Enterprise

Applicaties maken eindgebruikers en opdrachtgevers pas echt blij als ze niet alleen doen wat ze moeten doen - functioneel - maar dat ook een beetje vlot doen. Performance is het begrip dat we gebruiken als het gaat over dat non-functionele aspect van 'een beetje vlot' de taak verrichten. En heel vaak als we spreken over tekortschietende performance wordt het ook niet heel veel concreter dan 'het is niet snel genoeg'.

Verlichte applicaties

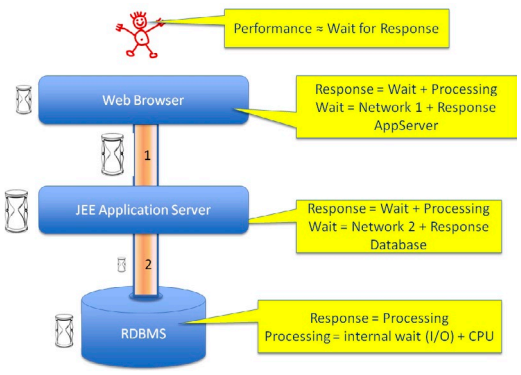
Over Java Enterprise Performance

Voor je het weet zijn de ontwikkelaars al zuchtend en steunend aan de slag gegaan om met profiling tools de bottlenecks in de applicatie op te sporen en met algoritmische verfijningen, geavanceerde tuning trucs en JVM parameter-optimalisatie de snelheid een beetje op te krikken.
In dit artikel nemen we iets meer afstand - om te kijken naar enterprise Java performance, over alle tiers van de architectuur. Maar vooral ook om allereerst grotendeels los van techniek te kijken naar de uitdaging en alledaagse oplossingen voor performance-problemen. En om vervolgens te zien hoe de alledaagse oplossingen vaak hun toepassing kennen in de techniek.
Dit artikel beoogt de lezer te inspireren en aan te zetten tot 'out of the box' denken waar het gaat om performance vraagstukken. En om te komen tot substantiële verbeteringen - niet enkele tientallen procenten, maar factoren en ordes van grootte verbetering. Het artikel wordt ondersteund door een presentatie met meer visuele illustraties van de concepten die worden besproken (zie: http://slidesha.re/iJSCEX).

Wat is performance?

Voordat we energie gaan steken in het verbeteren van performance zal toch allereerst duidelijk moeten zijn wat er onder performance wordt verstaan en wat de - op business doelen gebaseerde - eisen voor die performance zijn. Natuurlijk, het kan altijd beter. Maar is het zinvol om een bepaalde verbetering door te voeren? Na een migratie van een middleware infrastructuur was de response-tijd van een service dertig procent slechter dan voor de migratie. Hier was duidelijk een performanceprobleem geconstateerd en zes man overlegde een ochtend lang over hoe dat kon en wat er aan gedaan kon worden. Bij doorvragen bleek dat we spraken over een verslechtering van de response-tijd van dertig naar veertig ms. Voor een service die werd gebruikt voor raadpleging door een webapplicatie met responsetijden van anderhalve seconde of langer. Dat plaatste die dertig procent verslechtering uiteraard in een heel andere context. Performance moet beschouwd worden in samenhang met de totale keten die functionaliteit levert en met de eisen vanuit de business om ogenschijnlijke problemen op waarde te kunnen schatten. (Overigens bleek later dat het gebruik van een verkeerde versie van een databasedriver de dertig procent degradatie had veroorzaakt; door de nieuwste driver te installeren bereikten we responsetijden van pakweg 27 ms: een niet relevante performance-verbetering dus).

We hebben nu indirect al wel het begin van een definitie van het begrip performance: het gaat bij de beoordeling van performance veelal om de responsetijd die wordt ervaren vanaf het moment dat een actie in gang is gezet tot het moment dat een betekenisvolle reactie van het systeem wordt ontvangen.



Figuur 1.

Lucas Jellema
is Senor Consulant voor Java en SOA bij AMIS.

En nee - het tonen van een zandloper valt niet in de categorie betekenisvolle reacties.
In figuur 1 zien we een typische enterprise architectuur voor Java applicaties - een laag met database en andere enterprise resources, een middleware laag met applicatie server en eventueel enterprise service bus of andere SOA componenten en een client tier met web applicatie (in een browser) of WebService (afgenomen door een remote applicatie component).
De responsetijd die de gebruiker van de webapplicatie ervaart hebben we zojuist gedefinieerd als de tijd dat gebruiker wacht op een zinvol antwoord. Die wachttijd bestaat op haar beurt uit de tijd dat de browser bezig is om het request te versturen en later op basis van de response de pagina op te bouwen én de wachttijd die de browser zelf ervaart tijdens het wachten op de response van de server.
Die wachttijd in de browser is vervolgens opgebouwd uit de tijd die het netwerkverkeer vergt tussen browser en server en de tijd die de applicatieserver nodig heeft om tot een antwoord te komen. En zo verder tot in de uiterste lagen en componenten die bijdragen in de totstandbrenging van het antwoord op de vraag vanuit de browser door gebruiker geïnitieerd.

Het meten van de performance die er toe doet is het meten van de responsetijd die de gebruiker ervaart - afgezet tegen de functionele eis geformuleerd op basis van bedrijfsdoelstellingen. Het analyseren van die responsetijd omvat het meten van de netto verwerkingstijd en de wachttijd op iedere tier. Verbetering van de performance kan worden gerealiseerd op iedere tier die bijdraagt aan de totale responsetijd - en met name op de plek waar de grootste verwerkingstijd wordt gerealiseerd.
Dit artikel bespreekt acht zeer voor de handliggende manieren om de responsetijd te verkorten, daarmee de ervaren performance te verbeteren - en dus de tevredenheid van eindgebruikers en business eigenaren te vergroten. Voor elk van die acht ogenschijnlijke open deuren wordt vervolgens toegelicht hoe deze in termen van Java Enterprise architectuur een concrete uitwerking kennen.

De acht overwegingen

Een waardevolle stap in performance analyses is een reflectie op de volledige applicatie architectuur aan de hand van de volgende acht ogenschijnlijke 'open deur overwegingen.

De snelste manier om acties uit te voeren is door...
1. ze niet te doen;
2. ze niet vaker te doen dan strikt noodzakelijk is;
3. ze niet alleen te doen;
4. ze niet meteen te doen;
5. ze niet [alleen maar] op aanvraag te doen;
6. ze niet in te grote of te kleine stappen te doen;
7. ze op een nodeloos ingewikkelde manier te doen;
8. ze op een minder geschikte plaats te doen.

Dat is het. Beschouw aan de hand van deze invalshoeken en de performance gaat niet met 'small steps' maar met 'giant leaps' vooruit... Zo simpel is het natuurlijk ook weer niet. Maar deze invalshoeken zouden je wel serieus aan het denken kunnen zetten over de manier waarop en de plek waarin functionaliteit wordt geïmplementeerd en aangeboden. We zullen nu elk van deze suggesties in wat meer detail doornemen.

Niet doen

Is dat niet erg flauw? Dat zou je eerste reactie kunnen zijn. Maar er zit hier meer in dan je in eerste instantie zou denken. Allereerst is daar het bekende voorbeeld van de short circuit operator in Java, waarmee de volgende, ogenschijnlijk vrij redelijke evaluatie:

```
if ( expensiveEvaluation & someFlag) { ...}
```

enorm versneld kan worden door de volgorde van de condities te wijzigen en de short circuit operator && toe te passen die voorkomt dat de dure evaluatie nog plaatsvindt als de vlag de waarde false heeft:

```
if ( someFlag && expensiveEvaluation) { ...}
```

En de al bijna net zo bekende situatie met logging en dynamische logging-niveaus, waar

```
log.debug ( "Outcome step 2:"+resultOfExpensive-
Processing );
```

veel performance vriendelijker kan worden gecodeerd als:

```
if (log.doDebug) { log.debug ( "Outcome step 2:
 " + resultOfExpensiveProcessing ); }
```

Bovenstaande is waar, maar niet de essentie. In veel applicaties is vaak onnodige inefficiëntie aanwezig - die ervoor zorgt dat stappen worden gedaan die helemaal niet nodig zijn. Een typisch terrein is dat waar verzamelingen worden verwerkt. Een voorbeeld (ontleend aan een presentatie door Connor MacDonald):
Er is een methode beschikbaar ergens in onze applicatie onder de naam shopForItem(String item). Aan de methode wordt de naam van een product meegegeven en het betreffende product wordt aangekocht. Deze methode kan gebruikt worden voor het doen van de wekelijkse boodschappen:

```
getGroceries Item[] ( String[] shoppingList) {
  Item[] items = new Item[ shoppingList.length];
  for (int i=0; i < shoppingList.length; i++) {
     items[i] = shopForItem (shoppingList[i]);
  }
  return items;
}
```

Op het eerste gezicht ziet dit er volstrekt redelijk

In veel applicaties is vaak onnodige inefficiëntie aanwezig.

uit. Als we echter gaan kijken naar de implementatie van shopForItem valt ons op dat er ondanks de logisch correcte implementatie er alles bij elkaar toch sprake is van het uitvoeren van acties die niet uitgevoerd hoeven te worden:

```
shopForItem Item ( String itemName) {
  driveToShop;
  Item item = buyItemAtShop ( itemName);
  driveHomeFromShop;
  return item;
}
```

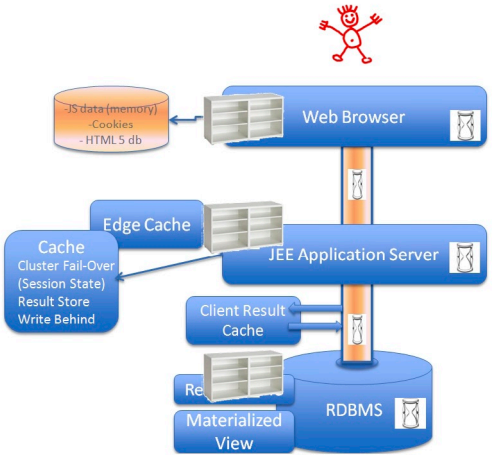
Deze methode staat visueel beschreven in figuur 2.

Door een methode herhaald aan te roepen voor ieder element in een verzameling, in plaats van de verzameling in zijn geheel als input mee te geven is inefficiëntie ontstaan op indrukwekkende schaal. En hoewel dit een gezocht voorbeeld is, kan dit soort verborgen performance-wurger eenvoudig een applicatie binnensluipen.

‘De capaciteit van snelwegen wordt vergroot met extra banen’.

Niet vaker doen dan noodzakelijk

Sommige acties moeten worden uitgevoerd om de gevraagde functionaliteit te leveren - maar niet noodzakelijkerwijs iedere keer dat bepaalde data wordt gevraagd. Net zoals we niet voor iedere boodschap opnieuw naar de winkel willen rijden en de winkel niet bevoorraad wordt per fles of blik maar per pallet, zouden we ook in onze applicaties gebruik moeten maken van een vorm van voorraadbeheer en hergebruik. In Enterprise Java architecturen betekent dat ondermeer dat een gegeven niet vaker dan nodig wordt opgehaald en afgeleid uit de diepste lagen van de applicatie. Op ondermeer dit principe is de NO SQL-beweging gebaseerd.
De term die bij deze benadering past is Cache (voor de gelegenheid uit te spreken als kas-je of eventueel kassie zie ook de figuur 3). Een cache bewaart op een tier gegevens die, vaak vanuit een diepere tier, zijn opgevraagd voor hergebruik bij een volgende behoefte. De aanname is dat uit het oogpunt van performance (lees responsetijd) en systeembelasting, opvragen vanuit de cache een gunstiger effect heeft dan opnieuw de gegevens op te bouwen vanuit de oorspronkelijke locatie.
Caches kunnen worden geïmplementeerd op iedere tier in een applicatie architectuur, van cookies, local storage (HTML 5) en JavaScript geheugen in de browser via een edge cache - vooral voor statische web elementen als CSS, images en JavaScript libraries - en een applicatie server-brede cache voorziening gebaseerd op static objecten en application scope state (binnen de JVM) en op een memory-grid voor cluster-brede caching over JVMs heen.



Figuur 3.

Ook de database, hoe raar dat misschien ook klinkt - heeft caching voorzieningen - bijvoorbeeld om resultaten van complexe queries te onthouden voor hergebruik of om (XML) documenten opgebouwd op basis van relationele data voor hergebruik te bewaren.
Uiteraard is de toepasbaarheid van caches afhankelijk van de periode gedurende welke gegevens in de cache als ‘actueel’ mogen worden beschouwd, de mate waarin hergebruik van gegevens voorkomt en eventueel de mechanismen die beschikbaar zijn om de cache actief up to date te houden. Overigens: ook als caching de responsetijd niet substantieel verbetert kan de resulterende verminderde belasting van onderliggende systemen de caching nog steeds zeer de moeite waard maken.

Niet alleen doen

Het vergroten van de verwerkingscapaciteit en daarmee de doorvoersnelheid van welke resource dan ook schuilt in parallellisatie. De capaciteit van snelwegen wordt vergroot met extra banen. Het afrekenvermogen van de supermarkt wordt met name vergroot door extra kassa’s te openen. Als de taken die onontkoombaar zijn kunnen worden uitgevoerd door meer dan één resource tegelijk neemt de tijd die nodig is om de taak af te ronden vrijwel altijd af. De voortzetting van Moore’s Law in het licht van de fysieke grenzen van processorsnelheden is hierop

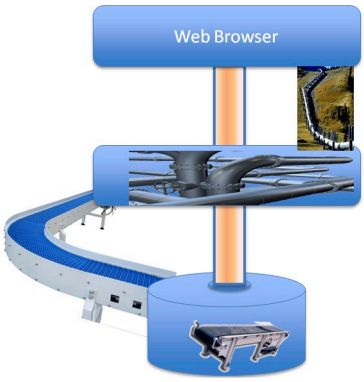
Figuur 2.

gestoeld: door multi-core processoren te produceren kan door parallelle verwerking de snelheid van de CPUs nog steeds toenemen.

Ook in Java Enterprise applicaties kan door uitvoering van taken te parallelliseren vaak een aanzienlijke verkorting van doorvoertijd (en daarmee responsetijd) worden behaald. Hiervoor is het natuurlijk wel noodzakelijk dat de betreffende acties in gelijktijdig uit te voeren brokken kunnen worden onderverdeeld en bovendien bij voorkeur I/O karakteristieken hebben die goed in parallelle CPU-threads kunnen worden uitgevoerd.
Parallelle verwerking is inherent in de CPU zelf, in de architectuur van de JVM en in de werking van moderne databases. Een vorm van parallelle verwerking krijg je dus gegeven door de infrastructuur. Dat helpt vooral met gelijktijdige requests van verschillende sessies - niet direct met het versnellen van een specifiek request. Daartoe kan het waardevol zijn om expliciete parallelle activiteiten te ontplooien - bijvoorbeeld door de concurrency op basis van Threadpools in de JVM of met behulp van background processen of jobs in de Application Server of de Database. Een aan populariteit en beschikbaarheid winnende optie is ook de inzet van een compute grid waar in vele nodes tegelijk aan complexe bewerkingen kan worden gewerkt.
Om het volgende niveau in parallelle verwerking te introduceren, maken we een korte uitstap naar een kinderpartijtje. Een feestje dat wordt afgesloten met pannenkoeken. Zie ook volgende figuur 4.

Figuur 4.

Er zijn in het pannenkoekenproces drie fasen te onderkennen: bereiden van beslag, bakken van de stapel pannenkoeken en vervolgens het eten van de pannenkoeken. Ons doel is zo snel mogelijk de kinderen met gevulde magen de deur uit te krijgen.
Met de gedachte aan parallelle verwerking nog vers in ons hoofd is het niet moeilijk om twee versnellingsmogelijkheden te zien in dit proces: laat de kinderen zoveel mogelijk gelijktijdig aan die pannenkoeken eten - het is onzin om er een van de stapel te pakken, die aan een kind uit te reiken en pas de volgende pannenkoek toe te wijzen als de vorige opgegeten is. En: het bakken van die hele stapel gaat aanzienlijk sneller als er meer dan één pan wordt gebruikt. Parallellisatie in twee tiers!
Maar wacht, er is meer. We kunnen gebruikmaken van pipelining om nog flinke verbetering te bereiken: we gaan niet alleen binnen de tiers parallel verwerken, we gaan dat ook tussen de tiers doen. In plaats van de hele stapel pannenkoeken te bakken - parallel weliswaar, maar toch de hele stapel - gaan we iedere pannenkoek die in de bak-tier gereed is direct doorschuiven ('pipelinen') naar de eet-tier. Als de laatste pannenkoek nog gebakken moet worden, staan de eerste verzadigde kinderen al buiten. Als dat geen versnelling is!
Het concept van pipelinen wordt in de wereld om ons heen veelvuldig toegepast, denk alleen maar aan de logistiek. In de wereld van Java Enterprise applicatie bestaat pipelinen zowel binnen tiers - bijvoorbeeld Database Pipes en Pipelined Table Functions in de Oracle Database - als tussen tiers.



Figuur 5.

Het gebruik van multiple threads en bijvoorbeeld message queues of shared collections op de middle tier en het gebruik van post-load AJAX requests zijn andere voorbeelden van pipelining.

Niet meteen doen

Onze definitie van performance heeft het specifiek over de tijd dat de gebruiker moet wachten tot de ontvangst van een betekenisvolle reactie op een actie die is gestart. Dat geeft ruimte voor interessante mogelijkheden om de ervaren performance enorm te verbeteren - zonder dat er een actie daadwerkelijk sneller wordt uitgevoerd. Als er maar eerder een betekenisvolle reactie komt, kan de beleving van performance vaak al enorm verbeteren.
Gebruikers die het systeem een opdracht geven - in plaats van een vraag stellen - zoals 'verstuur gegevens', 'print document' of 'stel maandrapportage samen' - zou je kunnen laten wachten tot de gevraagde actie is afgerond. Dat zou misschien wel zo eerlijk zijn. Maar zijn die gebruikers wel geïnteresseerd in de afronding van de betreffende actie - en helemaal op een synchrone (wachtende) manier? Misschien is de boodschap 'het systeem verwerkt

We kunnen gebruik maken van pipelining om flinke verbetering te bereiken.

Soms kan informatie meeliften op een andere request/response conversatie.

op dit moment uw verzoek' betekenisvol genoeg, zeker als die op een later moment wordt gevolgd door de asynchrone melding dat de taak is afgerond (of eventueel gefaald).

Een vergelijkbare gedachte, nog veel minder zichtbaar voor gebruikers, is de inzet van een grid oplossing voor database write-behind operaties: als de implementatie van het grid afdoende fail-over mechanismes kent, kan het als eerste kleine stap op de NoSQL-weg afdoende zijn om transacties in het grid te verwerken en pas op een later tijdstip, asynchroon te persisteren in de database. Deze manier van werken kan de database enorm ontlasten en kan applicaties een veel snellere response geven bij vastleggen van potentieel complexe transacties. Het gebruik in Java applicaties van Worker Threads die worden gestart om op de achtergrond een taak af te handelen en de toepassing van asynchrone jobs in de database zijn voorbeelden van 'doe het niet meteen'.

Niet [alleen maar] op aanvraag

Er zijn situaties waarin je heel graag zo snel mogelijk op de hoogte wilt zijn van een bepaalde ontwikkeling. "Weet u al of mijn bestelling er is" of "Is het al duidelijk of ik de rol wel of niet krijg" zijn typische zinnetjes van iemand die alsmaar contact opneemt om zijn of haar behoefte naar informatie te bevredigen. Het is niet voor niets dat de Hollywood regisseurs volgens de overlevering riepen: "Don't call us, we'll call you!" Om van het gezeur af te zijn - om de belasting van het onderliggende apparaat (de secretaresse) te verkleinen. Maar als je het vriendelijk uitlegt ook om de dienstverlening te vergroten en zelfs om er voor te zorgen dat de acteur zo snel mogelijk de gewenste informatie krijgt.

In Java Enterprise applicaties kan gebruik worden gemaakt van polling om actuele informatie te tonen. Met pollen van de database wordt de middle tier cache up to date gehouden en met pollen vanaf de client wordt de gebruiker van wijzigingen op de hoogte gehouden. Als we van polling zouden overgaan op een registratie-en-push mechanisme zouden we twee voordelen behalen: de (vaak forse) belasting van polling vervalt en de afnemers krijgen niet pas de gewenste update bij de eerstvolgende poll, maar via push vrijwel direct.
Ondermeer via de Servlet 3.0 specificatie, maar ook met frameworks als Dojo (op basis van Bajeux) is Server Push van middle tier naar browser beschikbaar. Messaging (JMS, Advanced Queuing) en event-mechanismen bieden push functionaliteit binnen applicatieserver en database terwijl met geavanceerde mechanismen als Query Result Change Notification (Oracle 10gR2 en hoger) en HTTP vanuit database triggers (ondermeer in Oracle Database sinds release 9i) is ook push tussen tiers mogelijk.

Niet te grote of te kleine stappen

De schermen in figuur 6 hebben een overeenkomst. Ze bevatten allemaal content die niet zichtbaar is op het moment dat de pagina wordt of net is geladen. De boom is initieel niet uitgeklapt. De inhoud van de niet-actieve tabs is onzichtbaar en pop-ups die in potentie geopend kunnen worden, zijn nu nog verborgen. In deze gevallen is het niet nodig dat de responsetijd die de gebruiker ervaart verlengd wordt vanwege het laden van content die niet nodig is voor het presenteren van de pagina. Dus in plaats van dat al die content in één grote brok wordt geladen, zou de pagina initieel alleen de benodigde componenten moeten laden. Direct nadat de pagina is geladen zou in AJAX interacties de aanvullende inhoud van de verborgen elementen in de achtergrond - zonder dat de gebruiker daar iets van merkt - kunnen worden opgevraagd. Als alternatief zou die content ook pas kunnen worden opgehaald als de gebruiker daar voor de eerste keer om vraagt; omdat het dan waarschijnlijk om betrekkelijk weinig elementen gaat zal de vertraging die gebruiker daarbij ervaart wellicht beperkt zijn.

Figuur 6.

Niet alles hoeft dus in één keer gedaan te worden. Data die in pagina's wordt aangeboden hoeft ook niet allemaal tegelijk worden opgeleverd vanuit de database.
Aan de andere kant is het soms gunstig om meerdere acties en berichten juist te combineren. Zoals we een klantenservice ook niet een antwoord vragen dat met één woord per telefoontje of email wordt gegeven, zo moeten we soms het aantal aanroepen tussen de tiers ook beperken. Door recordsets, master- en detailgegevens, alle plaatjes voor een webpagina en dergelijke te combineren in één bericht - XML, JSON, zip-file - kunnen we vaak besparen op het aantal aanroepen en de (netwerk) overhead die daarmee gepaard gaat.

Soms ook kan informatie meeliften op een andere request/response conversatie zodat niet een aparte roundtrip nodig is. Het plaatje van de space shuttle bovenop een Jumbo Jet illustreert dat mechanisme van piggy backing.

Niet te ingewikkeld doen

Ook dit klinkt weer als een kolossale open deur. Natuurlijk doe je dingen niet ingewikkelder dan nodig is...

Na de bijna-ondergang van J2EE en de narrow escape via JEE 5 rond 2005 zou je denken dat we dat nu toch wel geleerd hebben. Maar sommige vormen van complexiteit of omslachtigheid zijn soms op een heel impliciete, indirecte manier aanwezig. Een voorbeeld: er is niets ongebruikelijks aan een Java Web applicatie die met Hibernate data opvraagt uit een database en die instantieert als POJOs - Java Domain objecten. De applicatie zou die data vervolgens in XML kunnen aanleveren aan een mobiele app of AJAX web applicatie. Daartoe wordt een XML DOM document geïnstantieerd en op basis van de POJO gegevens gevuld. Dit document wordt vervolgens als string geserialiseerd in de Http Response. De architectuur-afspraken, de frameworks en de eigenschappen van Java en de JVM ondersteunen deze werkwijze. Maar als je weet dat die data die de uiteindelijke afnemer opvraagt op geen enkele andere manier in de Java applicatie een rol speelt en dat deze data in XML formaat al beschikbaar is in de database en als je in ogenschouw neemt dat object instantiatie (en uiteindelijke Garbage Collection) en de in-memory manipulatie van XML (DOM) documenten tot de zwaardere operaties hoort - dan zou je je moeten afvragen of deze ogenschijnlijke prima aanpak eigen wel de juiste is - en niet een nodeloos complexe.

Door afstand te nemen en de end-to-end stromen in de applicatie te beschouwen - niet alleen in het licht van wat gebruikelijk is en juist volgens de architectuur blueprint maar ook een keer vanuit het standpunt van wat mogelijk is - is het mogelijk dat pragmatischer alternatieven in beeld komen die op basis van aangetoonde voordelen ten aanzien van bijvoorbeeld performance de voorkeur verdienen. Voorbeelden zouden kunnen zijn: applicaties die vanuit de browser rechtstreeks met cloud services als Google Maps communiceren in plaats van dat aan de middle tier over te laten of mobiele clients die rechtstreeks - via firewall maar niet via middle tier - JSON services op de database aanroepen of transparante behandeling van XML documenten als strings in plaats van via DOM objecten.

Niet op een minder geschikte plaats doen

Bijna alle, zo niet alle, functionaliteit die wordt gevraagd in een Enterprise Java applicatie kan worden ontwikkeld door de Java/JEE ontwikkelaars in de JVM en de JEE applicatie server. Een database ontwikkelaar zou waarschijnlijk datzelfde constateren over implementatie binnen de database en een PHP ontwikkelaar weet er ook wel weg mee. Bijna alles kan tenslotte. Leuke uitdaging!

Echter, voordat de ontwikkeling ter hand wordt genomen is het waardevol om voor de verschillende onderdelen van de gevraagde functionaliteit te bepalen waar deze het best kan worden uitgevoerd – gelet op zaken als security, kosten van implementatie, out of the box beschikbare functionaliteit en schaalbaarheid en performance.

Zo lijkt in een enterprise architectuur de creatie van HTML elementen een taak die zo dicht mogelijk bij de client tier zou moeten liggen - eigenlijk in de client tier (DHTML op basis van JavaScript), al was het maar omdat de netwerkoverhead van volledige HTML documenten erg groot is. Zo ook zou je als Java ontwikkelaar niet moeten willen proberen de zoek-, sorteer- en aggregatie-functionaliteit van de enterprise database waar al tienduizenden manuren aan optimalisatie-inspanning in zit te verbeteren. Datzelfde geldt voor data integriteit als unieke sleutels en foreign keys - waarvan de implementatie op de middle tier uit oogpunt van performance een bedenkelijk idee zou zijn.

Daarnaast zijn er voor verschillende andere soorten functionaliteit specialistische componenten beschikbaar die - zeker als de betreffende functionaliteit een kritische factor is voor de gehele applicatie - serieus overwogen zouden moeten worden. Denk bijvoorbeeld aan Lucene voor geavanceerde tekst-indexering en search, complex event processoren voor real-time verwerking van signalen, matching engines als Elise voor verfijnde, fuzzy matching vraagstukken en engines voor scanning, document conversie en 3D imaging. Je eigen knutselwerk zal de functionaliteit, stabiliteit en performance van dit soort gespecialiseerde engines niet snel benaderen.

Conclusie

Het is pas zinvol over performance te spreken als duidelijk is wat er met die term wordt bedoeld. Het moet duidelijk zijn wat de eisen ten aanzien van performance zijn - uitgedrukt in objectief te meten waarden en gekoppeld aan bedrijfsdoelstellingen. Alvorens zich in detail-analyses en geavanceerde tuning-avonturen te storten is het aan te bevelen om van enige afstand de performance-eisen en -uitdagingen te wegen tegen de volledige applicatie architectuur. En daarbij out-of-the-box te denken over ondermeer de acht in dit artikel beschreven overwegingen, ontleend aan de niet-virtuele wereld om ons heen: doe het liefst niet, of in elk geval niet vaker dan nodig is en liefst ook niet in je eentje. Doe niet altijd 'klaar terwijl u wacht' en doe het soms proactief in plaats van op (herhaald) verzoek. Doe het niet ingewikkelder dan nodig is en ook niet op een manier of plek die daar niet zo geschikt voor is. Doe het niet in te grote stappen of juist in te kleine.

Met deze overwegingen toegepast over alle tiers van de applicatie architectuur kunnen vaak verrassende winsten worden behaald voor wat betreft de door eindgebruikers ervaren performance van een applicatie. Zonder dat het echt ingewikkeld wordt. «

Op het gebied van performance kun je verrassende winsten behalen.

Sander Hoogendoorn presenteert:

PRAGMATISCH IDENTIFICEREN, MODELLEREN, SCHATTEN EN TESTEN VAN SMART USE CASES

Nieuwe interactieve en praktijkgerichte workshop

5 en 6 oktober – Hotel Lapershoek Hilversum

Smart use cases is een snel in populariteit groeiende requirements techniek. Het is vrijwel de enige techniek die een volledige traceerbaarheid garandeert van de allereerste bedrijfsprocessen tot geteste code en zelfs applicatiebeheer. Zeker in agile, enterprise en service- en cloud georiënteerde projecten groeien smart use cases uit tot een de facto eenheid van werk, boven traditionele use cases en user stories. Deze nieuwe tweedaagse workshop is zeer interactief, bevat veel praktijkopdrachten en biedt het perfecte en complete overzicht in smart use cases.

Voor wie bestemd?
Deze workshop biedt een perfect inzicht in de populaire smart use case techniek. Iedere rol in software-ontwikkelprojecten en applicatiebeheer profiteert optimaal van de hier opgedane en direct in projecten toepasbare kennis, met name project managers, enterprise en software architecten, business en informatie-analisten, ontwikkelaars, testers en beheerders. Deze workshop mag u niet missen!

AGILE SOFTWARE DEVELOPMENT IN DE PRAKTIJK

Seminar met veel praktijkvoorbeelden en een interessante case

17 november 2011 – Hotel Lapershoek Hilversum

In dit seminar laat Sander Hoogendoorn, agile thought leader bij Capgemini, zien hoe agile software development bijdraagt aan het succesvol uitvoeren van software development projecten. Tijdens dit seminar bespreekt hij de nadelen van traditionele, lineaire methodieken, en gaat hij in op de karakteristieken van agile software development. Ook komen bekende misvattingen die rond agile de ronde doen aan bod. Het seminar geeft de overeenkomsten en verschillen tussen de belangrijkste agile methodieken, zoals Scrum, extreme programming, Smart, Lean en DSDM aan. Natuurlijk passeert ook een groot aantal best practices, tools en technieken uit de alledaagse praktijk de revue. Gastspreker Stefaan van Royen van Boondoggle toont de lessons learned bij de succesvolle invoering van agile software development in zijn bedrijf. Het seminar geeft de deelnemers een helder inzicht in de positieve bijdrage die agile software development levert aan projecten, en toont een reeks van concrete en pragmatische handvatten, technieken en best practices voor het implementeren van agile software development in uw organisatie.

Bestemd voor ú
De materie en de vele praktijkvoorbeelden in dit seminar hebben tot doel de kwaliteit en productiviteit van uw projecten te vergroten. Bent u betrokken bij softwaredevelopment? Bent u opdrachtgever, IT-manager, project-manager, architect, informatieanalist, ontwerper, ontwikkelaar of tester? Dan mág u dit seminar niet missen.

KIJK SNEL OP WWW.ARRAYSEMINARS.NL VOOR HET COMPLETE PROGRAMMA!

Array SEMINARS