Tuning von Oracle-Web-Applikationen im WebLogic Server 12c

Markus Klenke, Team GmbH

Beim Entwickeln der ersten prototypischen Applikationen mit JSF oder Oracle ADF wird häufig nicht direkt auf die Laufzeit und Performance der Applikation geschaut. Fehlende Optimierungen fallen meist erst bei größeren Applikationen, höheren Nutzerzahlen oder dem allgemeinen Produktivgang auf.

Es kommt nicht infrage, die Applikation neu aufzubauen. Das ist auch meist nicht notwendig, da einige Stellschrauben vorhanden sind, mit denen es häufig möglich ist, die notwendige Leistung aus der Anwendung und dem Server herauszukitzeln. Der Artikel zeigt verschiedene dieser Stellschrauben an der Applikation, dem Deployment und dem WebLogic Server 12*c* auf.

Das Erstellen von Geschäfts-Applikationen mit einem neuen Framework oder einer neuen Programmiersprache folgt meistens einem ähnlichen Schema: Schulungen erhalten, Prototypen bauen, Prototypen einstampfen, eigentliches Projekt beginnen.

Dieser Ansatz klingt nach einem guten Plan, er ist es in den meisten Fällen auch. Leider kann ein Prototyp bei Weitem nicht alle Anforderungen an das Echt-Projekt abbilden. In vielen Fällen wird mit Mock-Daten gearbeitet, höchstens mit einer oder zwei Personen getestet oder mal eben schnell die erste Lösung für ein Problem gewählt.

Beginnt dann die eigentliche Implementierung des Projekts, treten bei erstem Last-Verhalten die Schweißperlen auf die Stirn der Entwickler/Administratoren. Die Applikation ist auf einmal sehr langsam, bei vielen Benutzern bricht der Server zusammen, Benutzer sperren sich gegenseitig Datensätze etc. Woran kann das liegen und wo kann man nach diesen Problemfällen schauen?

Tendenziell gibt es mindestens drei Bereiche für potenzielle Applikations-Schwachpunkte: Die Applikation selbst, das Deployment und der Web-Server. Zusätzlich gibt es diverse Datenbank-Optimierungen, auf die dieser Artikel nicht eingehen wird.

Oracle ADF

Das Application Development Framework (ADF) ist das zentrale Java-Framework von Oracle zur Erstellung von Enterprise-Applikationen. Technisch gesehen bietet es diverse Interfaces, die beliebige Datenquellen über eine Abstraktionsschicht an eine Model-View-Controller-Muster-getriebene Web-Applikation binden. Dadurch ergeben sich innerhalb der ADF-Applikation mehrere Schichten, die für eine Performance-Analyse begutachtet werden müssen. *Abbildung 1* zeigt eine Übersicht über die verschiedenen Schichten. Dieser Artikel analysiert die Standard-Oracle-Implementierungen auf Performance-Optimierung.

ADF Business Components – ein Ebenbild der Datenbank

Die ADF Business Components sind das Persistenz-Framework der Daten für die ADF-Applikation. Aufgabe der Schicht ist es, Daten aus der Datenbank für Caching-Zwecke zu replizieren und diese den Oberflächen bereitzustellen. Um die Cache-Elemente zu erstellen und zu füllen, sind Queries an die Datenbank in den sogenannten „View-Objekten" gekapselt. Diese lassen sich ähnlich

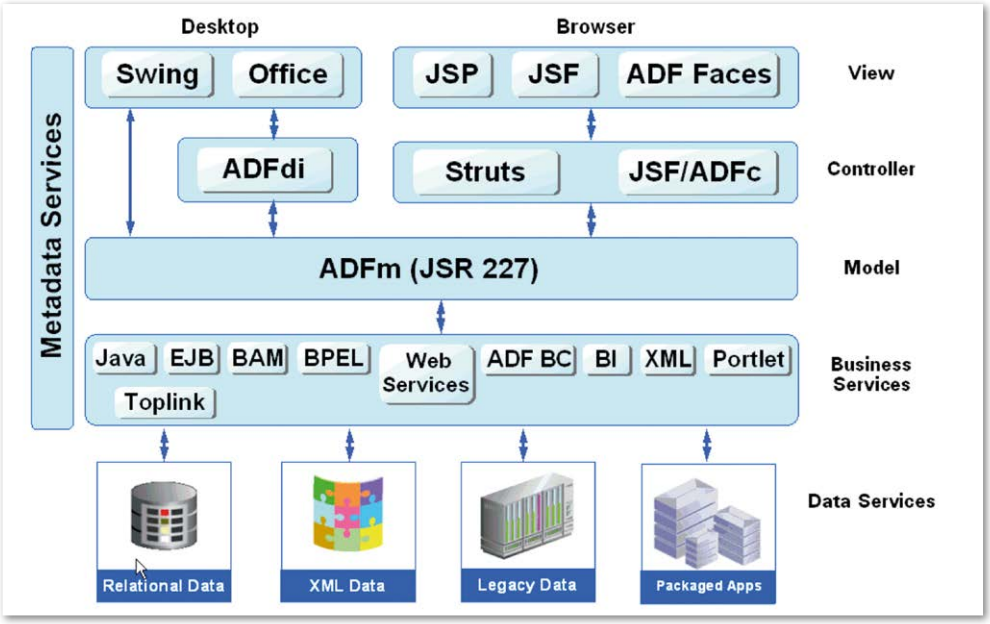

Abbildung 1: Schichten in der ADF-Entwicklung

wie allgemeine Datenbank-Queries optimieren. So können am View-Objekt selbst Optimizer Hints genauso benutzt werden wie bei der Datenbank, um der Datenbank bei der Abfrage direkt mitzugeben, worauf die Abfrage optimiert sein soll.

In vielen Fällen ist es absolut unnötig, alle Werte aus der Rückgabe-Menge sofort zu laden, da entweder eine Formular- oder eine Tabellen-Darstellung mit weniger als „n" Zeilen auf der Oberfläche vorliegt. Um nicht nur die Datenbank Queries, sondern auch das Instanziierungs-Verhalten der Objekte zu optimieren, bietet ADF an den View-Objekten weitere Einstellungen. So ist es zum Beispiel möglich, ein View-Objekt rein für die Erstellung eines Datensatzes zu benutzen. Die Query dient daher nur der Information, wo dieser Satz später in der Datenbank abgespeichert wird *(siehe Abbildung 2)*.

Application Modules sind die Services der Business Components, um die Datenabfragen und Rückgabewerte an die Außenwelt zu propagieren. Diese Objekte organisieren zusätzlich die Instanziierung von View-Objekten. Im Normalfall wird für jeden Anwendungsbenutzer mindestens ein Application Module erzeugt, um für die Nutzungszeit der Applikation eine Verbindung mit der Datenbank aufrechtzuerhalten.

Es gibt allerdings View-Objekte, bei denen es wenig Sinn ergibt, sie für jeden Benutzer neu zu instanziieren. Ein Beispiel ist ein View-Objekt, das etwa Übersetzungstexte aus der Datenbank liest. Diese Texte ändern sich für gewöhnlich nicht zwischen den Release-Zyklen. Um diese Objekte als Singleton zu erstellen, kann das Application Module als „Shared Application Module" deklariert sein. Dieser Modus sorgt dafür, dass nur ein einziges Modul dieses Typs erstellt und von allen Benutzern gleichermaßen verwendet wird. Man sollte dazu sagen, dass die Query des View-Objekts zur Laufzeit für jeden Benutzer individuell verändert werden kann, etwa für Sprach-Präferenzen. Das sorgt dafür, dass die abgefragten Elemente auch nur ein einziges Mal gecacht werden, was gerade bei Anwendungen mit vielen Benutzern einen enormen Speichervorteil mit sich bringt. Zusätzlich wird die Geschwindigkeit der Anwendung gesteigert, da bei schon gecachten Objekten die Query auf der Middleware ausgeführt wird und nicht auf der Datenbank.

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<deployment-plan …>
  <application-name>MyHRApp</application-name>
      <variable-definition>
          <variable>
              <name>Disable_Content_Compression</name>
              <value xsi:nil="false">false</value>
      </variable>
      </variable-definition>
      <module-override>
          <module-name>MyHRApp.war</module-name>
          <module-type>war</module-type>
          <module-descriptor external="false">
              <root-element>web-app</root-element>
              <uri>WEB-INF/web.xml</uri>
              <variable-assignment>
                  <name>Disable_Content_Compression</name>
                  <xpath>/web-app/context-param/[param-name="org.apache.
myfaces.trinidad.DISABLE_CONTENT_COMPRESSION"]/param-value</xpath>
              <operation>replace</operation>
          </variable-assignment>
          </module-descriptor>
      </module-override>
</deployment-plan>
```

Listing 1: Deployment-Plan zur Veränderung von „web.xml"-Parametern

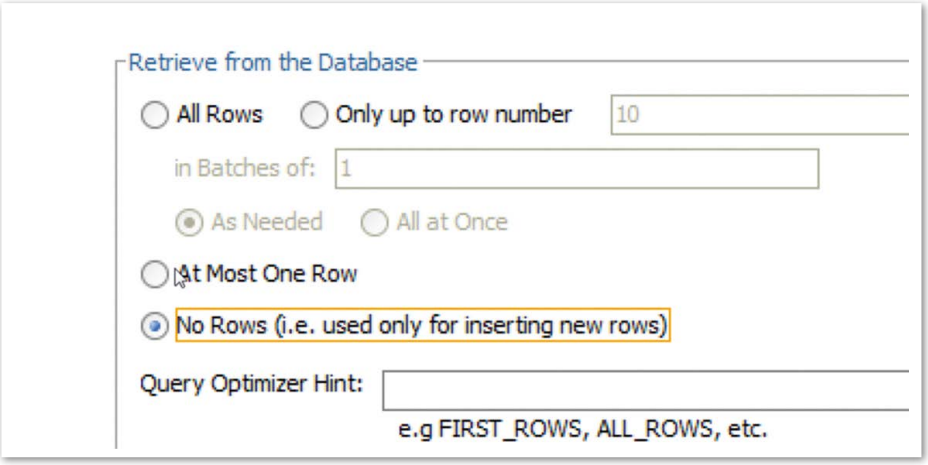

Abbildung 2: View-Objekt – Tuning-Einstellungen

Volle Kontrolle – Applikationsfluss über Managed Beans steuern

In Oracle ADF wird wie bei JSF der Kontext der Applikation in Managed Beans gespeichert. Dies sind Java-Klassen, die vom Framework gesteuert instanziiert werden und über einen bestimmten Zeitrahmen, den Scope der Bean, am Leben erhalten werden. Dadurch wird dem Entwickler einiges an Instanziierungs-Arbeit abgenommen. Managed Beans können in ADF in einem von sechs verschiedenen Scopes untergebracht sein, die unterschiedliche Lebenszyklen für die Bean bedeuten *(siehe Abbildung 3)*.

Ist eine Bean in einem kurzen Scope angesiedelt, wird sie entsprechend häufig aufgeräumt und neu instanziiert. Eine Bean im Application Scope würde hingegen ein einziges Mal beim Starten der Applikation instanziiert werden und dann für alle Benutzer der Applikation gültig sein.

Auf diesem Level der Applikation finden sich meistens zwei Extrema, die zu einer Performance-Verschlechterung oder einer System-Instabilität führen können. So kommt es immer mal vor, dass Beans mit einem hohen Kostenfaktor (lange Instanziierung, komplexe Berechnung etc.) in einem zu kleinen Scope liegen oder, anders herum gesagt, Java-Objekte mit einer großen Datenmenge für eine temporäre Berechnung im einem zu großen Scope.

Um die Auswirkung dieser beiden Fälle zu verdeutlichen, zwei Beispiele.

Erstes Beispiel: Eine Bean im Request-Scope (Lebenszyklus pro HTTP-Server-Anfrage) wird benutzt, um einen Wert aus der Datenbank zu lesen. Dieser wird zum Instanziierungs-Zeitpunkt über eine Datenbank-Funktion berechnet, die wiederum drei Sekunden für die Berechnung benötigt. Benutzer „A" lädt die Seite, die das Ergebnis der Funktion anzeigen soll. Er muss also entsprechend drei Sekunden warten, bis die Seite erscheint.

Benutzer „A" klickt auf der Seite auf einen Button, um eine weitere, aber sehr simple Funktion aufzurufen, die allerdings ebenfalls auf der Server-Seite ausgeführt wird. Statt einer kurzen Wartezeit muss Benutzer „A" nochmal drei Sekunden warten, da die Bean im Hintergrund neu instanziiert wird. Dies würde bei jeder weiteren Server-Anfrage wieder passieren. Wird die Bean in einen größeren Scope eingehängt (etwa „ViewScope", also Lebenszyklus einer Ansicht, die sich in unserem Beispiel nicht ändert), würde die erste Anfrage drei Sekunden benötigen, die anderen Anfragen könnten allerdings auf den bereits geladenen Wert zugreifen.

Zweites Beispiel: Eine Bean im Session Scope (Lebenszyklus pro User-Log-in) wird benutzt, um einen Summenwert über eine Liste von Artefakten zu generieren. Diese Liste besitzt 100.000 Einträge, die Daten-Akquise sowie die Summen-Berechnung sind sehr schnell. Ein Benutzer kommt auf die Seite, die das Summen-Ergebnis darstellt. Die Seite wird mehr oder weniger sofort angezeigt.

Für den End-Anwender läuft alles glatt und er beendet am Ende seines Arbeitstages die Session. Während der gesamten Zeit hält sich aber die Liste mit Einträgen im Speicher. Ist die Anzahl der Benutzer über den Tag auf eine bestimmte Anzahl angewachsen, ist die Gefahr, einen „Out of Memory"-Fehler zu erhalten, drastisch hoch, was zu einem Komplett-Ausfall des Servers führt. Wird die Bean hingegen in einem kleineren Scope gespeichert (wie „Request Scope"), muss zwar der Wert immer wieder neu berechnet werden, durch die geringe Komplexität fällt dies allerdings Performance-seitig nicht ins Gewicht; außerdem wird die Liste sehr zuverlässig vom Garbage Collector weggeräumt, was den Speicher-Lasten des Servers sehr guttut.

Von der Entwicklung zur Produktion – Deployment-Pläne

Ist die Applikation selbst so gut es geht optimiert, können Änderungen an Applikations-Parametern Performance-Gewinn einbringen. Im Entwicklungsmodus werden die Oberflächen-Komponenten von ADF meist unkomprimiert gerendert, was für Oberflächen-Tests auf Style-Klassen-Basis notwendig ist. Für den Produktiv-Einsatz ist dieser Modus allerdings vollkommen unnötig und verlangsamt nur die Renderzeit der Seite.

Muss man jetzt für den Entwicklungsmodus und den Produktivmodus zwei Applikationen simultan entwickeln? Glücklicherweise nicht, es besteht die Möglichkeit, einzelne XML-Strukturen der Applikation während des Deployments mithilfe eines Deployment-Plans an eine Umgebung anzupassen. So kann beispielsweise der Parameter zur Kompression der Oberflächen-Komponenten zu diesem Zeitpunkt auf „true" gestellt werden, damit auf der Produktion die Kompression der Klassennamen durchgeführt wird.

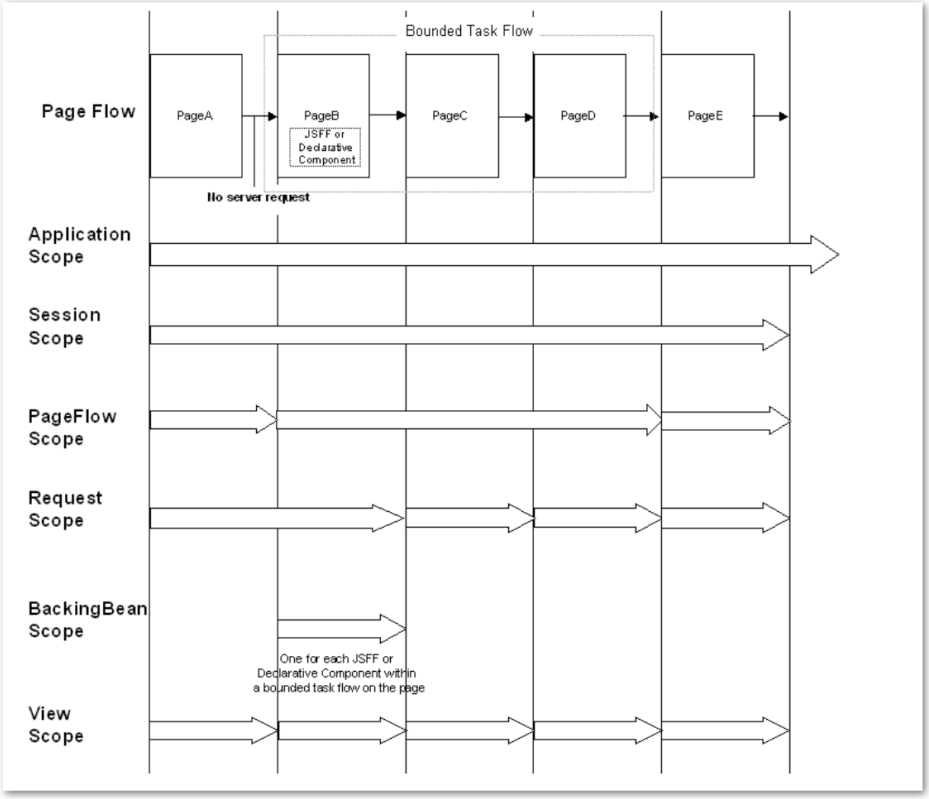

Abbildung 3: ADF Managed Bean Scopes

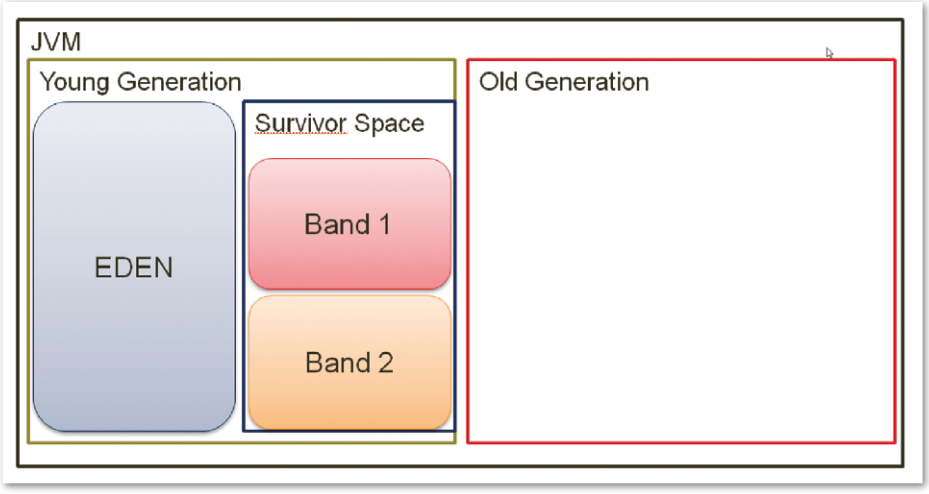

Abbildung 4: Visualisierung einer JVM

Es können insbesondere auch Parameter verändert werden, die das Applikationsverhalten alternieren. So lässt sich beispielsweise im Produktions-Modus eine Abfrage nicht mehr gegen eine Datenbank, sondern gegen einen parallel gestarteten Web-Service steuern, um Performance zu gewinnen. *Listing 1* zeigt ein Beispiel für einen Deployment-Plan, der die Kompression der Oberflächen-Komponenten aktiv schaltet:

JVM – Optimierung auf dem Server

Auf dem Server angekommen, gibt es immer noch diverse Möglichkeiten, die Applikations-Performance zu beeinflussen. Primär-Elemente dafür sind die Konfiguration der JVM des Servers und die Konfiguration der Datenquell-Pools. Dieser Artikel zeigt nur die JVM-Performance-Optimierungen.

Bei den Einstellungen der JVM scheiden sich die Geister. Daher sind die folgenden Einstellungen aus den Erfahrungen der ADF-Entwicklung entstanden und sollten nicht als Allheil-Werkzeug für die Entwicklung jeglicher Web-Applikationen gesehen werden. Wie beschrieben, muss die JVM großteilig mit Objekten aus den Managed Beans interagieren. Daher sollten die Speicher-Konfigurationsparameter auf genau dieses Szenario optimiert sein.

Generell benötigen Starts der Applikation mit einem noch nicht aktiven Benutzer einen relativ großen Anteil an „bootstrap"-Speicher. Bei einer Neuanmeldung wird also der größte Teil des gesamten vom Benutzer benötigten Speichers sofort benötigt. Es ist daher nicht sinnvoll, die initiale Heap-Size deutlich kleiner zu halten als die maximale. Als Best Practice für ADF-Applikationen hat sich ergeben, die JVM-Parameter „Xms" (initialer Heap Size) = „Xmx" (maximaler Heap Size) zu setzen, um das Nachladen von Speicherblöcken auf dem Arbeitsspeicher des Servers zu vermeiden.

Ein weiteres wichtiges Thema im Bereich „Speicherverwaltung" ist die Konfiguration der Garbage Collection (GC). Für ADF sind insbesondere die Parameter „XX:NewRatio" und „XX:SurvivorRatio" interessant, da sich diese mit den jeweiligen Teilbereichen des permanenten Heap auseinandersetzen, der „Young Generation" und der „Old Generation" *(siehe Abbildung 4)*.

```
Desired survivor size 255379422 bytes, new threshold 55 (max 55)
- age     1:  203476204 bytes,      203476204 total
- age     2:     4037572 bytes,     207513776 total
- age     3:    39200462 bytes,      246714238 total
```

Listing 2: Ausgabe der JVM Garbage Collection

Die Old Generation enthält Objekte, die entweder initial zu groß für die Young Generation gewesen sind, oder Objekte, die eine bestimmte Anzahl an GCs überlebt haben. Während einer GC ist der Young-Generation-Anteil der spannendere: Neu instanziierte Objekte sind generell in „Eden" gespeichert, überlebende Objekte in einem der beiden „Bands", sodass das jeweils andere leer bleibt. Es entstehen also ein Content Band und ein Copy Band.

Sind Objekte bei der GC nicht mehr erforderlich, werden sie einfach gelöscht und Speicher freigegeben. Sind in Eden und dem Content Band noch Referenzen aktiv, so werden diese Objekte als Survivor markiert und in dem Copy Band gespeichert. Sollte das Copy Band noch nicht gefüllt sein, tauschen Copy Band und Content Band die Rollen für die nächste GC. Ist das Band voll, werden die Objekte mit der längsten Lebenszeit in die Old Generation verschoben. Das Löschen von Objekten in der Old Generation ist aufgrund des fehlenden Copy Bands komplexer, da bei jedem Aufräumen zusätzlich eine Defragmentierung des Bereichs erfolgt.

„XX:NewRatio" stellt die Größe der Young Generation in Relation zur Old Generation dar. Ein Wert von „drei" ergibt, dass die Old Generation drei Mal so groß ist wie die Young Generation. Das Minimum ist eins, da die Old Generation mindestens so groß sein muss wie die Young Generation.

Die SurvivorRatio gibt die Relation von Eden zum Survivor-Bereich an. In ADF werden viele Java-Objekte einem Benutzer zur Session zugeordnet. Sie haben also im Allgemeinen einen hohen Lebenszyklus, während andere nur für einen Request benutzt und direkt danach wieder aufgeräumt werden sollten. Es ergibt also Sinn, die Session-Objekte in den Langzeit-Speicher zu verschieben. Daher sollte man, um die GC-Zyklen so kurz wie möglich zu halten, die Young Generation der JVM eher kleiner einkalkulieren.

Um eine Übersicht über die GC Zyklen zu bekommen, sollte in der Entwicklungsumgebung der Parameter „XX:+PrintTenuringDistribution" benutzt werden. Dieser sorgt dafür, dass bei jeder GC Informationen darüber in das Log geschrieben werden, wie lange sich Elemente schon in der JVM befinden und wie viele GC-Durchläufe sie schon überlebt haben. *Listing 2* gibt ein Beispiel:

Die Ausgabe zeigt, dass ca. 200 MB an Objekten einen GC-Zyklus überlebt haben, ca. 4 MB zwei Zyklen und ungefähr 40 MB schon drei Zyklen. Die Survivor Size ist 256 MB, daher ist es im nächsten GC-Zyklus nicht notwendig, Daten in die Old

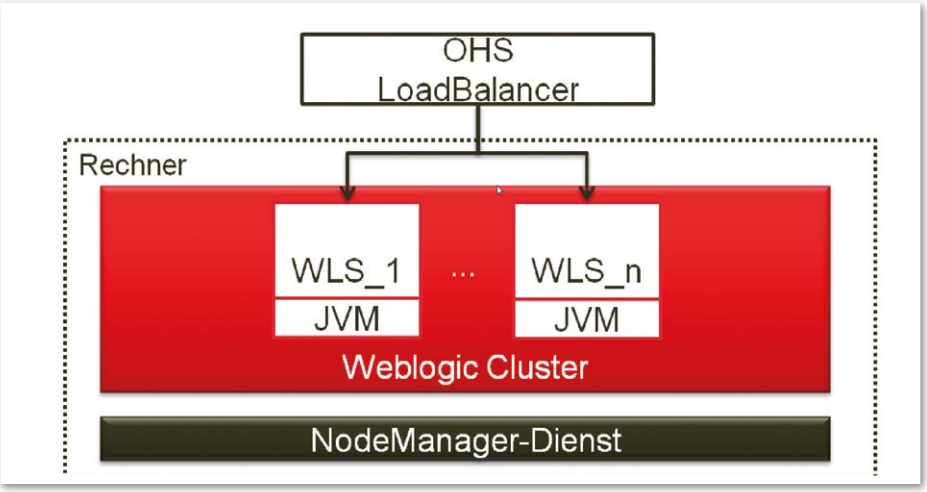

Abbildung 5: WLS Cluster für Performance-Optimierung

Generation zu verschieben. Aus der Erfahrung haben sich folgende JVM-Parameter für ADF-Web-Applikationen bewährt:

- Xmx = Xms
- XX:NewRatio 3
- XX:SurvivorRatio 6
- XX:+UseISM (allokiert 4 MB Blöcke statt 8 KB Blöcken)
- XX:+AggressiveHeap (dynamische Vergrößerung der maxHeap; nicht verwenden mit Xmx-Parametern)

Wenn nichts mehr geht – WebLogic Cluster

Es kann Fälle geben, in denen weder an der Applikation noch am Deployment oder an den Server-Einstellungen etwas optimiert werden kann. Eventuell ist die Last der User auf einem Server und auf einer Applikation einfach zu hoch. Spätestens jetzt muss man Fragen nach der Replikation stellen und Oracle bietet mit dem Oracle WebLogic Cluster einen mächtigen Baukasten für diverse Herausforderungen.

Um die Performance auf einem System mithilfe einer einzelnen Hardware zu verbessern, kann die Applikation durch eine simple Lastverteilung über einen Oracle-HTTP-Server auf einen Cluster von Managed-WebLogic-Servern repliziert werden *(siehe Abbildung 5)*.

Fazit

Durch einfache Konfiguration über die WebLogic-Konsole oder WLST kann man in kürzester Zeit eine (der User entsprechenden) Anzahl an JVMs auf einem Rechner zusammenschließen. Allerdings muss man fairerweise sagen, dass das auch seinen Preis hat (WebLogic Enterprise Edition).

Das Thema „Performance-Optimierung" ist ein spannendes Thema, weil in jedem Projekt individuelle Anforderungen an Applikationen, Deployments und Server gestellt werden. Der Artikel hat einen kleinen Einblick in die Vielfältigkeit der Optionen zur Performance-Optimierung gegeben.

Markus Klenke
mke@team-pb.de