Orientation in Objects

Auswirkung großer Komponentenbäume

auf die Performance von JSF Implementierungen

) Schulung)

) Beratung)

) Entwicklung)

AUTOR

Thomas Asel
Orientation in Objects GmbH

Veröffentlicht am: 31.12.2012

EINE VERGLEICHSSTUDIE DER IMPLEMENTIERUNGEN ORACLE MOJARRA UND APACHE MYFACES

Der komponentenorientierte Ansatz von JSF bedingt eine Verwaltung des Zustandes der verwendeten Komponenten. Das wirkt sich auf das Laufzeitverhalten JSF-basierter Anwendungen aus. Dieser Artikel untersucht anhand von Vergleichsmessungen zwischen den beiden Implementierungen Oracle Mojarra und Apache MyFaces die Auswirkungen unterschiedlich großer Komponentenbäume auf das Laufzeitverhalten.

) Artikel)

Orientation in Objects GmbH

Weinheimer Str. 68
D-68309 Mannheim

Tel. +49 (0) 6 21 - 7 18 39 - 0
Fax +49 (0) 6 21 - 7 18 39 - 50

www.oio.de info@oio.de

EINLEITUNG

MOTIVATION

Steve Souders beschreibt in seinem Standard-Werk "*High Performance Web Sites*" die grundlegenden Konzepte um Web-Anwendungen, unabhängig von der Implementierungstechnologie, performant zu gestalten. Während diese Regeln ebenfalls für JSF-Anwendungen gelten, existieren darüber hinaus technologiespezifische Einflussfaktoren auf die Performance von JSF-Anwendungen.

Oracle Mojarra [1] stellt die Referenzimplementierung von JSR 314 [2], der Spezifikation von Java ServerFaces 2.0 / 2.1, dar. Apache MyFaces [3] ist eine offene Implementierung des gleichen Standards. Beide Implementierungen verwenden die im Rahmen der JSF-Spezifikation definierte API und sind prinzipiell austauschbar. Neben diesen beiden Implementierungen existieren keine nennenswerten weiteren eigenständigen Implementierungen der JSF Spezifikation, daher kommt beiden Implementierungen eine besondere Rolle im Rahmen der Java EE 6 (definiert durch JSR 316 [4]) zu: Java EE 6 konforme Application-Server müssen laut Spezifikation eine JSF-Implementierung verwenden, d.h. eine der beiden verfügbaren Implementierungen wird sehr wahrscheinlich im Auslieferzustand des Servers enthalten sein. In der Vergangenheit wurde sich, je nach Hersteller, für die Referenzimplementierung Mojarra oder die freie Implementierung der Apache Foundation entschieden. Mittlerweile gehören oftmals sogar beide Implementierungen zum Auslieferungsumfang der Application-Servern, sodass die zu verwendende Implementierung durch Konfiguration des Application-Servers bestimmt werden kann. Da Servlet-Container keine JSF-Implementierung beinhalten, muss die Entscheidung für eine JSF-Implementerung explizit vom Applikationsentwickler getroffen werden.

Um eine fundierte Entscheidungsgrundlage für eine der verfügbaren Implementierungen bieten zu können, wurde eine vergleichende Studie erstellt. Dieser Artikel erläutert die Vergleichskriterien und stellt die ermittelten Messergebnisse vor. Durch die gewonnenen Daten soll ein direkter Vergleich der beiden Implementierungen ermöglicht werden.

FRAGESTELLUNG

Untersucht werden soll die Frage welchen Einfluss die Größe des JSF-Komponentenbaumes auf die Performance einer JSF-Anwendung hat. Dazu wird die Abarbeitungdauer des JSF-Lifecycles jeweils für einen Request einer fest definierten Abfolge gemessen. Die Messung wird mit unterschiedlich großen Komponentenbäumen wiederholt. Die Ergebnisse werden hinsichtlich des Laufzeitverhaltens der jeweiligen Implementierung interpretiert. Zusätzlich lässt sich das Verhalten der beiden Implementierungen für jeweils gleich große Komponentenbäume direkt vergleichen.

Die Messgrößen werden an einer Beispielapplikation ermittelt. Die Applikation soll jeweils mit den beiden JSF Implementierungen Oracle Mojarra und Apache MyFaces realisiert werden. Die Konfiguration der Anwendung soll dabei identisch sein, um die Implementierungen direkt vergleichen zu können.

Die JSF-Spezifikation definiert zwei unterschiedliche Methoden der Zustandsverwaltung:

1. **Server-Side-State-Saving**

   Stellt die Standard-Methode der Zustandsverwaltung dar. Beim Server-Side-State-Saving wird eine Repräsentation des Komponentenbaums einer Seite dauerhaft im Speicher auf Server-Seite behalten.

2. **Client-Side-State-Saving**

   Alternativ kann der Zustand des Komponentenbaumes auf der Client-Seite gehalten werden. In diesem Fall wird der Zustand serialisiert und mit dem Response an den Client ausgeliefert. Beim Submit des Formulars sendet der Client den serialisierten Zustand zurück zum Server, wo dieser deserialisert und ggf. verändert wird.

Die zu messenden Größen werden bei beiden Implementierungen jeweils mit Server-Side als auch mit Client-Side-State-Saving mit unterschiedlich großen Komponentenbäumen ermittelt.

Die Anzahl n der Komponenten im Komponentenbaum ist variabel, mit n {10, 100, 250, 500, 1000, 1500, 2000, 2500, 5000}.

EXKURS: KOMPONENTENANZAHL

Wie viele Komponenten sind für eine JSF-Anwendung typisch? Diese Frage kann natürlich nicht allgemein beantwortet werden. Die Erfahrungswerte in vom Autor betreuten Projekten belaufen sich auf mehrere hundert Komponenten auf einer Seite bis hin zu ca. 2000 Komponenten bei komplexen Views.

Als ursächlich für die hohe Anzahl an Komponenten werden folgende Punkte angesehen:

- **Nicht gerenderte Komponenten im Baum**

  In den betrachteten Projekten findet sich eine Vielzahl von Komponenten, die mittels Rendered-Attribut nur bei Bedarf gerendert werden. Diese Komponenten durchlaufen den Lifecycle immer vollständig, unabhängig von der Belegung des Rendered-Attributes und haben somit auch Einfluss auf das Laufzeitverhalten der Anwendung.

- **Inflationärer Gebrauch von Composite-Components**

  Composite-Components sind eine mit JSF 2.0 eingeführte Neuerung, welche die Entwicklung von JSF-basierten Anwendungen enorm vereinfacht hat. Anwendungsfallspezifische Komponenten lassen sich auf diese Weise sehr einfach rein deklarativ entwickeln. Dies führt in vielen Fällen jedoch dazu, dass Composite-Components als Universalwerkzeug für die Schaffung wiederverwendbarer Einheiten betrachtet werden: So werden Composite-Components verwendet um zustandslose HTML-Schnipsel zu kapseln obwohl dafür benutzerdefinierte Tags oder Facelet-Decorators besser geeignet wären. Composite-Components stellen ebenfalls Elemente des Komponentenbaumes dar, sodass in der Praxis häufig unbeabsichtigt Templating- und Design-Aspekte dafür sorgen, dass Markup den Komponentenbaum belastet. Unter [5] findet sich eine Beschreibung für standardkonforme Alternativen zu Composite-Components.

- **Komplexer Aufbau der Views**

  Häufig bedingen die fachlichen Anforderungen übermäßig komplexe Views. Hier ist vor allem die Verwendung von Tabbed Panes ("Reiter") zu nennen: Obwohl nur jeweils der Inhalt eines Tabs gleichzeitig sichtbar ist, enthält der Komponentenbaum auch die Komponenten der nicht sichtbaren Tabs. Dadurch ergeben sich auch bei Seiten, die augenscheinlich nur aus wenigen Komponenten bestehen mitunter sehr große Komponentenbäume.

Die JSF-Implementierungen bringen keine Werkzeuge mit, mit denen sich leicht die Anzahl der Komponenten im Komponentenbaum darstellen lässt. Unter [6] wird beschrieben, wie mit vertretbarem Aufwand ein solches Werkzeug entwickelt werden kann.

ERWARTUNG

Die unterschiedlichen Funktionsweisen beider Methoden der Zustandsverwaltung implizieren bereits Auswirkungen auf die Abarbeitungsdauer des Lifecycles. Es wird erwartet, dass diese mit zunehmender Größe des Komponentenbaumes ebenfalls zunimmt. Da bei Client-Side-State-Saving der Zustand bei eingehenden Requests aus diesen extrahiert, deserialisert und später wieder serialisiert werden muss, wird erwartet, dass die Lifecycle Duration bei Server-Side-State-Saving kürzer ausfällt.

Da sowohl Mojarra als auch MyFaces Implementierungen die gleiche Spezifikation darstellt, wird erwartet, dass beide Implementierungen sich in ihrem Laufzeitverhalten sehr ähnlich verhalten.

Die erwarteten Ergebnisse werden in Abschnitt 4 mit den gemessenen Ergebnissen verglichen.

AUFBAU

DATENERHEBUNG

Um den Effekt unterschiedlich großer Komponentenbäume auf das Laufzeitverhalten einer JSF-Anwendung beobachten zu können, muss ein Testszenario entworfen werden, dass folgende Anforderungen erfüllt:

Gleiche Codebasis der Testanwendung: Es muss eine einfache Web-Anwendung entwickelt werden, deren Codebasis für alle Messungen so weit wie möglich gleich ist. Die Konfiguration der Anwendung soll stabil bleiben. Konfigurationsänderungen umfassen entweder den Austausch der JSF-Implementierung oder die Anzahl der darzustellenden Komponenten oder aber die Methode der Zustandsverwaltung (Server- oder Client-Seitig). Es wird jeweils nur ein Konfigurationsparameter verändert.

Austauschbarkeit der JSF-Implementierung: Die verwendete Implementierung (Oracle Mojarra oder Apache MyFaces) muss ausgetauscht werden können, ohne dass Änderungen an der Testanwendung vorgenommen werden müssen. Somit kann abweichendes Verhalten direkt auf die jeweils verwendete Implementierung zurückgeführt werden.

Variable Anzahl der Komponenten im JSF-Komponentenbaum: Die Testanwendung muss mit unterschiedlich großen Komponentenbäumen untersucht werden können. Die Anzahl der Komponenten im jeweiligen Komponentenbaum muss dabei die einzige Größe sein, die zwischen den Messungen verändert wurde. So wird ein direkter Rückschluss auf die Auswirkung von unterschiedlich großen Komponentenbäumen auf das Laufzeitverhalten möglich.

Unterschiedliche Methode der Zustandsverwaltung: Die Testanwendung muss bei gleicher Implementierung und gleichbleibender Anzahl von Komponenten im Baum jeweils mit den Zustandsverwaltungsmethoden Server- und Client-Side-State-Saving untersucht werden.

Vergleichbarkeit der Messung: Die erhobenen Daten müssen in jedem Fall auf gleiche Weise gewonnen werden. Es gilt insbesondere implementierungsspezifische Abweichungen zu vermeiden. Konkret bedeutet dies: Es darf bei Messungen nicht auf Methoden zurückgegriffen werden, die nur für eine der zu untersuchenden Implementierungen verfügbar sind. Dies ist insbesondere bei der Konfiguration des Profilers zu berücksichtigen.

Das nachfolgende Klassendiagramm stellt die Realisierung der Lifecycle-Phasen in beiden JSF-Implementierungen dar. Die von JSR 314 [2] definierte API enthält die von den Implementierungen zu verwendende abstrakte Oberklasse Lifecycle. Die Lifecycle-Phasen selbst gehören nicht zum Umfang der standardisierten API, sondern sind implementierungsspezifisch. Da die Funktionsweise vergleichbar ist, dienen die jeweiligen execute()-Methoden der Lifecycle-Phasen implementierenden Klassen als Einstiegspunkte für den Profiler.

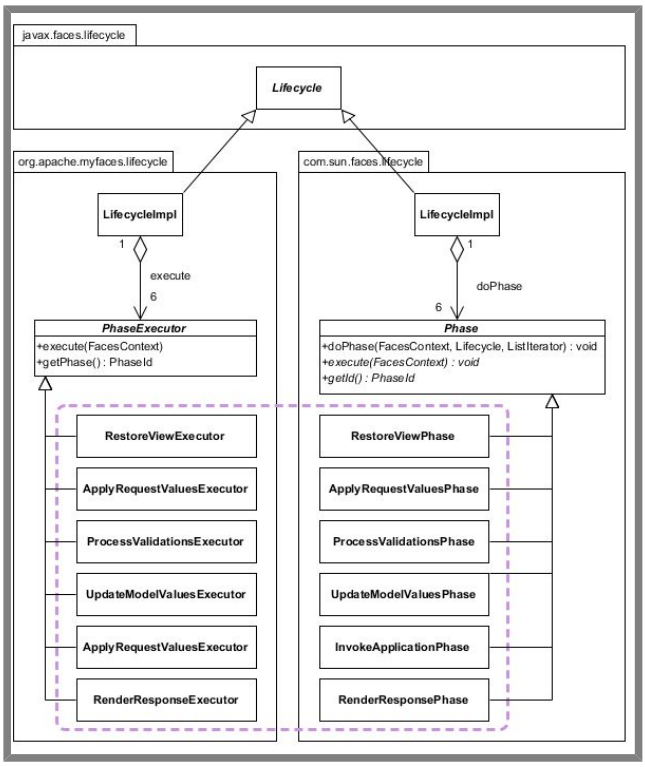

Abbildung 1: Realisierung der Lifecycle-Phasen in beiden JSF-Implementierungen

Vergleichbares Deployment-Szenario: Infrastruktur und Ausgangssituation für alle Messungen müssen identisch sein. Das bedeutet, dass die Testanwendungen auf gleicher Hardware und im gleichen Servlet-Container deployt werden müssen. Im Servlet-Container darf immer nur die gerade zu testende Anwendung deployt sein. Der Container muss vor dem Deployment gesäubert werden, d.h. alle bisherigen Deployments müssen aufgehoben sowie alle aktiven Sessions beendet werden.

ERHOBENE DATEN

Die Abarbeitungsdauer wird errechnet aus der Summe der Laufzeiten der Phasen 1-6 und stellt somit ein aggregiertes Datum dar (Lifecycle Duration).

TEST-SZENARIO

Es wird eine Anwendung entwickelt, die eine variable Anzahl von JSF-Komponenten auf einer Seite darstellen soll. Die Anwendung soll folgendes Szenario abdecken können:

Für ein beliebiges n:

1. **Initialer Request**:

   Darstellung einer Seite mit n Eingabekomponenten vom Typ `<h:inputText />`

2. **Submit der Eingaben**:

   Validierung der Eingaben

Darstellung der Eingaben auf einer Seite mit n Ausgabekomponenten vom Typ `<h:outputText />`

3. **Re-Submit der Eingaben (Verhalten bei Verwendung des Back-Buttons im Browser)**

   Wiederholung von 2. um Verhalten der Anwendung bei Benutzung des Back-Buttons durch den Benutzer zu untersuchen.

4. **Submit der Seite mit der Wert-Ausgabe**

   Erneute Darstellung der Seite mit n Eingabekomponenten

VARIATIONSPUNKTE

Um einen direkten Vergleich zwischen Implementierungen sowie einzelnen Konfigurationseinstellungen zu ermöglichen, sind folgende Variationspunkte vorgesehen:

1. Unterschiedliche Implementierungen: Oracle Mojarra und Apache MyFaces
2. Zustandsverwaltung: Client-Side vs. Server-Side
3. Anzahl der Komponenten im Baum: n {10, 100, 250, 250, 500, 1000, 1500, 2000, 2500, 5000}

Es werden deploybare Artefakte für alle Permutationen aus Implementierung, Zustandsverwaltung und Anzahl der Komponenten erzeugt. Es wird ein Servlet-Container gestartet, ein Artefakt in den Servlet-Container deployed und die Messungen daran durchgeführt. Nach der Messung wird das Deployment entfernt und der Container heruntergefahren.

TECHNISCHE UMSETZUNG

WEB-ANWENDUNG

Die Testanwendung besteht aus zwei Facelet-Views, die je nach gewähltem n eine variable Anzahl von Komponenten beinhalten. Die Erzeugung der Komponenten wird durch Verwendung des Iterations-Tag `<c:forEach>` der JSTL erreicht.

Die erste View der Anwendung, input.xhtml, enthält n Eingabekomponenten vom Typ `<h:inputText>`. Die zweite View detail.xhtml enthält n Ausgabekomponenten.

DURCHFÜHRUNG

Um aussagekräftige Ergebnisse zu erhalten werden die Messungen mehrfach wiederholt. Die Ergebnisse werden als Mittelwert der vorgenommenen Messungen ermittelt. Für n {10, 100, 250, 500, 1000, 1500, 200} werden 100 Wiederholungen durchgeführt. Für n = 2500 werden 75 und für n = 5000 werden 20 Wiederholungen durchgeführt. Die Reduzierung der Wiederholung bei hoher Anzahl von Komponenten erfolgt aus der rein praktischen Notwendigkeit die Laufzeit der Messungen zu begrenzen. Auch bei 20 Messungen ist eine ausreichende Glättung von Abweichungen gegeben.

WERKZEUGE

APACHE JMETER

Die Requests von Clients werden mit Apache JMeter simuliert. Beispielhafte Testskripte sind im Anhang enthalten.

EJ-TECHNOLOGIES JPROFILER

Die Ausführungszeiten innerhalb des JSF-Lifecycle werden mit dem Profiling Tool JProfiler [7] durchgeführt. JProfiler erlaubt die Entwicklung sogenannter Custom Probes. Custom Probes erlauben unter anderem das festhalten definierter Ereignisse bei Ausführung bestimmter Codestellen. Auf diese Weise können die einzelnen Phasen des JSF-Lifecycles vermessen werden ohne Overhead zu erzeugen. Dazu erlaubt JProfiler das Einschleusen von Code auf Ebene der Java Virtuellen Maschine. Der eingeschleuste Code kann bei Eintritt und Verlassen einer Methode, sowie bei auftretenden Exceptions abgearbeitet werden.

Damit die Messungen vergleichbar bleiben, müssen die festzuhaltenden Ereignisse bei beiden vermessenen Implementierungen an den gleichen Codestellen erzeugt und abgegriffen werden.

Da die JSF-Spezifikation lediglich die zu verwendenden Schnittstellen definiert, muss hier in den Implementierungen nach geeigneten Stellen gesucht werden, um den Code für die Erhebung von Messdaten einzuschleusen.

Beide Implementierungen kapseln die Logik der einzelnen Lifecycle-Phasen in einer jeweiligen Klasse. Die jeweiligen Phasen werden mit Aufruf einer Methode dieser Klassen durchlaufen

Für Oracle Mojarra sind dies folgende Methoden:

```
com.sun.faces.lifecycle.
RestoreViewPhase.execute(javax.faces.context.FacesContext)
com.sun.faces.lifecycle.
ApplyRequestValuesPhase.execute(javax.faces.context.FacesContext)
com.sun.faces.lifecycle.
ProcessValidationsPhase.execute(javax.faces.context.FacesContext)
com.sun.faces.lifecycle.
UpdateModelValuesPhase.execute(javax.faces.context.FacesContext)
com.sun.faces.lifecycle.
InvokeApplicationPhase.execute(javax.faces.context.FacesContext)
com.sun.faces.lifecycle.
RenderResponsePhase.execute(javax.faces.context.FacesContext)
```

Für Apache MyFaces sind dies die Methoden:

```
org.apache.myfaces.lifecycle.
ApplyRequestValuesExecutor.execute(javax.faces.context.FacesContext)
org.apache.myfaces.lifecycle.
InvokeApplicationExecutor.execute(javax.faces.context.FacesContext)
org.apache.myfaces.lifecycle.
ProcessValidationsExecutor.execute(javax.faces.context.FacesContext)
org.apache.myfaces.lifecycle.
RenderResponseExecutor.execute(javax.faces.context.FacesContext)
org.apache.myfaces.lifecycle.
RestoreViewExecutor.execute(javax.faces.context.FacesContext)
org.apache.myfaces.lifecycle.
UpdateModelValuesExecutor.execute(javax.faces.context.FacesContext)
```

Die Codebasis der für die Messung verwendeten Custom Probe findet sich im Anhang.

INFRASTRUKTUR

Servlet-Container und Messwerkzeuge werden auf unterschiedlichen Rechnern ausgeführt. Als Ausführungsplattform wird Java 7u9 (64 Bit) verwendet, als Servlet Container kommt Apache Tomcat 7.0.27 auf folgender Hardware zum Einsatz:

- **CPU**: Intel Core i5-2400 (3,10GHz, 6MB)
- **RAM**: 4GB 1333MHz DDR3 Non-ECC
- **HDD**: 250GB S-ATA (7.200 1/min)
- **OS**: Windows 7 (x64)

JProfiler wird auf einem separaten Rechner ausgeführt, das Profiling erfolgt über die Remote-Schnittstelle. Auch JMeter wird auf einer spearaten Maschine ausgeführt um die Ergebnisse nicht zu verfälschen.

ERGEBNISSE - ABARBEITUNGSDAUER DES JSF-LIFECYCLES

Entgegen der Erwartung, dass beide Implementierungen sich in ihrem Laufzeitverhalten ähneln, zeigt sich bei der Abarbeitungsdauer des Lifecycles ein deutlicher Unterschied:

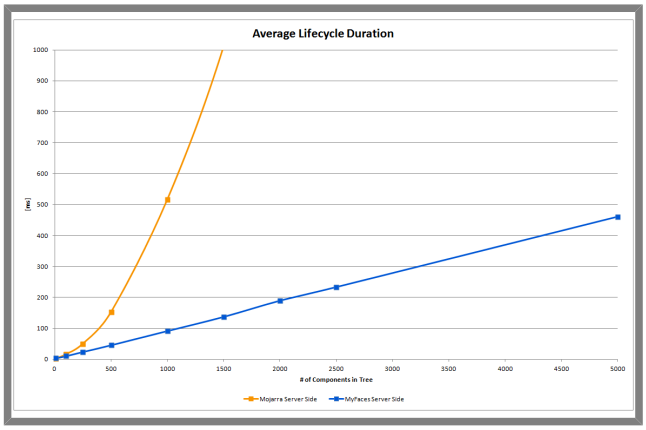

Abbildung 2: Diagramm 1: Abarbeitungsdauer des JSF-Lifecycles

Bereits ab 500 Komponenten im Baum benötigt Mojarra ein Mehrfaches der Abarbeitungszeit von MyFaces. Während die Durchlaufzeit des Lifecycles bei Apache MyFaces linear mit der Anzahl der Komponenten im Baum ansteigt, verhält sich dieses Wachstum bei Oracle Mojarra deutlich schlechter. Für Seiten mit großen Komponentenbäumen muss aus Performance-Sicht daher der Einsatz von MyFaces empfohlen werden.

Vergleicht man das Verhalten der Implementierung jeweils bezüglich State-Saving ergibt sich folgendes Bild:

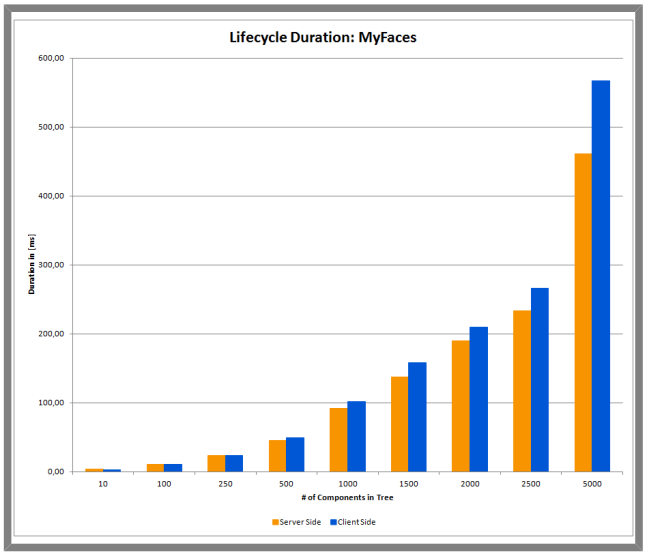

Abbildung 3: Diagramm 2: Server- vs. Client-Side-State-Saving bei MyFaces

Die Abarbeitungsdauer des Lifecycle liegt bei MyFaces erwartungsgemäß bei Server-Side-State-Saving unter der für Client-Side-State-Saving. Dies ist bedingt durch die Tatsache, dass der Anwendungszustand bei Client-Side-State-Saving zusätzlich serialisiert und obfuskiert oder verschlüsselt in den Response eingearbeitet werden muss. Bei eingehenden Requests muss der Zustand zunächst wieder de-obfuskiert und –serialisiert werden.

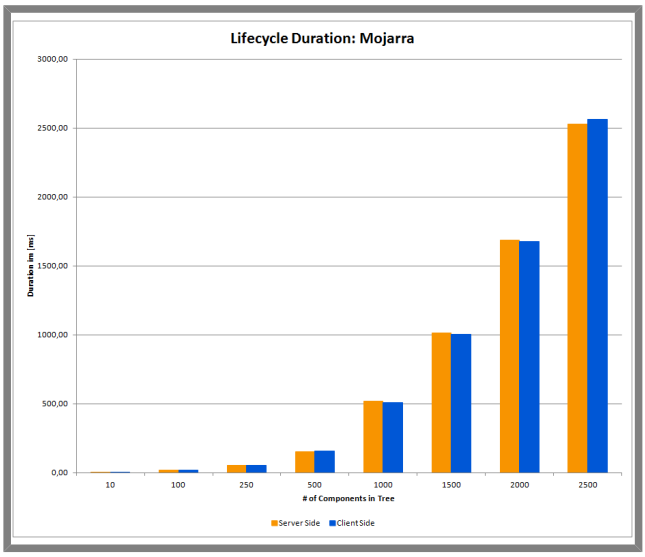

Abbildung 4: Diagramm 3: Server- vs. Client-Side-State-Saving bei Mojarra

Interessanterweise verhält sich Mojarra bei beiden State-Saving Methoden annähernd gleich, jedoch deutlich unperformanter als MyFaces. Es gilt zu beachten, dass die Skalierung der Zeitachse für Mojarra aus Darstellungsgründen um Faktor 5 größer gewählt ist. Hinsichtlich Abarbeitungsdauer des Lifecycles verspricht ein Wechsel der State-Saving-Methode also nur geringe Performance-Gewinne bzw. -Einbußen.

MyFaces benötigt bei 2500 Komponenten im Baum und Server-Side-State-Saving eine mittlere Abarbeitungsdauer von 255,57 ms. Bei gleichen Parametern benötigt die Referezimplementierung Mojarra mit 2527,94 ms annähernd 10x so lange wie die freie Implementierung.

ERGEBNIS-INTERPRETATION

Anhand der gesammelten Daten lassen sich folgende Aussagen zum Laufzeitverhalten von JSF Anwendungen treffen:

1. Die Anzahl der verwendeten JSF-Komponenten auf einer Seite hat direkte Auswirkungen auf die Performance der Anwendung. Dies äußert sich hinsichtlich der Abarbeitungsdauer des Lifecycle.
2. Apache MyFaces durchläuft den JSF-Lifecycle erheblich schneller als Oracle Mojarra. Insbesondere Anwendungen mit großen Komponentenbäumen würden somit von einem Umstieg von Mojarra auf MyFaces profitieren.
3. Apache MyFaces bietet unter Berücksichtigung aller untersuchten Aspekte ein gleichwertiges oder besseres Verhalten als die Referenzimplementierung Oracle Mojarra.
4. Durch den beobachteten linearen Anstieg der Abarbeitungsdauer des JSF-Lifecycles ermöglicht MyFaces eine verlässlichere Skalierung der zu erwartenden Laufzeiten.

Die gemessenen Ergebnisse zeigen, dass die Anzahl von Komponenten innerhalb einer JSF-View bedeutenden Einfluss auf das Laufzeitverhalten haben kann. Anwendungsentwicklern wird daher dringend empfohlen darauf zu achten den Komponentenbaum nicht zu überfrachten.

Aus Sicht des Autors wird in der Praxis am häufigsten durch inflationären Gebrauch von Composit Components verstoßen: Das mit JSF 2.0 eingeführte Komposit-Komponentenmodell wird häufig verwendet, wenn innerhalb des UI wiederverwendbare Einheiten geschaffen werden sollen. So werden Composit-Components häufig zur Kapselung von reinem Markup oder für Templating-Zwecke eingesetzt, obwohl JSF 2 mit benutzerdefinierten Tags und Facelet-Dekoratoren dafür bessere Werkezuge bereitstellt.

Hinsichtlich der Wahl der Implementierung lässt sich eine klare Empfehlung zugunsten Apache MyFaces aussprechen. Neben den im Rahmen dieses Artikels besprochenen Performance-Vorteilen verfügt MyFaces über eine äußerst aktive Entwicklergemeinde. Das Projekt wird von mehreren in Teilprojekten entwickelten Bibliotheken vervollständigt. MyFaces hat den zur JSF 2 Spezifikation gehörenden Tests des TCK (Technology Compatibility Kit) bestanden und gilt somit als vollständig konform zu JSR 314 [8].

AUSBLICK

Im Rahmen dieses Artikels wurde die Abarbeitungsdauer des Lifecycle als einziges Kriterium für die Performance-Messung betrachtet. Da weitere Performance-kritische Größen existieren, ist dies ist noch nicht ausreichend um eine allgemeingültige Aussage hinsichtlich Performance aufzustellen.

Weiterführende Untersuchungen könnten für Web-Anwendungen spezifische Größen wie Response-Größen und -Zeiten sowie die Session-Auslastung analysieren. Weiterhin könnten Messgrößen auf Ebene der Java Virtuellen Maschine wie Anzahl und Dauer der Garbe-Collector-Läufe sowie der Verlauf der Heap-Size aufschlussreich sein. Dem interessierten Leser wurde mit diesem Artikel ein kompakter Einstieg in die Thematik präsentiert.

REFERENZEN

- [1] Oracle Mojarra JavaServer Faces
  http://javaserverfaces.java.net (http://javaserverfaces.java.net)
- [2] JSR 314 - Java ServerFaces 2.0 Specification
  http://www.jcp.org/en/jsr/detail?id=314 (http://www.jcp.org/en/jsr/detail?id=314)
- [3] Apache MyFaces
  http://myfaces.apache.org/ (http://myfaces.apache.org)
- [4] JSR 316 - JavaTM Platform, Enterprise Edition 6 (Java EE 6) Specification
  http://www.jcp.org/en/jsr/detail?id=316 (http://www.jcp.org/en/jsr/detail?id=316)
- [5] Custom Tags as an Alternative to Composite Components in JSF
  http://blog.oio.de/2012/05/23/custom-tags-as-an-alternative-to-composite-components-in-jsf
  (http://blog.oio.de/2012/05/23/custom-tags-as-an-alternative-to-composite-components-in-jsf)
- [6] How to get the total Number of Components in the JSF Component Tree
  http://blog.oio.de/2013/01/11/how-to-get-the-total-number-of-components-in-the-jsf-component-tree/
  (http://blog.oio.de/2013/01/11/how-to-get-the-total-number-of-components-in-the-jsf-component-tree/)
- [7] ej-Technologies JProfiler
  http://www.ej-technologies.com/products/jprofiler/overview.html (http://www.ej-technologies.com/products/jprofiler/overview.html)
- [8] Offizielles Announcement zum MyFaces Core v2.0.0 Release
  http://markmail.org/message/vhywkfftgyuga3dp?q=tck+list:org%2Eapache%2Emyfaces%2Eannounce+order:date-forward
  (http://markmail.org/message/vhywkfftgyuga3dp?q=tck+list:org%2Eapache%2Emyfaces%2Eannounce+order:date-forward)
- [9] Messwerte, Test-Anwendung, JMeter und JProfiler-Konfiguration
  JSF-Performance.zip (http://www.oio.de/public/java/jsf/JSF-Performance.zip)