit niet de entiteiten zijn die onder het 'MODEL' verzameld worden), en anderzijds in een component die weet waar die toestand in de sessie te stockeren, hoe ze te wijzigen en op basis ervan de juiste 'VIEW' te selecteren.

Een voorbeeld Het nieuwe antwoord dat Cocoon op deze problematiek levert komt spontaan tot stand door zijn focus op het mappen van binnenkomende request-URI's. De klassieke vraag voor de webontwikkelaar "Hoe zal mijn web applicatie omgaan met het stateless karakter van HTTP?" wordt dan eerder zo gesteld: "Hoe moeten we de verschillende interactie-stappen binnen een use case coderen als URI's, en dus als Resources?" Gezien het interactieve karakter gaat het hier om 'dynamische' resources, en de wetmatigheid volgend zullen die opvraagbaar zijn op 'dynamische' URI's.

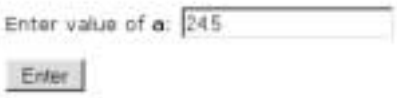
Nemen we het voorbeeld van de volgende 'rekenmachine' web-applicatie[6]. De use case blijft eenvoudig:

5 Separation of Concerns wordt als term toegeschreven aan wijlen E. W. Dijkstra. Hij argumenteerde dat het wezen van 'intelligent denken' gekenmerkt wordt door de mogelijkheid om elk van de verschillende aspecten van een bepaald onderwerp geïsoleerd in ogenschouw te kunnen nemen. Het ontbreken van dat talent leidt meestal tot een onoverzichtelijk en (ogenschijnlijk) onoplosbaar probleem.

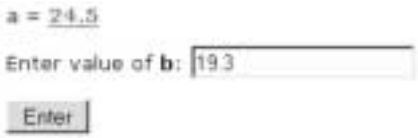
6 Dit voorbeeld is ook opgenomen in de Apache Cocoon (http://cocoon.apache.org) distributie. De source-code ervan is online na te lezen via http://cvs.apache.org/viewcvs.cgi/cocoon-2.1/src/blocks/apples/

vraag de gebruiker een eerste getal - vraag een tweede getal - vraag om een bewerking te selecteren - toon het resultaat van de bewerking op de twee getallen. De toegevoegde uitdaging (om het typische webflow probleem uit te lokken), is evenwel dat elke stap hierin ook als een gescheiden interactie wordt aangeboden. De verschillende schermen (resources) die bij lineair doorlopen van de use-case worden gepresenteerd zijn dan achtereenvolgens:

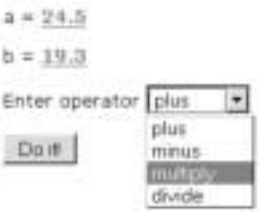
1. Welkom-scherm vanwaar we de link naar de 'rekenmachine' kunnen volgen
2. Een formulier dat een eerste getal vraagt.



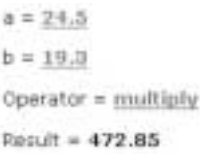
3. Een formulier dat een tweede getal vraagt. (en het eerste getal toont met een link terug om het te wijzigen)



4. Een formulier dat de berekening vraagt. (met echo van de input tot hier)



5. Een resultaat-pagina (ook al met input-echo en links terug)



Elk van deze Views kan eenvoudig vormgegeven worden door één van de beschikbare template generators binnen Cocoon te combineren met een klassieke XSLT transformer die een consistente *look and feel* aan de pagina's kan geven alsook toelaat een intern pagina-XML-formaat af te spreken dat zich tot de essentie beperkt. De ontwikkelaar van de template pagina hoeft dat dus niet in XHTML te doen, en dient dus niet alle nuances van het display-formaat te kennen. De mini-pijplijnen die we voor de productie van deze views nodig hebben worden dan door de sitemap (zie hoger) getriggered door een unieke URI voor elk ervan: [calc/getNumberA], [calc/getNumberB], [calc/getOperator] en [calc/showResult].

CONTROLLERS MET TOESTANDSVARIABELEN De dynamiek van de applicatie weerhoudt ons helaas om deze URI's ook als dusdanig publiek te maken: ze moeten per gebruik van de 'rekenmachine' namelijk andere resultaten tonen en dat kunnen ze bezwaarlijk doen als eenieder dezelfde paden zou gebruiken. We hebben dus dynamische URI's nodig die voor elk gebruik van de rekenmachine een eigen plaatsje in de URI request-ruimte voorziet. En één zo'n dynamische URI per 'in gebruik zijnde rekenmachine' zal dan volstaan: want op elk moment weet die zelf welke view er moet getoond worden om de volgende stap in de logische flow aan de gebruiker aan te bieden. Tenslotte dienen we als bootstrap nog een publieke (vast gekende) URI aan te maken om de eindgebruiker toe te laten de interactie te starten en dus een dergelijke 'rekenmachine' aan te maken. Zo krijgen we dat elke request op die laatste (bijvoorbeeld [calc.start-flow]) als gevolg heeft dat er een dynamische URI [continue-flow/276488373] beschikbaar wordt. Deze URI impliceert dan een dynamische 'resource' die door Cocoon ook als dusdanig expliciet wordt aangemaakt en beheerd door de FlowProcessor. De 'dynamische' resource is dan niet zozeer een kant-en-klare HTML file die op de harde schijf staat te wachten maar een Java-object. Het is de effectieve instantie van de use-case controller die zelf rechtstreeks de input te verwerken krijgt, zijn huidige 'VIEW' beslist en die via de magie van de sitemap laat produceren. In ons voorbeeld is dat bewuste Java Object niets anders dan de 'een in gebruik zijnde rekenmachine'.

De gehele afgelegde weg geeft de OO ontwikkelaar terug wat het web hem had ontnomen: controllers met eigen toestandsvariabelen die het de duur van de use case nodig heeft. Een stapje dichter bij een controller zoals we ze herkennen in een MVC architectuur:

```
<!– deze extra declaraties in Cocoon sturen
de FlowProcessor
  in het creëren en gebruiken van dit soort
  use-case-controllers –>

<map:flow language="apples"/>

<map:pipelines>
  <map:pipeline>

  <map:match pattern="continue-flow/*">
     <map:call continuation="{1}"/>
  </map:match>

  <map:match pattern="calc.start-flow">
     <map:call function="package.WebReken-
     Machine"/>
  </map:match>

  </map:pipeline>
</map:pipelines>
```

Marc Portier (e-mail: mpo@outerthought.org) is mede-oprichter van Outerthought, een technisch Java & XML kenniscentrum in België.