Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Diplomarbeit von Manfred Meyer

Modellierung und Implementierung eines Message Flows in einer EJB-Umgebung

Diplom Arbeit DA-2001.21
April 2001 bis August 2001
Betreuer: Erik Wilde
Professor: Bernhard Plattner

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Vorwort

Das World Wide Web Consortium (W3C) bemüht sich schon seit langem, eine einheitliche Beschreibungssprache für Webinhalte durchzusetzen. Das Ursprüngliche Format HTML (Hyper Text Markup Language) ist im Browserkrieg zwischen Netscape und Microsoft stark beschädigt worden. Zwar wurde versucht dies durch immer wieder neue Versionen zu reparieren - mit mässigem Erfolg. Dann wurden Cascading Style Sheets (CSS) eingeführt um die Darstellung vom Browser unabhängig zu machen, in dem Inhalt und Layout von einander getrennt werden. Extensible Markup Language (XML) mit der Extensible Style Language (XSL) ist eine vielversprechende Fortsetzung davon. Doch XML geht weit darüber hinaus und gerät immer mehr in den Bereich des *Content Management*. XML wird als das Datenformat der Zukunft propagiert und sehr viele IT Firmen mit Rang und Namen haben dessen Unterstützung in ihren Produkten angekündigt. Doch es gibt noch weitere interessante Ideen, wie die Information im Internet in Zukunft angeordnet werden könnten. Unter den Begriffen *Open Hypermedia System* und *Topic Maps* ist man daran, das schon sehr erfolgreiche Konzept der Hyperlinks zu revolutionieren.

An diesem Punkt wird auch im Projekt XLinkbase gearbeitet. Erik Wilde hat hier schon einiges geleistet, wie zum Beispiel *Wilde's WWW glossary* (siehe [3]) eindrücklich demonstriert. Etliche Studenten haben schon Semester oder Diplomarbeiten der XLinkbase gewidmet, wie auch Yves Langisch, dessen Arbeit [1] Grundstein für meine Diplomarbeit ist. Mit Yves' und meinen Studien bewegen wir uns in das sehr interessante Gebiet der modernen Webserver-Technologien, die auf SUNs neuer Java 2 Enterprise Edition Plattform aufbauen. Darin enthalten sind Konzepte für Servlets, Enterprise Java Beans, Java Messaging Service (JMS), die Java Transaction Architecture (JTA) und vieles mehr.

Rückblickend, war es für mich eine wunderbare Erfahrung, mich mit all den beschriebenen Technologien zu beschäftigen und das in einer Arbeit die zu einem Gebiet zählt, in dem zur Zeit sehr aktiv geforscht und entwickelt wird. Ich hatte während der ganzen Zeit immer sehr grosse Freiheit und die Möglichkeit mitzubestimmen, in welche Richtung das Projekt gehen soll. Ich möchte an dieser Stelle meinem Betreuer Erik Wilde recht herzlich für die angenehme Zusammenarbeit in den letzten vier Monaten danken. Ebenfalls vielen Dank gebührt der Firma Bea Systems, für die freundliche Gewährung einer verlängerten Testlizenz des Weblogic Servers, die mir meine Arbeit sehr erleichtert hat.

Manfred Meyer
Zürich, im August 2001

Institut für
Technische Informatik und
Kommunikationsnetze

Dr. Erik Wilde, Institut für Technische Informatik und Kommunikationsnetze, ETH-Zentrum, CH - 8092 Zürich
fon +41-1-6325132; mobile/sms +41-79-6431851; fax +41-1-6321035; mailto:net.dret@dret.net; http://dret.net/

Zürich, 17.4.01

Herrn
Manfred Meyer

Aufgabenstellung Diplomarbeit SS 2001 für Manfred Meyer

Im XLinkbase-Projekt entstehen ein Modell und eine Implementierung einer Datenbank, die spezifisch dafür ausgelegt ist, beliebige Ressourcen-Informationen zu sammeln (meist Referenzen auf Internet-basierte Ressourcen in Form von HTTP- oder FTP-URIs) und in komplexe inhaltliche Zusammenhänge zu bringen. Das zugrundeliegende Modell verwendet an vielen Stellen XML-basierte Technologien, um die verschiedenen Komponenten des Gesamtsystems miteinander zu verbinden. Die Grundarchitektur des Systems ist ein Client/Server-Modell, in dem der Client entweder ein normaler HTML-basierter Web-Browser, ein XML/XLink-Browser der nächsten Generation, oder auch ein spezialisierter Client sein kann, der spezifisch für den XLinkbase-Server programmiert ist und die Informationen in der XLinkbase in anschaulicher Weise graphisch modelliert. Der XLinkbase Server ist ein durchgängig XML-basiertes System, das über HTTP mit dem Client kommuniziert und die Requests und Responses über eine flexibel konfigurierbare Menge von XSLT Style Sheets verarbeitet. Die eigentliche Speicherkomponente ist ebenfalls austauschbar, im Normalfall wird dies jedoch entweder ein relationales Datenbanksystem oder eine XML-Lösung sein.

Der existierende XLinkbase-Server Prototyp ist momentan als auf Enterprise JavaBeans (EJB) basierende Architektur realisiert. Die EJB Komponenten kommunizieren miteinander mittels XML Messages. Diese Messages werden verwendet, um Requests, die der Server erhält und verarbeiten muss, von den jeweils zuständigen EJBs verarbeiten zu lassen. Im Rahmen der Diplomarbeit sollen die folgenden Probleme auf der Serverseite konzeptionell betrachtet und implementiert werden:

- Das Format für die zwischen den EJBs ausgetauschten XML Messages (Requests und Responses) muss festgelegt werden. Dies muss in enger Abstimmung mit der Diplomarbeit passieren, die sich mit der Query-Sprache für die XLinkbase beschäftigt.
- Ein weiterer wichtiger Aspekt ist der der Dienstbeschreibung der EJBs, die so ausgelegt sein muss, dass bei der Konfiguration einer XLinkbase die vorhandenen EJBs beschrieben werden können, um so von der Routing-Komponente entsprechend berücksichtigt werden zu können.
- Ausgehend von den konfigurierten EJBs und einem konkreten Request muss ein Algorithmus für das Berechnen einer Route gefunden werden, der für jeden ankommenden Request die optimale Verarbeitungsweise im Server berechnet. Diese Route ist ein Graph, der konditional den Weg eines Requests durch verschiedene EJBs beschreibt.
- Ein Konzept für das Routing von Messages zwischen den verschiedenen EJBs muss erstellt werden. Dieses Routing in abhängig von vom berechneten Graphen und wird wahrscheinlich auf dem Java Messaging Service (JMS) beruhen.

Ziel der Diplomarbeit ist es, ein Konzept und eine prototypische Implementierung zu erstellen, die es ermöglicht, mit den erarbeiteten Konzepten praktische Tests durchzuführen. Die wichtigsten Aspekte bei diesen Tests sind Robustheit, Skalierbarkeit, Flexibiliät und Erweiterbarkeit der Lösung.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einführung

Hintergrund dieser Diplomarbeit ist die *XLinkbase*. Um die folgenden Kapitel zu verstehen, sollte man mindestens eine Ahnung haben worum es dabei geht. Eine geeignete Einführung in dieses Projekt wäre zum Beispiel das Paper [2]. Es wird aber hier nicht direkt an der XLinkbase gearbeitet, sondern es geht um die Anbindung dieser an einen Webserver. Warum gerade die im folgenden beschriebene Architektur gewählt wurde und was die Vorteile davon sind, kann unter [1] nachgelesen werden. Darin findet sich auch einen ausführlichen Vergleich mit anderen möglichen Technologien wie etwas CORBA, COM/DCOM inklusive Microsofts *.NET* Plattform.

Die Grundidee war also bereits gegeben - eine Schar von *Message Driven Beans* bearbeiten in einem *J2EE*-Konformen Application Server von aussen über das Internet ankommende Requests. Wie diese Beans sich aber die Arbeit aufteilen, um Abfragen oder Manipulationen an der XLinkbase zu vollziehen, wie sie miteinander kommunizieren und vorallem wie sie genau programmiert werden, das war noch völlig offen.

Da also bei diesem Projekt nicht schon zum vorherein klar war, wie das System am Schluss aussehen soll, habe ich mich entschlossen für die Organisation der Implementierung ein Iterationsmodell anzuwenden. Aufsetzend auf den Vorschlägen in der Arbeit [1], versuchte ich in jeder Iteration neu zu entscheiden, wohin die Entwicklung gehen soll und welche Features als nächstes eingebaut werden sollten.

Die Kapitel 2, 3 und 4 stellen den chronologischen Fortgang in dieser Arbeit dar. Es wird jeweils beschrieben was die Zielsetzung der Iteration war, gefolgt von Konzepten und Implementationsdetails zu den einzelnen Themen. Nach jeder Iteration habe ich dann nochmals zurückgeschaut, einige Erkenntnisse notiert und Ideen geäussert wie es weiter gehen könnte.

Bevor eine neue Iteration gestartet wurde, habe ich mit meinem Betreuer Erik Wilde über den Fortgang der Projekts diskutiert und entschieden welche Features als nächstes Eingebaut werden sollten, welche realistisch sind und welche noch zurückgestellt werden müssen. Dabei sind wir zur Überzeugung gelangt, dass dieses im Entstehen begriffene Produkt zu allgemeineren Zwecken geeignet ist als nur für die Konstruktion eines XLinkbase Servers. Nach langem Sinnieren zur Positionierung dieser Arbeit im Software Markt, haben wir uns entschieden ein offizielles Paper zu verfassen und an eine geeignete Konferenz zur Veröffentlichung einzusenden. Das Paper trägt den Titel *Routed Message Driven Beans: A new Abstraction for using EJBs* und ist in Kapitel 5 in diese Dokumentation eingebunden. Dieses Paper ist auch als ein zentraler Bestandteil für die Dokumentation meiner Diplomarbeit zu betrachten. Darin enthalten ist ein einleitender Abschnitt über die XLinkbase und die Motivation für diese Arbeit, Vergleiche und Abgrenzungen zu anderen Produkten sowie Ideen für zukünftige Ergänzungen. Schliesslich wird noch weiter auf gewonnene Erkenntnisse eingegangen, inklusive Testergebnisse und Performance-Messungen.

Ein sehr wichtiges Kapitel für Leute die sich intensiv mit dieser Arbeit beschäftigen, sie weiterführen oder neue RMDBs implementieren wollen, ist der Anhang A. Neben einer

Auflistung was sich auf der beiliegenden CD-ROM befindet, sind auch Anleitungen zur Installation und Konfiguration der Application Server, sowie Hinweise zur Programmierung und Erzeugung von RMDBs. Der JBoss Application Server und insbesondere der Bea Weblogic Server sind äusserst komplex. Ohne Starthilfe braucht es sehr viel Zeit bis man sich darin zurecht findet. In der Bibliographie sind zudem Links aufgeführt zu wichtigen Dokumentationen der beiden Server.

Im Anhang B befindet sich die API Dokumentation von sämtlichen RMDBs und deren Hilfsklassen. Diese wurde mit dem *javadoc*-Tool erstellt und ist auch im HTML-Format auf der CD verfügbar.

Kapitel 2

Iteration 1

2.1 Use Case

Der Systemaufbau in diesem ersten Prototyp ist nahezu identisch zu dem in meiner Vorgängerarbeit. Ein Servlet (Kicker) nimmt http-Requests entgegen, erzeugt daraus eine JMS-Message, die dann in einem Pool von Message-Driven-Beans verarbeitet wird. Der erste Empfänger der Message heisst Controller. Dieser bekommt in meinem System eine besondere Aufgabe, denn er ist für die Initialisierung der Routingmechanismus verantwortlich. Für die weitere Verarbeitung des Requests, soll die Message an einen Logger weitergereicht werden, welcher vorerst nichts weiteres tut als eine Nachricht auf die Konsole auszugeben und die Message an den RequestHandlerßu leiten. Der RequestHandlerïst das Bean welches die eigentliche Behandlung der Anfrage übernimmt. Der Request wird darin geparst, die verlangten Daten beschafft und an den Kicker zurück gesendet.

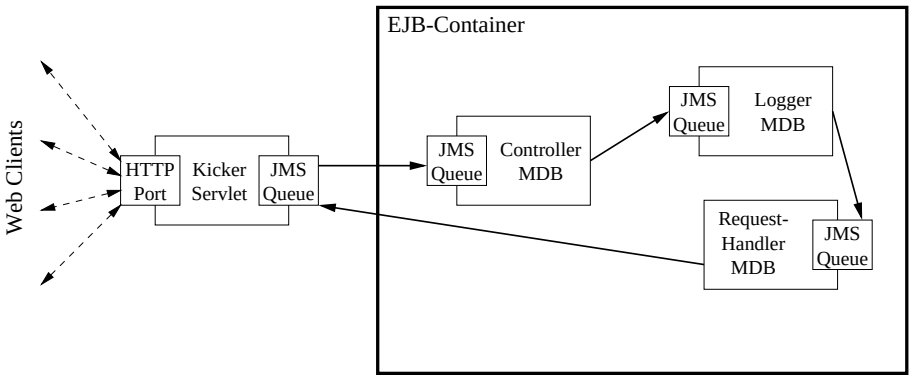


Abbildung 2.1: Systemaufbau

Das Neue daran ist bloss, dass der Verarbeitungpfad nicht mehr fest kodiert sein wird, sondern durch einen Routingalgorithmus zur Laufzeit bestimmt und soll somit jederzeit geändert werden können. So wird es zum Beispiel möglich sein, dass ich dem Router mitteile, er soll den Logger auslassen, und die Nachricht vom Controller direkt zum RequestHandler leiten. Das muss komplett ohne Veränderung am Sourcecode machbar sein.

2.2 Routing

Es gibt verschiedene Ideen wie der Abarbeitungspfad bestimmt werden kann. Beispielsweise von der Person, die den Server startet und konfiguriert und dabei die Route “von

Hand" in ein Konfigurationsfile niederschreibt. Das könnte auch unter Zuhilfenahme eines Softwaretools geschehen, mit dem es möglich, wäre die Beans grafisch zusammenzustellen. Weiter könnte dabei ein intelligentes System, welches die Eigenschaften der Beans kennt, den Operator bei seiner Arbeit unterstützen oder ihn im Idealfall ersetzen und die Komponenten automagisch zu einem lauffähigen System verbinden.

Wie das in Zukunft passieren soll, möchte ich nicht vorweg nehmen. Ich bin aber der Meinung, dass sich diese Frage leicht vom eigentlichen Routingalgorithmus entkoppeln lässt. Als gemeinsamen Nenner aller Möglichkeiten zur Routenbestimmung sollte dabei eine Datenstruktur erstellt werden, welche dem weiteren Routingmechanismus einen eindeutigen Abarbeitungspfad liefert. Diese Datenstruktur könnte in XML formuliert sein und in einer Datei gespeichert werden. Wie sie aber entsteht ist vorerst unwesentlich und somit werden ich für diese erste Iteration einfach mit einem beliebigen Editor ein solches XML-Routingfile erstellen.

Die XML-Code in Abbildung 2.2 zeigt eine simple Instanz für eine Routing-Datenstruktur.

```
<?xml version="1.0"?>
<graph>
 <node name="Controller">
  <route>
   <input>Kicker</input>
   <output>Logger</output>
   <exception>Kicker</exception>
  </route>
  <route>
   <input>RequestHandler</input>
   <output>Kicker</output>
   <exception>Kicker</exception>
  </route>
 </node>
 <node name="Logger">
  <route>
   <input>Controller</input>
   <output>RequestHandler</output>
   <exception>Kicker</exception>
  </route>
  <route>
   <input>RequestHandler</input>
   <output>Kicker</output>
   <exception>Kicker</exception>
  </route>
 </node>
 <node name="RequestHandler">
  <route>
   <output>Kicker</output>
   <exception>Kicker</exception>
  </route>
 </node>
</graph>
```

Abbildung 2.2: Routingfile

Das Hauptelement <*graph*> enthält mehrere Knoten <*node*>. Es muss für jedes MDB genau ein solches Element existieren und durch das Attribut *name* gekennzeichnet werden,

welcher *node* zu welchem Bean gehört. So kann jedes MDB schnell seinen Teilbereich aus der Struktur herauslesen, welches die Routinginformation für sich selber enthält. In diesem Knoten sind dann eine oder mehrere Routen definiert. Das Tag <*input*> bestimmt ob die Route zur Verarbeitung des aktuellen Requests passt oder nicht. Das EJB muss dazu jedoch wissen, von wem es die Message empfangen hat. Ist kein Input-Tag angegeben heisst es, dass die Message irgendwo her kommen kann und diese Route trotzdem verwendet wird. Grundsätzlich soll für die Wahl des Pfades die first fitRegel gelten, also die erste Route die mit dem gegenwärtigen Absender überein stimmt, soll gewählt und die darin enthaltenen Output- und Exception-Pfade für die Weiterleitung von Nachrichten oder Fehlermeldungen verwendet werden.

2.3 Message Properties

Es gibt verschiedene Arten von Messages, die mit dem Java Messaging Service (JMS) verschickt werden können. Etwa einfache Text oder Byte Messages, Stream Messages oder solche die ein Java Objekt als Inhalt bergen können. Zusätzlich zum Inhalt der Message bietet JMS die Möglichkeit Properties zur Nachricht hinzuzufügen. Dies ist eine Möglichkeit verschiedene Argumente in der Message getrennt ablegbar und zugreifbar zu speichern. Durch einen *String* werden die Properties identifiziert und als Typ kommen *boolean*, *byte*, *short*, *int*, *long*, *float*, *double* und *String* in Frage. In diesem ersten Prototyp habe ich die folgenden Properties benutzt, um Informationen über den Request zu übermitteln.

RequestId: Eine eindeutige Nummer zur Unterscheidung der einzelnen Requests. Wird vom Kicker Servlet berechnet und hinzugefügt.

Request: Die eigentliche, zu verarbeitende Anfrage. Soll in einer Xlinkbase spezifischen XML Query Language formuliert sein, die aber bisher noch nicht definiert ist.

Sender: Der Name des Senders der gegenwärtigen Nachricht.

RoutingInfo: Beinhaltet den Routing Graphen wie der in Abschnitt 2.2 beschrieben ist. Dieser wird vom Controller MDB in die Message eingefügt.

Response: Hier kann ein Bean, das auf den Request eine Antwort produziert hat, den Output in einer geeigneten Codierung anfügen.

Exception: Falls die Verarbeitung zu einem Fehler führt, kann hier Information über die Ursache mitgeliefert werden.

Der Typ aller beschriebenen Properties ist String, ausser die RequestID, die vom Typ int ist.

2.4 Subclassing

Die Wahl der Message Driven Beansäls Grundlage zur Implementierung des Xlinkbase Servers, macht es für einen Programmierer sehr einfach weitere Beans zu erstellen. Im Gegensatz zu einem Entity oder Session Bean braucht ein Message Driven Bean kein Home-Interface oder sonst eine zusätzliche Klasse die programmiert werden muss. Einzig die beiden Interfaces `MessageDrivenBean` und `MessageListener` müssen implementiert werden. Konkret heisst das, die Methoden `ejbActivate()`, `ejbRemove()`, `ejbPassivate()` und `ejbCreate()`, die zur Verwaltung der Instanzen vom Container aufgerufen werden und die Methode `onMessage(Message)`, die bei Ankunft einer Message aufgerufen wird und worin dann die Business Logic des Beans stehen soll.

Durch diese Einfachheit dieses Bean Typs bleibt ein sehr nützliches Konzept der Objektorientierten Programmierung zur Verwendung offen - und zwar das Subclassing. In der EJB Spezifikation [7] steht dazu folgendes:

> The message-driven bean class may have superclasses and/or superinterfaces. If the message-driven bean has superclasses, the `ejbCreate` method, and the methods of the `MessageDrivenBean` and `MessageListener` interfaces may be defined in the message-driven bean class or in any of its superclasses.
>
> The message-driven bean class is allowed to implement other methods (for example, helper methods invoked internally by the `onMessage` method) in addition to the methods required by the EJB specification.

In dieser ersten Version meines Xlinkbase Servers habe ich diese Eigenschaft ausgenützt und eine Superklasse `XlinkbaseBean` erstellt. Diese Klasse erfüllt alle Forderungen der EJB Spezifikation für Message Driven Beans, ausser dass diese nicht direkt instanzierbar ist, sondern erst durch eine Subklasse erweitert werden muss.

2.5 Programmierung

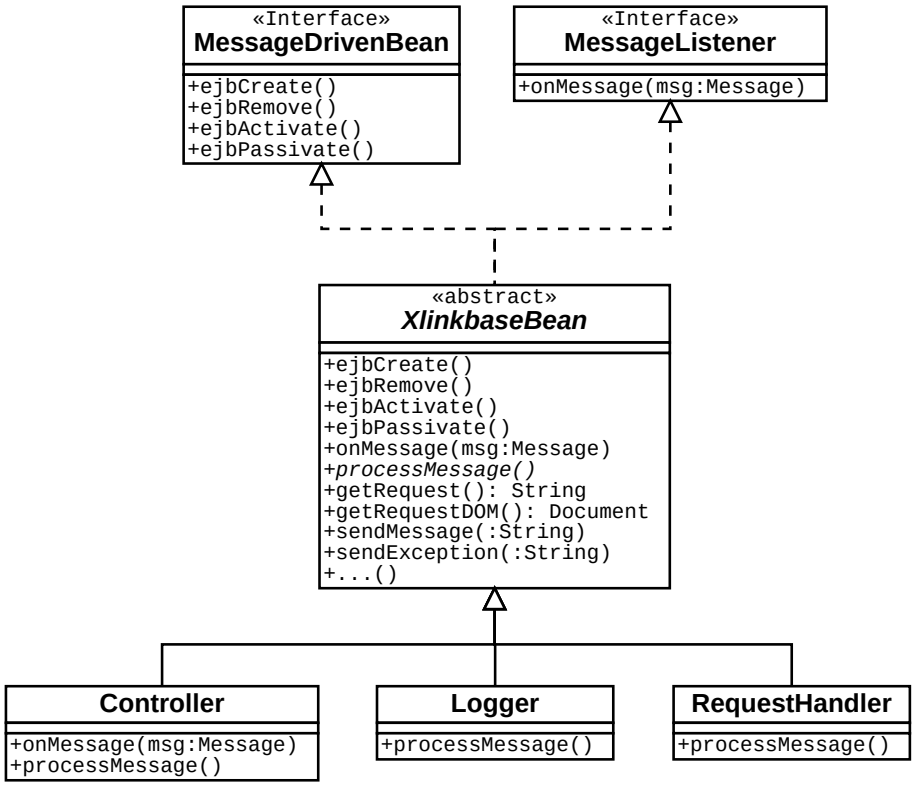


Abbildung 2.3: Klassendiagramm

Die Abbildung 2.3 zeigt das Klassendiagramm für die erste Version meines Xlinkbase Servers. Die Klasse `XlinkbaseBean` enthält schon die fertig ausprogrammierten Methoden des `MessageDrivenBean` Interfaces und die Methode `onMessage` aus dem `MessageListener` Interface und erledigt bereits alle notwendigen Massnahmen zum Routing der Messages. Die Properties werden aus der Message herausgelesen, der Absender festgestellt, die Routingstruktur geparst, daraus die Route festgestellt und schliesslich die Output und Exception Pfade gesetzt. Dann wird die Kontrolle an die neue Methode `processMessage()` weiter gegeben. Diese ist in `XlinkbaseBean` mit dem Keyword

`abstract` definiert und ist somit die einzige Klasse, die dann in der Subklasse noch implementiert werden muss. Aus diese Weise wird der Routingmechanismus für den Beanprogrammierer völlig transparent. Man könnte Subklassen von `XlinkbaseBean` gar als neuen Beantypen ansehen, als Routed Message Driven Beans (RMDB).

Das Controller Bean beinhaltet in diesem Schema eine kleine Ausnahme. Da bei Ankunft eines Requests von Servlet die Routing-Datenstrukur noch nicht in der Message enthalten ist, funktioniert die `onMessage` Methode noch nicht. Aus dem Grund wird im Controller diese Methode überschrieben mit einer eigenen Implementation, welche zuerst das File mit der Routingstruktur einliest und deren Inhalt in die Message kopiert. Alles andere funktioniert jedoch identisch wie im Logger und im RequestHandler.

Für die auf der `XlinkbaseBean`-Klasse aufbauenden Beans stehen in ihrer Superklasse weitere unterstützende Methoden zur Verfügung. Einerseits zum Anfordern des Requests als String oder gerade in geparster Form als DOM-Tree, andererseits zum Weitersenden von neuen Messages oder Exceptions.

2.6 Erkenntnisse aus der Iteration 1

Aus der Iteration 1 ist ein sehr einfaches Framework entstanden, mit dem Requests auf die Xlinkbase verarbeitet werden können. Die Beans sind einfach zu programmieren und laufen unabhängig und asynchron zu einander, was wichtig ist um eine gute Performance zu ermöglichen. Das System ist gut skalierbar. Das heisst, dass im Server beliebig viele Instanzen der Beans parallel arbeiten können und das über mehrere, in Clusters organisierte Rechner verteilt. Die meisten modernen Application Server unterstützen dieses Clustering, auch der von mir verwendete *Bea Weblogic 6.0sp1*.

Die verschiedenen Beans können in beliebiger Reihenfolge zusammengesteckt werden um dem Request zu behandeln. Aber nur in einer Linie. Es ist also bisher nicht möglich im selben Bean mehrere Messages zu senden oder zu empfangen. Komponenten die mehrere Datenpakete zu einem verschmelzen oder eines in mehrere aufteilen, sind so nicht machbar. Das wäre sicher ein nützliches Feature für eine der nächsten Iterationen.

Es gibt einige Punkte die zum Zwecke der Einfachheit noch nicht sauber gelöst wurden. Das Aussehen des Messages und deren Inhalt sollte noch genauer definiert werden. Der Name des Beans, das für das Routing verwendet wird, muss separat als Environmentvariable in das Deploymentfile geschrieben werden. Weiter sollte die Routing-Datenstruktur nicht in einer simplen Datei abgelegt werden, weil das in über mehreren Rechner verteilten Clusten zur Unauffindbarkeit des Files führen kann. Ausserdem sollte in Enterprise Java Beans generell auf das Paket “java.io.*” verzichtet werden.

Mit dem Essen kommt der Appetit! Beim Programmieren ist das ähnlich, und so sind mir weitere Features eingefallen, die in Zukunft möglich sein sollten:

- Das “Session Konzept” das in den Servlets vorhanden ist, sollte auch in den MDBs verwendbar sein. Authentifizierende Beans könnten eine Anwendung davon sein.

- Die Java Transaction Architecture (JTA), die bereist für die EJBs zur Verfügung steht, könnte auch in den Xlinkbase Beans Verwendung finden. Abfragen und Manipulationen an der Xlinkbase könnten so die ACID-Eigenschaften (atomicity, consistency, isolation, durability) verliehen werden, was in sicherheitskritischem Umfeld sicher sehr geschätzt würde.

Kapitel 3

Iteration 2

3.1 Use Case

In der zweiten Iteration möchte ich ich einen der Hauptanwendungszwecke des Xlinkbase-Servers untersuchen, nämlich die XSL-Transformation von XML Files und Topic Maps und wie sie mit dem vorhandenen Framework realisiert werden könnten. Dabei soll der Prozess im mehrere einzelne Aufgaben zerlegt werden, damit die asynchrone und parallele Funktionsweise der Message Driven Beans voll greifen kann. So könnten für die Phasen, Beschaffung des zu verarbeitenden XML Files, Beschaffung des Stylesheets und Durchführung der Transformation, drei selbstständige Beans programmiert werden. Das XML File und das Stylesheet könnten aus dem sekundären Speicher gelesen werden oder auch von weiteren MDBs generiert werden.

3.2 Synchronizer

Es gibt mehrere Möglichkeiten wie die drei Beans für die XSL-Transformation mit Messages miteinander kommunizieren können. Die Abbildungen 3.1 bis 3.3 zeigen drei davon.

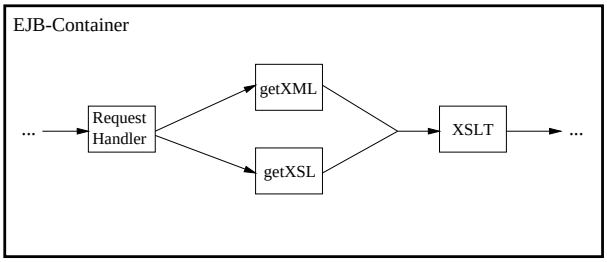


Abbildung 3.1: Stylesheet Processor 1

Um die erste Variante implementieren zu können, müssen die Messages vom getXML- und getXSL-Beans miteinander synchronisiert werden und gemeinsam an ein XSLT-Bean verschickt werden. Das ist aber nicht trivial, weil von jedem MDB beliebig viele Instanzen existieren und diese auch noch über mehrere Rechner verteilt sein können. Man bräuchte eine Art Singleton, mit dem sichergestellt werden kann, dass beide Messages von der selben Instanz empfangen werden. Ein Singleton wäre aber eine gravierende Einschränkung in die Parallelisierung und Verteilbarkeit der Serverkomponenten und widerspricht auch völlig dem Konzept der Enterprise Java Beans. Rein technisch gesehen, gäbe es schon Möglichkeiten dies zu realisieren. Zum Beispiel mit nur einfach vorhandenen Beans, speziellen

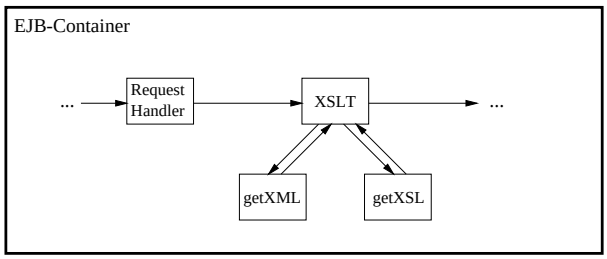


Abbildung 3.2: Stylesheet Processor 2

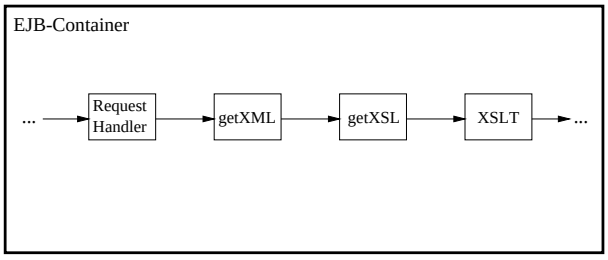


Abbildung 3.3: Stylesheet Processor 3

Entity-Beans oder mit statischen Datenstrukturen. Die Abbildung 3.4 zeigt einen Kommentar aus einer FAQ [19], welcher die Problematik von Singletons verdeutlicht. Weitere interessante Ressourcen dazu gibt es unter [20].

Auch der *Stylesheet Processor 2* hat Mängel was die Parallelisierung betrifft. Hier wird einfach eine synchrone Kommunikation durch die JMS-Messages emuliert. Aber auch das ist nicht ohne weiteres programmierbar, weil mehrfacher Empfang von Messages in MDBs nicht unterstützt wird.

Dir dritte Variante ist vom Messaging her die einfachste, da jedes Bean nur eine Message empfängt und auch nur eine weiter sendet. So ist es völlig egal, welche Instanz eine Message empfängt und auf welchem Rechner sie lebt. Die einzige Schwierigkeit besteht darin, dass das XSLT-Bean auch auf die Ausgabe des getXML-Beans angewiesen ist, dessen Output aber nur an das getXSL-Bean geschickt wird. Es muss also sichergestellt werden, dass auch auf weiter zurückliegende Resultate zugegriffen werden kann.

Nach langem Überlegen habe ich mich für die dritte Variante entschieden. Die erste Variante wäre zwar die flexibelste, liesse sich aber nur durch sehr unschöne Programmiertricks implementieren. Die lineare Anordnung der Abbildung 3.3 passt am besten in das Konzept der EJBs.

3.3 Stack

3.3.1 Datenstack

Damit in einer linearen aneinander Reihung von MDBs noch genügend Flexibilität vorhanden ist, um auch komplexere Aufgaben implementieren zum können, brauche ich eine Möglichkeit auf die Daten aller Vorgänger-Beans zuzugreifen. Dazu habe ich den Datenstack eingeführt. Die Ausgabe eines Beans, die ein Programmierer anderen Beans zur Verfügung stellen möchte, wird durch das Framework automatisch in XML formuliert und zuoberst auf einen Stack gelegt. Dieser Stack wird immer als Teil der JMS-Message weiter geschickt. Es wird auch ein API angeboten, um die einzelnen Datenelemente wieder

What should I do if I need a singleton?

Say you want a stateful, shared, server-side object--that is, a singleton. EJB does not provide this design pattern explicitly. You could use an entity bean, but these are really intended to be stored in either a database or some (other) legacy backend. Writing a stand-alone entity bean (which provides full transactional semantics, as entities should) is overkill for a simple singleton. Or, you could use a stateful session bean, but these are intended to be single-user. (In reality, it is not possible for a container to enforce this rule and, thus, it could be ignored, but again we find ourselves doing something quite unnatural in EJB.)

So we reach the inevitable conclusion that EJB provides a useful set of server-side design patterns, but does not provide ALL the useful patterns. Developers will find that jumping out of EJB (presumably into CORBA) will be useful in a number of cases. And, of course, our product makes this transition painless, in that it preserves the transactional semantics, and security, when crossing the EJB-CORBA boundary. This preservation of distributed-system semantics is not easily doable (and is not being done by competitors' products) if the EJB container is not built on top of IIOP (for example, CORBA).

Abbildung 3.4: Kommentar über Singletons in EJBs

auszulesen.

3.3.2 Kommandostack

Die Wegbeschreibung durch die einzelnen MDBs wird nicht mehr, wie in der ersten Iteration, aus den Routing-File gelesen und in jeder Instanz neu ausgewertet. Neu steht jetzt die Route in einer *Environment Variable*. Diese Variable wird im Deployment-File des Controller gesetzt. Dadurch werden die in einem verteilten System problematischen Dateizugriffe eliminiert. Der Controller wertet die Angaben darin aus und erzeugt einen Kommandostack. Der Kommandostack wird wie der Datenstack, in XML-Form der Message angehängt. So kann der Routingalgorithmus einfach das oberste Element von Stack ziehen und findet darin die Information, wohin die Message weiter geschickt werden soll.

Dieser Kommandostack bietet noch die sehr interessante Funktion des *Conditional Routing*. Da in vielen Anwendungen erst klar wird wohin die Route führen soll, wenn der empfangene Request geparst worden ist, ist es oft wünschenswert, den Abarbeitungspfad dynamisch zu setzten. Das ist nun sehr einfach möglich, indem nun jedem Bean eine API-Methode zur Verfügung steht, die neue Kommandos auf den Stack legt. Ein Kommando entspricht einfach dem Routingnamen des Beans, das aufgerufen werden soll. Zusätzlich kann eine Reihe von Attributen in Form von Strings angehängt werden, die zur Konfiguration des Arbeitsschrittes verwendet werden können, wie man aus Unix Shell-Kommandos kennt. Der Programmierer eines neuen Beans der neue Kommandos hinzufügen möchte, muss sich eines Bewusst sein. Da neue Kommandos immer zuoberst auf den Kommandostack gelegt werden, ist die tatsächliche Abarbeitung genau in umgekehrter Reihenfolge, als sie im Programmcode aufgeführt sind.

3.4 Message Frame

In allen Messages die unter den *XlinkbaseBeans* ausgetauscht werden, muss ein *Message Frame* enthalten sein. Die Abbildung 3.5 zeigt ein Beispiel davon.

Das Frame wird vom Controller aufgebaut, falls noch keines in der Message vorhanden ist, wobei der Request-String einfach als ein erstes Datenelement auf den Datenstack gelegt wird. Ebenso entfernt der Controller am Ende der Abarbeitung das Frame aus der Message und gibt das Resultat der Anfrage an den Client zurück. Dabei wird nun einfach verlangt, dass die Response zuoberst auf dem Stack und in der geforderter Formatierung vorliegt.

```
<message>
  <commands>
    <Logger/>
    <RequestHandler/>
    <Controller/>
    <Kicker/>
  </commands>
  <stack>
    <data>
      request=linux
    </data>
  </stack>
</message>
```

Abbildung 3.5: Message Frame, Iteration 2

3.5 Message Properties

Die Message Properties spielen in dieser Version des Xlinkbase Servers nicht mehr eine so grosse Rolle. Es gibt jetzt nur noch drei davon.

RequestId: Ist nachwievor wichtig, damit der Client die Response zu einer gestellten Anfrage zuordnen kann, wird aber vom Server selber nicht verwendet.

Sender: Der Name des Senders, bzw. der Queue die mit diesem assoziiert ist.

JMSReplyTo: Dies ist streng genommen keine Property. Hiermit kann aber die Queue angegeben werden, wohin die Response oder auch allfällige Exceptions geschickt werden sollen.

3.6 Erkenntnisse aus der Iteration 2

Mit dieser zweiten Iteration ist die Programmierung von verteilten Webapplikationen noch einfacher geworden. In diesem Framework fehlt wirklich nur noch die Business Logic, alles andere wird von der Superklasse abgenommen. Das API ist sehr schmal, sollte aber ausreichend sein für alle Anwendungen, für die ein solches System auch sinnvoll ist. Einige Detailverbesserungen sind aber schon noch angebracht. So würde ich es zum Beispiel noch nützlich finden, wenn ähnlich wie bei den Kommandos auch die Datenelemente Attribute hätten, so dass sie für die Nachfolger-Beans einfacher identifizierbar sind.

Beim Test dieser Version der *Routed Message Driven Beans* hat sich herausgestellt, dass die Kommando-Attribute nicht richtig funktionieren. Die Behebung dieses Fehlers verschiebe ich jedoch in die nächste Iteration, weil da noch weitere Änderungen an diesem Teil der Beans vorgesehen sind.

Kapitel 4

Iteration 3

4.1 Use Case

Für die dritte und letzte Iteration habe ich nur wenige Änderungen vorgesehen. Der Anwendungsfall bleibt der selbe wie in der Version zuvor, doch stehen noch ein paar Detailverbesserungen an.

Es ist der Wunsch aufgekommen, dass die abgearbeiteten Kommandos nicht einfach aus dem Stack gelöscht werden, sondern in der Struktur verbleiben und im nachhinein eingesehen und ausgewertet werden können. Ein zusätzliches Bean, genannt Monitor, kann am Ende der Route eingefügt werden, welches diese Daten ausliest und dann zum Beispiel entscheidet ob die Abarbeitung korrekt und vollständig erfolgt ist und dann eventuell weitere Massnahmen auslöst.

Die zweite wichtige Ergänzung betrifft die Datenelemente auf dem Datenstack. Diese sollen von den MDBs besser identifiziert werden können.

4.2 History

Um den vollständigen Weg durch die RMDBs am Schluss der Verarbeitung untersuchen zu können, habe ich am Message Frame ein paar Erweiterungen vorgenommen. Ein neuer Ast in der XML-Message mit dem Namen *history* nimmt jetzt die ausgeführten Kommandos auf. Für das Kommando, welches gerade von einem Bean abgearbeitet wird, gibt es jetzt auch einen eigenen Platz im XML-Baum. Dieses wird unter dem Tag *current* angehängt. Mit dieser Änderung wurde auch gerade der Bug mit den Kommando-Attributen, der in der Iteration 2 noch vorhanden war, behoben. Im schematisierten XML-Baum in Abbildung 4.1 wird verdeutlicht, wie die Kommando-Elemente im DOM-Tree umgehängt werden.

Diese *History* wird aber nicht unbedingt in jedem Server benötigt und kann darum auf Wunsch auch abgeschaltet werden. Dazu muss im *Document Element*, also im Tag *message*, ein Attribut mit der Bezeichnung *history* angebracht werden und dessen Wert auf *no* statt auf *yes* setzen. Dieser Schritt wird vom Controller Bean erledigt und kann im Deploymentfile *ejb-jar.xml* über die Environment Variable *history* mit den Stringwerten *yes* oder *no* gesteuert werden.

4.3 Data ID

Neu werden allen Datenelementen auf dem Stack eine Identifikationsnummer zugeordnet. Das bietet eine einfachere Möglichkeit diese zu identifizieren. Beim Hinzufügen eines Eintrags auf den Stack wird eine neue ID berechnet und dem Benutzer zurückgeliefert. Diese

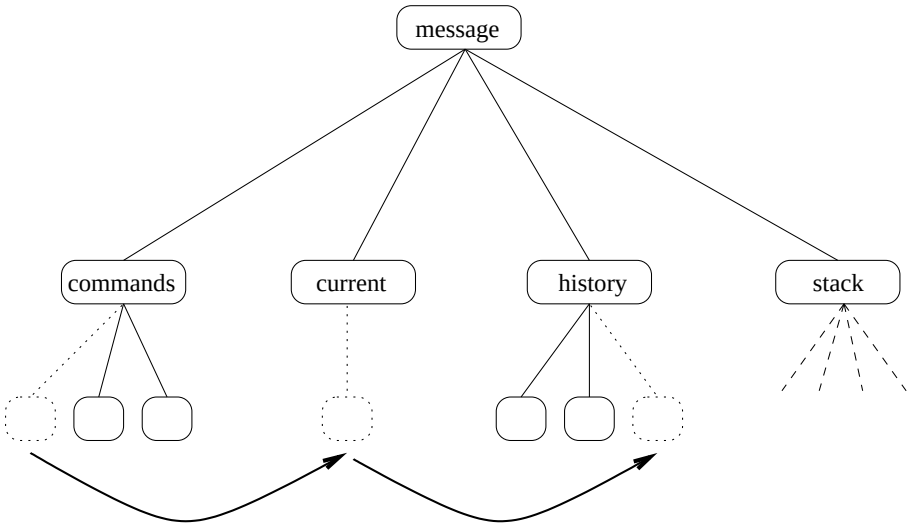


Abbildung 4.1: History Mechanismus

ID könnte dann einem neuen Kommando als Attribut mitgegeben werden, um dem Bean mitzuteilen, auf welchem Datensatz es zu arbeiten hat. Entsprechend gibt es eine API Methode, die für eine ID das entsprechende Datenelement vom Stack zurück gibt.

Die Daten ID wird auch im *Current*-Kommando als Eintrag im Attribut *idrefs* vermerkt; falls ein RMDB mehrere Datenelemente auf den Stack legt, entsprechend als Liste durch Leerzeichen getrennt. Wenn das abgearbeitete Kommando später in die History verschoben wird, kann somit auch festgestellt werden, welches Bean welche Daten produziert hat.

4.4 Message Frame

Mit den neusten Ergänzungen sieht ein typisches Message Frame etwa wie in Abbildung 4.2 aus.

4.5 Erkenntnisse aus der Iteration 3

Durch die Daten IDs, zusammen mit den jetzt gut funktionierenden Kommando-Attributen, ist der Datenstack jetzt einiges durchsichtiger geworden. Statt sich immer nur auf das Stackprinzip verlassen zu müssen, können jetzt Argumente explizit adressiert werden. Damit wird die Programmierung von RMDBs intuitiver und eine Quelle für Fehler kann so ausgeschlossen werden.

Die History verschafft dem Administrator ein Werkzeug zur Kontrolle des Message-Routing in seinen Beans. Das gibt ihm mehr Überblick und eine Möglichkeit zum Debuggen seiner Anwendung.

Abschliessend meine ich, dass dieses Framework eine gewisse Reife erlangt hat, um in Webservern eingesetzt zu werden. Der Anwendungsbereich geht dabei weit über die Generierung von Webseiten hinaus. Vielmehr ist es eine Basis für die Erstellung allgemeiner Applikationen, die über das Internet einer breiten Öffentlichkeit zur Verfügung gestellt werden kann.

```
<message history="yes">
  <commands>
    <Controller/>
    <Kicker/>
  </commands>
  <current>
    <Monitor/>
  </current>
  <history>
    <Controller idrefs="0"/>
    <Logger/>
    <RequestHandler idrefs="1">
      <attribute>
        request=0
      </attribute>
    </RequestHandler>
  </history>
  <stack>
    <data id="1">
      // Daten ausgelassen
    </data>
    <data id="0">
      request=linux
    </data>
  </stack>
</message>
```

Abbildung 4.2: Message Frame, Iteration 3

Routed Message Driven Beans: A new Abstraction for using EJBs

Erik Wilde and Manfred Meyer
Computer Engineering and Networks Laboratory
Swiss Federal Institute of Technology, Zürich

ABSTRACT

Asynchronous messaging between cooperating software components proves to be useful in many scenarios. One framework supporting this functionality is Sun's J2EE platform with its *Message-Driven Beans (MDB)* model. We present a novel way to use MDBs by providing a way to add routing information to the messages, which is then used to send a message through a given path of processing components. We call this model *Routed Message-Driven Beans (RMDB)*, and the two main topics that are important for RMDBs are (1) the message format that is used for the routing information, and (2) the API which can be used by programmers to take advantage of the abstraction provided by RMDBs. Performance measurements show that the overhead caused by our RMDB framework is acceptable if messages are routed through several EJBs.

1. MOTIVATION

Programming frameworks are used to make program development easier, faster, and more reliable, and the overall goal is to produce better software with less expenses. In the last 10 years, the *World Wide Web (WWW)* [15] — and in particular applications built on top of the services provided by it — has increasingly been the focus of software developments, as indicated by frequently used keywords such as *Intranet*, *Business-to-Business (B2B)* and *Business-to-Consumer (B2C)*, or other keywords referring to Web- or Internet-specific technologies. Thus, programming environments for Web-enabled applications have become quite popular.

In the context of a Web-focused software system, we have developed the *Routed Message-Driven Bean (RMDB)* abstraction for asynchronously routed messages between *Enterprise Java Beans (EJB)*, based on a pre-computed path through a any number of RMDBs. These RMDBs all contribute to the processing of the message as cooperating software components. In the following sections, we briefly introduce the topics relevant to our work.

1.1 Web Servers

Web server architectures have gone a long way from the first Web server (implemented in 1989/90 [1]) to the complex integrated server architectures of today. The first Web server simply mapped the request's URI path to a file system path, and then returned the file's content in the response. While the communications mechanism between Web clients and servers, the *Hypertext Transfer Protocol (HTTP)* [5], basically still works the same way than it did 10 years ago (clients send a request with a URI path and get back a response containing a result), the architectures on the server side have evolved significantly. Three basic evolutionary steps can be identified:

- *Server directives*

  Simple directives for invoking pre-defined server operations, such as inclusion of a file's content or inclusion of the modification date of a file. These mechanisms are very easy to use, but also very limited in their functionality. Extending the server's functionality means extending the server's set of supported directives, which normally involves modifying the server software.

- *Scripting languages*

  Scripting languages are still very popular today, the most widely used are probably PHP[1], Sun's *Java Server Pages (JSP)*, and Microsoft's *Active Server Pages (ASP)*. Scripting languages are embedded into Web pages, and the Web server scans the Web page for scripting before delivering it. Scripting languages are a very powerful way to embed dynamically generated information into static content.

  Scripting languages can be used in conjunction with other technologies, a very popular example being the *LAMP* (Linux, Apache, MySQL, PHP) application suite [14], which makes it possible to easily access relational databases from within PHP's scripting environment. However, the applications used by the scripting languages are outside of the scope of the scripting language itself.

- *Programming environments*

  Complete programming environments for Web services not only include a scripting language for embedding dynamically generated information into static content, but also an entire programming framework for building

[1] See `http://www.php.net/`

Web-based applications. These environments are undoubtedly more complex than the rather simple scripting languages, but they are becoming increasingly popular due to the growing number of complex Web-based applications.

One important class of programming environments are application servers based on Sun's *Java 2 Enterprise Edition (J2EE)* [11], which is a platform based on a Java run-time environment and a number of additional services. J2EE is not the only programming environment for Web-centric applications (Microsoft's .NET platform is a well-known competitor), but with dozens of implementations (including several free software initiatives) it is the most widely implemented one.

J2EE is a specification from Sun Microsystems [12], including a large number of technologies, most notably a Java runtime environment as the most basic component, *Java Database Connectivity (JDBC)* for database connectivity, *Java Server Pages (JSP)* as the scripting approach for Web pages, and EJB as its software component model. In the latest release of J2EE (version 1.3), the *Java Message Service (JMS)* [7] and the *Java Transaction API (JTA)* [3] have been added to better support EJB development. Furthermore, the following enhancements have been made to the architecture:

- A new kind of enterprise bean, the *Message-Driven Bean (MDB)*, enables the asynchronous consumption of messages.
- Message sends and receives can participate in JTA transactions, thus making it possible to combine JMS-based asynchronous messaging with the transaction services provided by JTA.

In many application scenarios involving several autonomous services, asynchronous messaging can be a very powerful abstraction. In our application scenario, we have a large database of URI references and semantic relationships between them, and we are implementing a server that can make use of this data in a variety of ways, effectively implementing a specialized database for linking information (often also referred to as a linkbase [4]).

1.2 XLinkbase

The XLinkbase system is based on the observation that it is not the amount of information available on the Web which often limits our ability to use it in a meaningful way, but the lack of relationships between individual information resources. XLinkbase is based on the idea of an *Open Hypermedia System (OHS)* [9], and its data model is similar to Topic Maps [8,10]. In the context of this paper, the interesting aspect of XLinkbase is that its current implementation is built on top of the J2EE platform and JMS. For a more detailed discussion of the ideas and concepts behind XLinkbase, please refer to a paper by Lowe and Wilde [16].

Basically, requests to the XLinkbase server may be issued via HTTP to a Web server, or from other applications via an EJB API. The request is then handled by a controller EJB, which computes routing information for the request and then sends it as a message via the RMDB mechanism. The routing information specifies which EJB(s) should process the message, and the RMDB framework makes the routing transparent for the XLinkbase-specific EJB code.

Seen from an architectural perspective, the RMDB framework is used by the XLinkbase server to implement message routing among the EJBs within the XLinkbase server. These EJBs may implement functionalities such as XSLT transformations, logging, security checks, access to data storage for XLinkbase data (such as access via JDBC), or any other functionality that might be useful within the XLinkbase server. XLinkbase is designed as a generic platform for implementing linkbases, and it can easily be extended by implementing new functionality in RMDB components, which can then be used to process requests.

A typical example of an XLinkbase request is the following scenario: The XLinkbase Web server receives a request and forwards it to the controller EJB. The controller computes a processing graph, consisting of any number of RMDBs, for example a logging component, a security check, a database access component retrieving data, an XSLT transformation component converting the XLinkbase data to HTML, and another logging component, before the controller finally gets back the result of the request processing and sends it to the client via the Web server. All the components are RMDBs, and they use the RMDB framework to transparently route the message between the components. If at any time the processing graph changes (for example because the security components rejects the request and needs to re-route it), the RMDB routing information can be changed via the API to reflect the new processing graph. In general, all interaction between the application functionality and the RMDB framework is handled via the RMDB API.

2. RELATED WORK

Web-based application frameworks are very popular today, and they range from rather small software suites to rather heavy programming environments such as J2EE. One of the early approaches to use XML technologies for building a publishing framework is Apache's *Cocoon*[2] project, which is built on standard Web technologies such as DOM, XML, and XSL(T). However, Cocoon does not support distributed processing and load balancing, and because we wanted to be able to support this, we decided to not use it.

One step further, the *Java 2 Platform Enterprise Edition (J2EE)* supports distributed processing and load balancing, and (starting with version 1.3) also supports message passing between components. Furthermore, J2EE is a specification (and not a product), and therefore software built on top of it is not restricted to one vendor's platform. Even though most J2EE products are commercial products (with BEA System's *WebLogic* and IBM's *WebSphere* currently being the market leaders), there are also open source J2EE implementations such as JBoss[3].

J2EE provides a powerful programming environment, but from an application programmer's point of view, the level of support for the typical J2EE application is not as good as it could be. Sun Microsystems, author of the J2EE specification, has recognized this and now offers the so-called *J2EE Blueprints*[4], which basically are design patterns [6] for using EJBs. At the time of writing, the set of blueprints available is quite small, and we see our framework as one contribution to the overall goal of producing generic solutions for J2EE,

[2]See `http://xml.apache.org/cocoon/`
[3]See `http://www.jboss.org/`
[4]See `http://java.sun.com/j2ee/blueprints/`

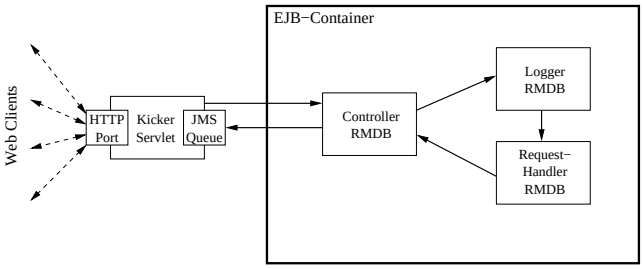


Figure 1: XLinkbase server

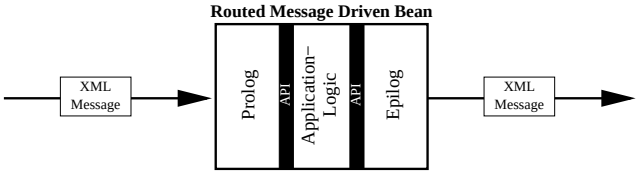


Figure 2: RMDB framework

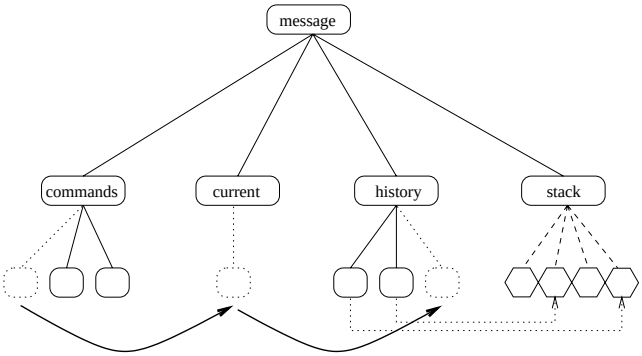


Figure 3: RMDB message structure

which may be re-used for similar application scenarios.

Asynchronous messaging is provided in many environments[5], but the ability to compute (and modify) paths through any number of components, and to pass the path information as well as any results through these components, to our knowledge is not available as a generic component in any J2EE platform.

3. THE CONCEPT

In a very simple setup, an XLinkbase server could be constructed as in Figure 1. A servlet is set up to receive client requests from browsers or other Internet applications. This servlet then kicks off the RMDB mechanism by sending a JMS message to the Controller in the EJB container. The Kicker servlet must have its own JMS Queue to be able to receive the response in order to send it back to the client.

RMDBs are based on the *Message-Driven Beans (MDB)* model of the new *Enterprise Java Beans (EJB)* 2.0 specification. MDBs are a very simple type of EJBs. They are exclusively addressed through the JMS queue or topic they are deployed to listen to. There is no `home` and no `remote` interface and they can be coded in one single class file by simply implementing two interfaces.

Like any MDB, a RMDB starts to work when a message arrives in the JMS queue. In the MDB model, next to the *JMS Queues* there also exist *JMS topics* which make it possible to send a message to many recipients. However, for RMDBs these JMS topics are not supported, because the message path is not allowed to split anywhere in the processing path, as there can only be one message per request in the server. All communication is done using JMS text messages, because they can carry a payload of type `string`, which enables the RMDBs to send and receive XML strings. The message is an XML document containing all the information about what the RMDB is supposed to do:

- The command and its parameters
- The data to process
- The route to the next RMDBs

The message format and its semantics are described in detail in Section 4.1. Before the application logic of the RMDB has access to the data in the message, the RMDB message (which is an XML document) must be parsed. After processing it, a new message must be compiled and serialized to a `string`, in order to send it to the next RMDB. Figure 2 shows the schematic message flow through one RMDB.

The parsing is done in the RMDB's prolog, while the generation, routing, and sending of the new message is done in the epilog phase. This behavior is identical for every RMDB, and consequently the code for doing this can be encapsulated in a framework to be transparently processed at the beginning and the end of an RMDB's execution. Creating a new RMDB using this framework is then reduced to implementing the application logic. The access to the XML data is provided by an API and similarly data can be added to the message as well as adding new commands to the path for delegating work to other RMDBs.

4. IMPLEMENTATION

Two main aspects discussed in this section are the format of the XML message, and the class structure used to implement the routing and data handling mechanisms. The actual XML message is never visible for the RMDB programmer, since access to it is exclusively provided through an API. Because of the sophisticated design of the message format, RMDBs can be kept stateless and the processing of messages can be easily parallelized and distributed.

The class structure described in Section 4.2 has been invented to hide framework code from the developer who wants to create new RMDBs and to provide a hook to insert the RMDB's application logic into the framework. The transparency achieved by this is a relief for programmers, reduces the opportunities to make errors in the own code, and enables rapid software development.

4.1 RMDB Message Format

The XML structure of messages exchanged by RMDBs is shown in Figure 3. It contains the document element `message` with four child elements. The message DTD is

[5]IBM's *MQSeries* and Microsoft's *MSMQ* currently are probably the most widely deployed messaging platforms.

```
<!ELEMENT  message   (commands,current,history,stack) >
<!ATTLIST  message
  history  ( yes | no )  "yes" >
<!ELEMENT  commands  ( rmdb* ) >
<!ELEMENT  current   ( rmdb ) >
<!ELEMENT  history   ( rmdb* ) >
<!ELEMENT  stack     ( data* ) >
<!ELEMENT  rmdb      ( arg* ) >
<!ATTLIST  rmdb
  name     CDATA          #REQUIRED
  data     IDREFS         #IMPLIED >
<!ELEMENT  arg       ANY >
<!ATTLIST  arg
  name     CDATA          #REQUIRED >
<!ELEMENT  data      ANY >
<!ATTLIST  data
  id       ID             #IMPLIED >
```

Figure 4: RMDB message DTD

listed in Figure 4, and the document element's child element types have the following semantics:

commands: A list of the following RMDBs which are scheduled to further process the request. They are registered by their JMS queue name, and may contain several arguments (wrapped in `arg` elements) to customize their behavior or tell them on which data stack entries to operate. The `command` element can be regarded as a stack, since the topmost command is always next to receive the message. Newly added commands are pushed on the top of the stack exclusively. This ensures that the message finally returns to the client and that the order of the RMDBs on the stack remains unchanged, as they might rely on that by passing information on the data stack. For the case of an unrecoverable error occurring in an RMDB so that normal processing is not possible anymore, the API function `sendException` is available, which interrupts the routing and returns an error message to the client.

current: The current command, representing the RMDB which is about to process the message. We use an own branch of the XML tree to enable the framework to easily access it. This means to read the arguments from it, or add references to freshly generated data stack entries. During the RMDB's epilog phase, just before the message is sent to the next RMDB, the current command is exchanged. The first element in the `commands` stack is moved here, while the replaced one is appended to the `history`.

history: This branch contains a list of all successfully processed commands. A developer could use this for logging or debugging purposes as well as for discovery of data on the data stack, learning by which RMDB it has been produced. If an application developer does not intend to access the `history`, the XML attribute `history="no"` can be set in the document element of the message to disable the RMDB history mechanism, in which case executed commands will be discarded instead of being added to the history.

stack: This is a data stack where all the information needed while processing a request can be stored. Its entries can be of type `string` or of arbitrarily complex XML structures. For retrieval, they can be addressed implicitly through the position on the stack, or explicitly by their ID. The ID is automatically generated by the framework as a unique number[6], and returned to the application logic for later reference.

```
<message history="yes">
  <commands>
    <rmdb name="Controller"/>
    <rmdb name="Kicker"/>
  </commands>
  <current>
    <rmdb name="Monitor"/>
  </current>
  <history>
    <rmdb name="Controller" data="i0"/>
    <rmdb name="Logger"/>
    <rmdb name="RequestHandler" data="i1">
      <arg name="cmd">request=0</arg>
    </rmdb>
  </history>
  <stack>
    <data id="i1">
      <!-- XML data omitted for brevity -->
    </data>
    <data id="i0">request=linux</data>
  </stack>
</message>
```

Figure 5: An example message

The complete XML DTD of RMDB messages is shown in Figure 5. Since we assume that messages are only produced and consumed by the RMDB framework itself, we currently do not validate the messages (ie, upon arrival at an RMDB, the message is only checked for wellformedness), but this behavior could be changed easily with switching on validation of the XML parser.

4.2 The Class Structure

Regular J2EE MDBs do not inherit any code from superclasses and are easily constructable by implementing two interfaces. However, the EJB specification 2.0 explicitly allows to use subclassing for MDBs. This gives us the possibility to write our own RMDB class, including the prolog and epilog code, the API to access the message, and a hook for later insertion of the application logic. The `RMDB` class implements all necessary methods of the interfaces `MessageDrivenBean` and `MessageListener` to fulfill the requirements set by the EJB container (see Figure 6).

The `RMDB` class also contains the abstract method `processMessage`, which has to be overridden by a subclass, because it will be called to execute the actual application logic of the RMDB. This is the only programming work a developer has to do to construct a new RMDB. The example classes `Logger` and `RequestHandler` shown in Figure 6 demonstrate the simplicity of RMDBs.

The `Controller` class is a special case. At least one instance of this bean must be present in every RMDB based

[6]Including a leading character to satisfy the XML naming constraints for ID attributes.

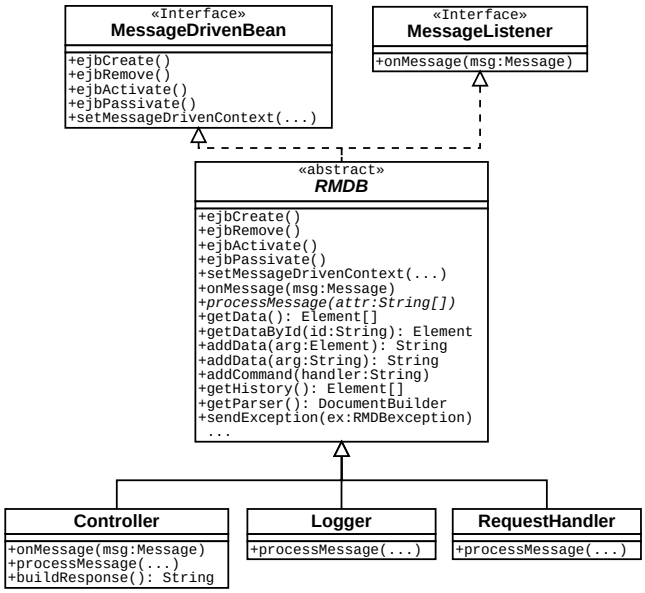


Figure 6: RMDB class structure

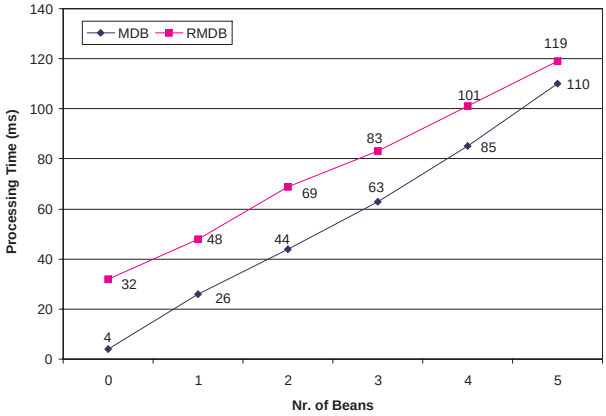


Figure 7: RMDB Performance

server and is located at the beginning and the end of every message path. It is designed to handle the direct communication with the client. Upon reception of a new request, the controller builds a new XML message frame, adds a default route to the command stack, and the actual request as a first data stack element. The default route always leads to the Controller a second time at the end of the path. This is the moment when the Controller finally removes the message frame and passes the response to the Client. It is the responsibility of the extra method `buildResponse` to return the correct response string. By default, it simply returns the top of stack element as a plain text string. This is a simple default strategy, and it is possible to override this method if a different behavior is required (for example for passing complex data structures as responses).

5. PERFORMANCE

Programming RMDBs means programming on a very high level of abstraction. This is very comfortable, but there are a lot of lower software layers (the Java Virtual Machine, the J2EE application server, and the RMDB class) which all consume CPU time and other system resources. To get an idea how RMDBs perform, we measured their minimal processing time compared to the simpler MDBs (which provide the foundation for our RMDBs). Figure 7 shows the chart of the benchmark measurement. The X-axis shows zero to five Beans. On the lower line there are just MDBs, the upper one has been measured using RMDBs containing the full routing and XML messaging mechanisms. Both types of beans do not contain any application logic.

The Y-axis shows the time for a round trip from a client application through a variable number of beans and back to the client. For the RMDBs as well as for the MDBs, the total time depends linearly on the number of beans in the path. In the measurement of the RMDBs, the controller bean has not been counted as a working bean. Hence the value in the chart for zero RMDBs is 32ms and contains the time the controller needs to set up the initial message and communicate with the client. The client application also consumes about 4ms in both test cases.

The measurement has been made on Intel Pentium III processor machine with 500MHz and 384MByte RAM running Linux 2.2.14 and the BEA Weblogic 6.0sp1 server. The application server has been advised to create a sufficient amount of beans at deployment time, so the time for instantiating and initializing them did not affect the measured delay. Besides that, no further optimizations have been made for the application server, but there are plenty of parameters to tweak in the Weblogic server and in the operating system.

From the measured data, it can be concluded that the average time needed by a single RMDB is 17.4 milliseconds to fulfill all the required parsing, routing and communication work. This is to see as a minimal response time under very low work load. But for applications with very high work load the system is designed to achieve a good load balancing to keep the response time acceptable.

Naturally, performance for RMDB depends on many factors, the most important one being the size of the messages, ie the size of the data being sent through the RMDB message path. If there are big and complexly structured messages, parsing and serializing them will be more expensive than in our example, which used a very simple message structure. However, XML parsing and serialization may be neglected if the application logic is very complex and time-consuming and therefore predominates the overhead caused by the RMDB framework and the lower layers (JVM and J2EE).

6. CONCLUSIONS

We started thinking about an abstraction for routing messages through a number of software component as a result of our work on the XLinkbase system. When we started our work on the XLinkbase server, JMS was not part of J2EE, and we were very pleased with the inclusion of the messaging service. However, we discovered that asynchronous messaging as provided by MDBs was not as powerful an abstraction as we were looking for. In an effort to create a generic solution to the problem of a message flow through a complex

set of software components, we invented the concept of the *Routed Message-Driven Bean (RMDB)*, which proved to be very useful in our application scenario.

It was our goal to separate the generic concept of RMDBs from our XLinkbase application, and we believe that we have succeded in creating an abstraction which might prove useful in a wide variety of application involving asynchronous messaging. It would be pretentious to think of RMDBs as something to be included in the next release of J2EE, but we believe that something similar to it, ie supporting a more abstract way of messaging as through the rather simple mechanisms of JMS queues and topics, would be an interesting idea. However, there are still some open issues, which we discuss shortly in the following section.

7. CHALLENGES AND FUTURE WORK

So far, we have concentrated on making the RMDB framework as generic and flexible as possible, so that it may be re-used in similar scenarios than our linkbase application. However, one important aspect we have not yet implemented is the issue of reliability. While JMS guarantees the delivery of messages, there is no transaction concept. We are currently investigating how to integrate J2EE's JTA service with our RMDB abstraction for implementing transactional semantics, ideally being able to make the whole RMDB processing of a request one transaction. Our plan is to extend the RMDB API and the RMDB message format with support for transactions, so that critical applications may choose between the more efficient and faster messaging service as provided by JMS only, and a more expensive, but more reliable transactional variant implemented by using JMS and JTA services.

Another issue we would like to investigate is the support for conditional paths within the messages, where the routing of RMDB messages is based on the result of computations earlier in the path. Again, this would require changes to the API as well as to the message format, but we believe that conditional routing could be useful in many scenarios, the most widely known being the handling of exceptions (ie, error conditions) in a more general way than only providing one exception handler.

We currently use an XML DTD to describe the RMDB message format. We are currently looking into using XML Schema [2, 13] for the schema definition, which would enable us to specify a better formal model of the message format. In particular, XML Schema's datatype concept could be used to make the RMDB code lighter by moving more data checking into the validation of the messages. However, this would require using a schema-validating XML parser, and we still have to look into the performance implications of such a step.

In our current RMDB framework, the calculation of the processing graph has to be done by the controller, and is completely handled by the application logic. One could easily think of making the RMDB model even more abstract by specifying certain constraints and conditions for each RMDB available to the controller, and then having the RMDB code compute the actual processing graph based on the incoming request and the RMDB specifications. This, however, would be a very hard task to solve generically, which is the main reason why we did not put any effort into it.

8. ACKNOWLEDGEMENTS

We would like to thank BEA Systems Switzerland for providing us with an extended trial license for their WebLogic application server. We would also like to thank Yves Langisch for providing the foundations of the XLinkbase EJB architecture.

9. REFERENCES

[1] Tim Berners-Lee. The World Wide Web. In *Proceedings of the 3rd Joint European Networking Conference*, Innsbruck, Austria, May 1992.

[2] Paul V. Biron and Ashok Malhotra. XML Schema Part 2: Datatypes. World Wide Web Consortium, Recommendation REC-xmlschema-2-20010502, May 2001.

[3] Susan Cheung and Vlada Matena. Java Transaction API (JTA) — Version 1.0.1. Technical report, Sun Microsystems, April 1999.

[4] Steven J. DeRose, Eve Maler, and David Orchard. XML Linking Language (XLink) Version 1.0. World Wide Web Consortium, Recommendation REC-xlink-20010627, June 2001.

[5] Roy T. Fielding, Jim Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. Hypertext Transfer Protocol — HTTP/1.1. Internet proposed standard RFC 2616, June 1999.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, January 1995.

[7] Mark Hapner, Rich Burridge, Rahul Sharma, and Joseph Fialli. Java Message Service — Version 1.0.2b. Technical report, Sun Microsystems, August 2001.

[8] International Organization for Standardization. Information technology — SGML Applications — Topic Maps. ISO/IEC 13250, 2000.

[9] Peter J. Nürnberg and John J. Leggett. A Vision for Open Hypermedia Systems. *Journal of Digital Information*, 1(2), 1997.

[10] Steve Pepper and Graham Moore. XML Topic Maps (XTM) 1.0. TopicMaps.Org Specification xtm1-20010806, August 2001.

[11] Ed Roman. *Mastering Enterprise JavaBeans and the Java 2 Platform, Enterprise Edition*. John Wiley & Sons, New York, September 1999.

[12] Bill Shannon. Java 2 Platform Enterprise Edition Specification, v1.3. Technical report, Sun Microsystems, July 2001.

[13] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML Schema Part 1: Structures. World Wide Web Consortium, Recommendation REC-xmlschema-1-20010502, May 2001.

[14] Luke Welling and Laura Thomson. *PHP and MySQL Web Development*. Sams, Indianapolis, Indiana, March 2001.

[15] Erik Wilde. *Wilde's WWW — Technical Foundations of the World Wide Web*. Springer-Verlag, Berlin, Germany, November 1998.

[16] Erik Wilde and David Lowe. From Content-Centered Publishing to a Link-Based View of Information Resources. In *Proceedings of the 33rd Hawaii International Conference on System Sciences*, Maui, Hawaii, January 2000. IEEE Computer Society Press.

Anhang A

Inhalt der CDROM

A.1 XLinkbaseServer

Im Verzeichnis *XLinkbaseServer* befindet sich der komplette Source-Code wie er in dieser Diplomarbeit entstanden ist. Das Unterverzeichnis *xlinkbaseBeansIteration3* enthält die letzte Iteration und das Unterverzeichnis *alteVersionen* enthält frühere, zum Teil nicht korrekt lauffähige Vorabversionen. Der gesamte Code wurde mit dem *Borland JBuilder 4 Foundation* entwickelt. Auf der CD befindet sich jeweils das ganze Projektverzeichnis wie es vom JBuilder angelegt wird:

xlinkbase.jpr Diese Datei ist das Projektfile und enthält die ganze Konfiguration für den JBuilder.

.../src Hier drin ist der eigentliche Source-Code, angeordnet in der Package Struktur und alle wichtigen Deployment-Files im Unterverzeichnis *META-INF*.

.../classes Das Zielverzeichnis bei der Kompilation.

.../bak Alte Sicherungskopien. Diese werden vom JBuilder automatisch angelegt und können bei Bedarf zurück kopiert werden.

.../jars In diesem Verzeichnis werden die *Java Archiv Dateien* erstellt. Bereit zum *deploying* im Application Server.

.../doc Die Ausgabe des *javadoc* Generators wird hier abgelegt. Es beinhaltet die API-Dokumentation im HTML-Format.

Makefile Mit diesem Makefile können automatisch jar-Dateien erstellt werden und zum "Deployenän den richtigen Ort im Weblogic oder JBoss Server kopiert werden. Das Kompilieren muss mit dem JBuilder gemacht werden, alle weiteren Schritte, inklusive Erstellen der Dokumentation kann mit *make* gemacht werden. Der Prefix *WL* in den Targets steht jeweils für den Weblogic Server, der Prefix *JB* für JBoss. Bei Wiederverwendung dieses Makefiles müssen im Kopf der Datei die korrekten Verzeichnisse gesetzt werden.

Um die Projekte im JBuilder kompilieren zu können, werden noch einige Bibliotheken benötigt:

- Der Xerces XML Parser. Dieser befindet sich ebenfalls auf der CD. Es muss die Datei *xerces.jar* als Bibliothek eingebunden werden.

Für den Weblogic Server

- *tools.jar* und

- *ejb20.jar* aus dem *lib*-Verzeichnis des Servers

Für den JBoss Server

- *jboss-j2ee.jar* und
- *servlet.jar* aus dem *lib*-Verzeichnis des Servers

Wie im JBuilder diese Bibliotheken erstellt und in Projekt übernommen werden, kann im Hilfe-Programm nachgelesen werden.

A.2 Application Server Installationen

Unter *installationen* befinden sich die beiden Application Server *Bea Weblogic 6.0sp1* und *JBoss-2.4.0 beta mit Tomcat 3.2.2*. Auf meinem Linux-Rechner haben sich diese beiden Server unter */opt* installiert gehabt. Für weiteren problemlosen Betrieb ist es ratsam, diese wieder so einzurichten. Zudem ist eine aktuelle Version des JDK nötig. Der Bea Weblogic Server bringt die Version 1.3.0 gleich selber mit. Diese habe ich auch gerade für den JBoss verwendet. Ich musste dazu die Environment Variable `JAVA_HOME` auf */opt/bea/jdk130* setzen.

A.2.1 Bea Weblogic

.../bea/Xlinkbase/xlx Hier drin befinden sich alle *xlx*-Dateien die vom RequestHandler Bean zurück geliefert werden.

.../bea/lib Alle Bibliotheken in Form von jar-Files müssen sich hier befinden, auch die Zusätzlichen Dateien *ejb20.jar* und *xerces.jar*.

.../bea/config/xlinkbase Hier ist das Hauptkonfigurationsfile *config.xml* und die Startskripte. *config.xml* kann bei ausgeschaltetem Server von Hand geändert werden oder einfacher bei laufendem Server im Browser unter *http://localhost:7001/console*. Mit Username *system* und Passwort *xlinkbase* kann man so komfortabel die meisten Administrationsaufgaben erledigen. Vor dem Starten des Servers mit *startXlinkbase.sh* sollten mit *setEnv.sh* alle Umgebungsvariablen richtig gesetzt werden. Das Passwort für den Start ist wieder: *xlinkbase*.

.../bea/config/xlinkbase/applications Das ist der Ort wo die EJBs in Form von jar-Files hin kopiert werden müssen. Der Application Server erkennt sie automatisch und bindet sie in den laufenden Server ein.

.../bea/config/xlinkbase/applications/DefaultWebApp_xlinkbaseServer/WEB-INF/classes In diesem Verzeichnis wird das Kicker Servlet abgelegt.

.../bea/config/xlinkbase/applications/DefaultWebApp_ xlinkbaseServer/WEB-INF/web.xml Das Deployment-File für Servlets und andere Web Applications.

A.2.2 JBoss-Tomcat Server

Der JBoss ist nur ein EJB-Container und unterstützt selber keine Servlets. Darum wird er oft mit einer anderen Servlet-Engine betrieben, wie in meinem Fall mit Tomcat.

.../JBoss-2.4.0_Tomcat-3.2.2/jboss/conf/tomcat Alle Konfigurationen des Servers müssen in diesem Verzeichnis vorgenommen werden, falls JBoss zusammen mit Tomcat gestartet wird. Andernfalls gibt es noch das Verzeichnis *default*. Die Erstellung von neuen JMS-Queues kann in der Datei *jbossmq.xml* gemacht werden.

.../JBoss-2.4.0_Tomcat-3.2.2/jboss/bin Hier sind die Startskripte. Es sollte immer *run_with_tomcat.sh* verwendet werden, damit Tomcat in der selben Virtual Machine gestartet wird.

.../JBoss-2.4.0_Tomcat-3.2.2/jboss/deploy EJB-JAR Dateien in diesem Verzeichnis werden von JBoss Server automatisch eingebunden.

.../JBoss-2.4.0_Tomcat-3.2.2/jboss/admin Mit Hilfe des Monitors unter diesem Verzeichnis kann Einblick in den laufenden JBoss Server genommen werden, ähnlich wie mit der Browser-Konsole des Bea Weblogic Server.

Den JBoss Application Server habe ich nur oberflächlich angeschaut. Die Beans die ich in Iteration 3 programmiert habe funktionieren auch nicht ganz richtig mit dieser JBoss-Installation. Es ist zu bedenken, dass ich hier eine Beta Version ausprobiert habe, die noch nicht alle Features unterstützt die in einem EJB 2.0 Server vorhanden sein sollten. Dieses Open Source Projekt macht aber sehr schnell Fortschritte, so dass sicher in naher Zukunft ein reifer Application Server verfügbar sein wird.

A.3 Test Code

Für Test des Xlinkbase Servers und Messung der Performance habe ich einige Programme geschrieben, die unter *testCode* archiviert sind. Eine ganz einfache Funktionskontrolle habe ich jeweils mit der HTML Datei *testPost.html* in einem beliebigen Browser machen können. Diese Enthält ein Formular das mit der HTTP-Post Methode einen Request an das Kicker Servlet sendet. Die Rückgabe wird dann einfach als *plain text* im Browser angezeigt. Soll der XLinkbase Server von einem anderen Rechner aus getestet werden, muss im HTML-Code *localhost* durch den richtigen Servername ersetzt werden.

A.4 Software Tools

Alle wichtigen Programme, die ich während meiner Diplomarbeit verwendet habe, sind unter *softwareTools* auf die CD gebrannt. Das sind die beiden Application Server (*weblogic* und *jboss*), der JBuilder 4 Foundation (*jb4*), der Xerces XML Parser (*xerces*) und Ant für die Kompilierung von Java-Sourcen (*ant*). Allfällige Lizenzen für die Programme müssen selber beschafft werden. Für den Weblogic Server gibt es 30-Tage Trial Lizenzen, für den JBuilder kann gratis eine Seriennummer angefordert werden und alle anderen Programme sind frei verwendbar.

A.5 Bericht

Die Dokumentation dieser Diplomarbeit liegt unter *bericht* in den Formaten PostScript (PS), PortableDocumentFormat (PDF) und HTML vor. Ebenso befindet sich der LaTeX Code sowie das PowerPoint-File der Präsentation in diesem Verzeichnis.

com.xlinkbase.ejb

Class XlinkbaseBean

```
java.lang.Object
  |
  +--com.xlinkbase.ejb.XlinkbaseBean
```

All Implemented Interfaces:

javax.ejb.EnterpriseBean, javax.ejb.MessageDrivenBean, javax.jms.MessageListener, java.io.Serializable

Direct Known Subclasses:

Controller, Logger, Monitor, RequestHandler

---

public abstract class **XlinkbaseBean**
extends java.lang.Object
implements javax.ejb.MessageDrivenBean, javax.jms.MessageListener

XlinkbaseBean is a abstract Superclass that can be used to implement a Xlinkbase Server. Build a subclass and override the method processMessage to fill it with the business logic.

Diploma Thesis 2001.21
Computer Engineering and Networks Laboratory
Swiss Federal Institute of Technology, Zürich

See Also:

Serialized Form

Field Summary

static int	**DEBUG** Constant for the importance of a log message.
static int	**ERROR** Constant for the importance of a log message.
static int	**NOTE** Constant for the importance of a log message.
static int	**WARNING** Constant for the importance of a log message.

Constructor Summary

XlinkbaseBean()	

Method Summary

void	**addCommand**(java.lang.String handler, java.lang.String[] attributes) Adds a new command to the processing path.
java.lang.String	**addData**(org.w3c.dom.Element arg) Pushes a new data element on top of the data stack.
java.lang.String	**addData**(java.lang.String arg) Pushes a new data element on top of the data stack.
void	**ejbActivate**() Method of the MessageDrivenBean interface.
void	**ejbCreate**() Method of the MessageDrivenBean interface.
void	**ejbPassivate**() Method of the MessageDrivenBean interface.
void	**ejbRemove**() Method of the MessageDrivenBean interface.
java.lang.String[]	**getAttributes**() Returns the Attributes which have been provided with the invocation present bean.
org.w3c.dom.Element[]	**getData**() Returns the complete data stack.
org.w3c.dom.Element	**getDataById**(java.lang.String id) Searches the data stack for a data element with the id string provided and returns it.
org.w3c.dom.Element[]	**getHistory**() Returns a list of already executed commands as an array of elements.
javax.xml.parsers.DocumentBuilder	**getParser**() Returns a DocumentBuilder Object.
java.lang.String	**getSerializedForm**(org.w3c.dom.Document doc) Helper method to serialize a XML document into a string.
void	**log**(java.lang.String s) Prints a log message to stdout.
void	**log**(java.lang.String s, int level) Prints a log message to stdout.
boolean	**messageContainsFrame**() Checks whether there's an XML message frame in the present message.
void	**onMessage**(javax.jms.Message msg) Method of the MessageListener interface.
abstract void	**processMessage**(java.lang.String[] attributes) This is an abstract method, so it must be implemented in your subclass.
void	**readMessage**(javax.jms.Message msg) Reads a received JMS Message.
void	**sendException**(XlinkbaseException exception) If something bad happens, an exception can be sent using this method.
void	**sendMessage**(java.lang.String stringMessage) Sends the message to the next RMDB in the processing path.

void	**setLoglevel**(int arg) Changes the current log level.
void	**setLogPrefix**(java.lang.String arg) Sets prefix arg before each log entry.
void	**setMessageDrivenContext**(javax.ejb.MessageDrivenContext mctx) Method of the MessageDrivenBean interface.
void	**setupMessageFrame**(javax.jms.Message msg) This method builds a complete XML message frame and fills in the default route and the request from the received message as a data element.
void	**showMessage**() Dumps the received message on stdout.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

ERROR

```
public static final int ERROR
```

Constant for the importance of a log message.

WARNING

```
public static final int WARNING
```

Constant for the importance of a log message.

NOTE

```
public static final int NOTE
```

Constant for the importance of a log message.

DEBUG

```
public static final int DEBUG
```

Constant for the importance of a log message.

Constructor Detail

XlinkbaseBean

```
public XlinkbaseBean()
```

Method Detail

setLogPrefix

```
public void setLogPrefix(java.lang.String arg)
```

Sets prefix arg before each log entry. Defaults to "MDB:".

Parameters:

`arg` - the prefix string

---

setLoglevel

```
public void setLoglevel(int arg)
```

Changes the current log level.

Parameters:

`arg` - log level: OFF, ERROR, WARNING, NOTE, DEBUG

---

getParser

```
public javax.xml.parsers.DocumentBuilder getParser()
```

Returns a DocumentBuilder Object. Use this to create new XML documents or parse and serialize existing ones.

---

log

```
public void log(java.lang.String s)
```

Prints a log message to stdout. Uses default log level NOTE.

Parameters:

`s` - the message string

---

log

```
public void log(java.lang.String s,
                int level)
```

Prints a log message to stdout.

Parameters:

`s` - the message string

`level` - log level: ERROR, WARNING, NOTE, DEBUG

---

ejbActivate

```
public void ejbActivate()
```

Method of the `MessageDrivenBean` interface. Called by the EJB Container. Don't use or override this method in your subclass.

---

ejbRemove

```
public void ejbRemove()
```

Method of the `MessageDrivenBean` interface. Called by the EJB Container. Don't use or override this method in your subclass.

Specified by:

`ejbRemove` in interface `javax.ejb.MessageDrivenBean`

---

ejbPassivate

```
public void ejbPassivate()
```

Method of the `MessageDrivenBean` interface. Called by the EJB Container. Don't use or override this method in your subclass.

---

ejbCreate

```
public void ejbCreate()
            throws javax.ejb.CreateException
```

Method of the `MessageDrivenBean` interface. Called by the EJB Container. Don't use or override this method in your subclass.

Here the JNDI context and the Java Messaging Service is initialized, the parser is created and the environment variables are read in.

---

setMessageDrivenContext

```
public void setMessageDrivenContext(javax.ejb.MessageDrivenContext mctx)
```

Method of the `MessageDrivenBean` interface. Called by the EJB Container. Don't use or override this method in your subclass.

Specified by:

`setMessageDrivenContext` in interface `javax.ejb.MessageDrivenBean`

onMessage

```
public void onMessage(javax.jms.Message msg)
```

Method of the `MessageListener` interface. Called by the EJB Container. Don't use or override this method in your subclass.

This method calls the `processMessage()`. Before and after that all the work is done that enhances the Message Driven Beans (MDB) to the Routed Message Driven Beans (RMDB).

Specified by:
`onMessage` in interface `javax.jms.MessageListener`

---

getAttributes

```
public java.lang.String[] getAttributes()
                                  throws XlinkbaseException
```

Returns the Attributes which have been provided with the invocation present bean. These are also given as parameter to the `processMessage` method, so a direct call to this method should not be necessary.

---

processMessage

```
public abstract void processMessage(java.lang.String[] attributes)
                             throws XlinkbaseException
```

This is an abstract method, so it must be implemented in your subclass. It is supposed to contain the business logic of the RMDB. It is called by the framework each time a message arrives.
Parameters:
`attributes` - optional attributes used for steering the beans work.
See Also:
`addCommand(String, String[])`

---

getHistory

```
public org.w3c.dom.Element[] getHistory()
                                  throws XlinkbaseException
```

Returns a list of already executed commands as an array of elements. If the history is turned off in this message `null` is returned.

---

getData

```
public org.w3c.dom.Element[] getData()
                               throws XlinkbaseException
```

Returns the complete data stack. Each stack entry is wrapped in an XML element with name `data` and filled up in an array. The "top of stack" element is in the first place of the array (item 0).

---

getDataById

```
public org.w3c.dom.Element getDataById(java.lang.String id)
                                throws XlinkbaseException
```

Searches the data stack for a data element with the id string provided and returns it. If there's no stack entry with that id `null` is returned.

Parameters:

`id` - id string of the requested data element

See Also:

`addData(String)`, `addData(Element)`

---

addData

```
public java.lang.String addData(java.lang.String arg)
                         throws XlinkbaseException
```

Pushes a new data element on top of the data stack. The argument is stored as a "XML text node" an inserted in a new data element. A unique id String is calculated, added to the data node and return to the caller.

Parameters:

`arg` - string to be stored on the data stack

Returns:

unique id string to retrieve the stored data

See Also:

`getDataById(String)`

---

addData

```
public java.lang.String addData(org.w3c.dom.Element arg)
                         throws XlinkbaseException
```

Pushes a new data element on top of the data stack. The argument can be an element from any XML document. It is recursively imported into a new data node and numbered with a unique id.

Parameters:

`arg` - element to be stored on the data stack

Returns:

unique id string to retrieve the stored data

See Also:

`getDataById(String)`

---

addCommand

```
public void addCommand(java.lang.String handler,
                       java.lang.String[] attributes)
                throws XlinkbaseException
```

Adds a new command to the processing path. It is executed after the present bean finishes its work unless you add more commands. Every command is stored on a stack an therefore processed in reversed order as this method is being called. The command name must match a live message queue and a RMDB has to be deployed to listen on this queue.

Parameters:

`handler` - the command name.

`attributes` - an array of strings, containing parameters for the called RMDB

---

readMessage

```
public void readMessage(javax.jms.Message msg)
                 throws XlinkbaseException
```

Reads a received JMS Message. This method is automatically called by the framework and must not be called by the user.

---

getSerializedForm

```
public java.lang.String getSerializedForm(org.w3c.dom.Document doc)
                                   throws XlinkbaseException
```

Helper method to serialize a XML document into a string.

Parameters:

`doc` - a XML Document

Returns:

a string containing the serialized XML code

Throws:

`XlinkbaseException` - if the document is invalid or the serializer is not initialized.

---

sendMessage

```
public void sendMessage(java.lang.String stringMessage)
                 throws XlinkbaseException
```

Sends the message to the next RMDB in the processing path. This method is automatically called by the framework and must not be called by the user.

---

sendException

```
public void sendException(XlinkbaseException exception)
```

If something bad happens, an exception can be sent using this method. It tries to find a reasonable recipient for this error message, but if all fails it stops the processing of the request.

Parameters:

`exception` - an exception containing a description of what went wrong

---

setupMessageFrame

```
public void setupMessageFrame(javax.jms.Message msg)
                       throws XlinkbaseException
```

This method builds a complete XML message frame and fills in the default route and the request from the received message as a data element. It should only be called at the very beginning of the processing path which is usually done by the Controller RMDB.

Parameters:

`msg` - the message sent by the client containing the request.

---

messageContainsFrame

```
public boolean messageContainsFrame()
```

Checks whether there's an XML message frame in the present message.

Returns:

`true` if there's a message frame, else `false`

---

showMessage

```
public void showMessage()
```

Dumps the received message on stdout. Useful for debugging.

---

com.xlinkbase.ejb
Class XlinkbaseException

```
java.lang.Object
  |
  +--java.lang.Throwable
        |
        +--java.lang.Exception
              |
              +--com.xlinkbase.ejb.XlinkbaseException
```

All Implemented Interfaces:
java.io.Serializable

---

public class **XlinkbaseException**
extends java.lang.Exception

This is a simple exception class used by the Routed Message Driven Beans and its framework.

Diploma Thesis 2001.21
Computer Engineering and Networks Laboratory
Swiss Federal Institute of Technology, Zürich

See Also:
Serialized Form

Constructor Summary

XlinkbaseException(java.lang.String e)

Methods inherited from class java.lang.Throwable
fillInStackTrace, getLocalizedMessage, getMessage, printStackTrace, printStackTrace, printStackTrace, toString

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

XlinkbaseException

```
public XlinkbaseException(java.lang.String e)
```

Parameters:

e - The error Message

---

com.xlinkbase.ejb.Controller

Class Controller

```
java.lang.Object
  |
  +--com.xlinkbase.ejb.XlinkbaseBean
        |
        +--com.xlinkbase.ejb.Controller.Controller
```

All Implemented Interfaces:

javax.ejb.EnterpriseBean, javax.ejb.MessageDrivenBean, javax.jms.MessageListener, java.io.Serializable

---

public class **Controller**
extends XlinkbaseBean
implements javax.ejb.MessageDrivenBean, javax.jms.MessageListener

Controller is special implementation of the Xlinkbase Superclass. It is designed to be the first and the last RMDB in a processing path and to handle the direct communication with the client.

Diploma Thesis 2001.21
Computer Engineering and Networks Laboratory
Swiss Federal Institute of Technology, Zürich

See Also:

Serialized Form

Fields inherited from class com.xlinkbase.ejb.XlinkbaseBean
DEBUG, ERROR, NOTE, WARNING

Constructor Summary

Controller()	

Method Summary

java.lang.String	**buildResponse**() This method filters the plain response out of the XML message by simply returning the content of the data element on top of the stack.
void	**onMessage**(javax.jms.Message msg) This method overrides the implemenation in the XlinkbaseBean class.
void	**processMessage**(java.lang.String[] attribs) This method is empty, because the Controller is not supposed to do any business logic.

Methods inherited from class com.xlinkbase.ejb.XlinkbaseBean

addCommand, addData, addData, ejbActivate, ejbCreate, ejbPassivate, ejbRemove, getAttributes, getData, getDataById, getHistory, getParser, getSerializedForm, log, log, messageContainsFrame, readMessage, sendException, sendMessage, setLoglevel, setLogPrefix, setMessageDrivenContext, setupMessageFrame, showMessage

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Methods inherited from interface javax.ejb.MessageDrivenBean

ejbRemove, setMessageDrivenContext

Constructor Detail

Controller

public **Controller**()

Method Detail

onMessage

```
public void onMessage(javax.jms.Message msg)
```

This method overrides the implemenation in the XlinkbaseBean class. When the Controller is invocated by a client, this method sets up a message frame and a default route and then kicks off the request handling by the other RMDBs. After the response returms to the Controller the onMessage method removes the message frame and sends the response as plain text to the client.

Specified by:

`onMessage` in interface `javax.jms.MessageListener`

Overrides:

`onMessage` in class `XlinkbaseBean`

---

buildResponse

```
public java.lang.String buildResponse()
                                throws XlinkbaseException
```

This method filters the plain response out of the XML message by simply returning the content of the data element on top of the stack. Could be overridden if other functionality is required.

---

processMessage

```
public void processMessage(java.lang.String[] attribs)
                    throws XlinkbaseException
```

This method is empty, because the Controller is not supposed to do any business logic.

Overrides:

`processMessage` in class `XlinkbaseBean`

Following copied from class: `com.xlinkbase.ejb.XlinkbaseBean`

Parameters:

`attributes` - optional attributes used for steering the beans work.

See Also:

`XlinkbaseBean.addCommand(String, String[])`

---

com.xlinkbase.ejb.Logger

Class Logger

```
java.lang.Object
  |
  +--com.xlinkbase.ejb.XlinkbaseBean
        |
        +--com.xlinkbase.ejb.Logger.Logger
```

All Implemented Interfaces:

javax.ejb.EnterpriseBean, javax.ejb.MessageDrivenBean, javax.jms.MessageListener, java.io.Serializable

---

public class **Logger**
extends XlinkbaseBean
implements javax.ejb.MessageDrivenBean, javax.jms.MessageListener

This is an implemention of the XlinkbaseBean class. It's a Routed Message Driven Beans used to do Logging.

Diploma Thesis 2001.21
Computer Engineering and Networks Laboratory
Swiss Federal Institute of Technology, Zürich

See Also:

Serialized Form

Fields inherited from class com.xlinkbase.ejb.XlinkbaseBean
DEBUG, ERROR, NOTE, WARNING

Constructor Summary

Logger()	

Method Summary

void	**processMessage**(java.lang.String[] attributes) Implements the abstract method of the superclass XlinkbaseBean.

Methods inherited from class com.xlinkbase.ejb.XlinkbaseBean
addCommand, addData, addData, ejbActivate, ejbCreate, ejbPassivate, ejbRemove, getAttributes, getData, getDataById, getHistory, getParser, getSerializedForm, log, log, messageContainsFrame, onMessage, readMessage, sendException, sendMessage, setLoglevel, setLogPrefix, setMessageDrivenContext, setupMessageFrame, showMessage

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Methods inherited from interface javax.ejb.MessageDrivenBean
ejbRemove, setMessageDrivenContext

Methods inherited from interface javax.jms.MessageListener
onMessage

Constructor Detail

Logger

```
public Logger()
```

Method Detail

processMessage

```
public void processMessage(java.lang.String[] attributes)
                    throws XlinkbaseException
```

Implements the abstract method of the superclass XlinkbaseBean. It contains the Application Logic.

Overrides:

`processMessage` in class `XlinkbaseBean`

Following copied from class: `com.xlinkbase.ejb.XlinkbaseBean`

Parameters:

`attributes` - optional attributes used for steering the beans work.

See Also:

`XlinkbaseBean.addCommand(String, String[])`

---

com.xlinkbase.ejb.RequestHandler

Class RequestHandler

```
java.lang.Object
  |
  +--com.xlinkbase.ejb.XlinkbaseBean
        |
        +--com.xlinkbase.ejb.RequestHandler.RequestHandler
```

All Implemented Interfaces:

javax.ejb.EnterpriseBean, javax.ejb.MessageDrivenBean, javax.jms.MessageListener, java.io.Serializable

---

public class **RequestHandler**
extends XlinkbaseBean
implements javax.ejb.MessageDrivenBean, javax.jms.MessageListener

This is an implemention of the XlinkbaseBean class. It's a Routed Message Driven Beans used to do read xlx files from disk. This bean must be deployed on the computer where the xlx files are.

Diploma Thesis 2001.21
Computer Engineering and Networks Laboratory
Swiss Federal Institute of Technology, Zürich

See Also:

Serialized Form

Fields inherited from class com.xlinkbase.ejb.XlinkbaseBean
DEBUG, ERROR, NOTE, WARNING

Constructor Summary

RequestHandler()	

Method Summary

void	**processMessage**(java.lang.String[] attributes) Implements the abstract method of the superclass XlinkbaseBean.

Methods inherited from class com.xlinkbase.ejb.XlinkbaseBean
addCommand, addData, addData, ejbActivate, ejbCreate, ejbPassivate, ejbRemove, getAttributes, getData, getDataById, getHistory, getParser, getSerializedForm, log, log, messageContainsFrame, onMessage, readMessage, sendException, sendMessage, setLoglevel, setLogPrefix, setMessageDrivenContext, setupMessageFrame, showMessage

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Methods inherited from interface javax.ejb.MessageDrivenBean
ejbRemove, setMessageDrivenContext

Methods inherited from interface javax.jms.MessageListener
onMessage

Constructor Detail

RequestHandler

```
public RequestHandler()
```

Method Detail

processMessage

```
public void processMessage(java.lang.String[] attributes)
                    throws XlinkbaseException
```

Implements the abstract method of the superclass XlinkbaseBean. It contains the Application Logic.

Overrides:

`processMessage` in class `XlinkbaseBean`

Following copied from class: `com.xlinkbase.ejb.XlinkbaseBean`

Parameters:

`attributes` - optional attributes used for steering the beans work.

See Also:

`XlinkbaseBean.addCommand(String, String[])`

---

com.xlinkbase.ejb.Monitor

Class Monitor

```
java.lang.Object
  |
  +--com.xlinkbase.ejb.XlinkbaseBean
        |
        +--com.xlinkbase.ejb.Monitor.Monitor
```

All Implemented Interfaces:

javax.ejb.EnterpriseBean, javax.ejb.MessageDrivenBean, javax.jms.MessageListener, java.io.Serializable

---

public class **Monitor**
extends XlinkbaseBean
implements javax.ejb.MessageDrivenBean, javax.jms.MessageListener

This is an implemention of the XlinkbaseBean class. It's a Routed Message Driven Beans used to do monitor the history.

Diploma Thesis 2001.21
Computer Engineering and Networks Laboratory
Swiss Federal Institute of Technology, Zürich

See Also:

Serialized Form

Fields inherited from class com.xlinkbase.ejb.XlinkbaseBean
DEBUG, ERROR, NOTE, WARNING

Constructor Summary

Monitor()	

Method Summary

void	**processMessage**(java.lang.String[] attributes) Implements the abstract method of the superclass XlinkbaseBean.

Methods inherited from class com.xlinkbase.ejb.XlinkbaseBean
addCommand, addData, addData, ejbActivate, ejbCreate, ejbPassivate, ejbRemove, getAttributes, getData, getDataById, getHistory, getParser, getSerializedForm, log, log, messageContainsFrame, onMessage, readMessage, sendException, sendMessage, setLoglevel, setLogPrefix, setMessageDrivenContext, setupMessageFrame, showMessage

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Methods inherited from interface javax.ejb.MessageDrivenBean
ejbRemove, setMessageDrivenContext

Methods inherited from interface javax.jms.MessageListener
onMessage

Constructor Detail

Monitor

```
public Monitor()
```

Method Detail

processMessage

```
public void processMessage(java.lang.String[] attributes)
                    throws XlinkbaseException
```

Implements the abstract method of the superclass XlinkbaseBean. It contains the Application Logic.

Overrides:

`processMessage` in class `XlinkbaseBean`

Following copied from class: `com.xlinkbase.ejb.XlinkbaseBean`

Parameters:

`attributes` - optional attributes used for steering the beans work.

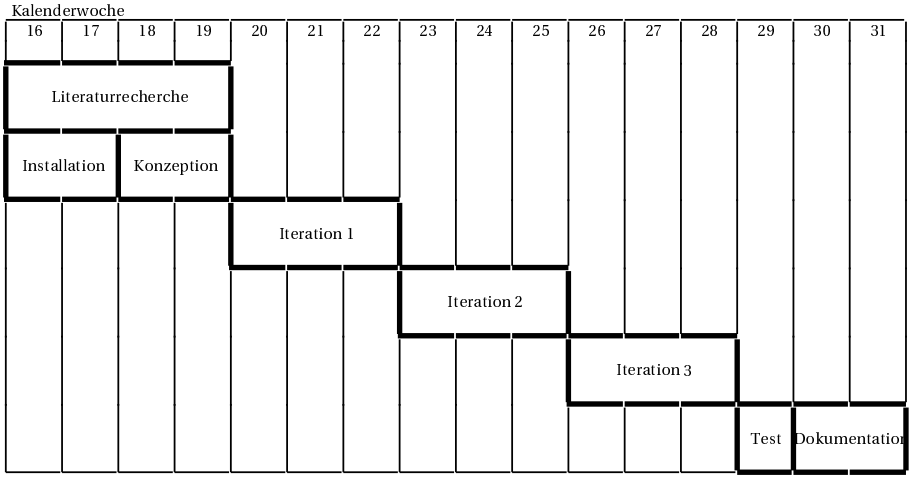
See Also:

`XlinkbaseBean.addCommand(String, String[])`

Anhang C

Zeitplan

Zeitplan



Details

Literaturrecherche:

1. Einarbeitung in die Vorgängerprojekte
2. Einarbeitung in die benötigten Technologien
3. Inspiration aus anderen Projektion

Installation:

1. Einrichtung des Arbeitsplatzes
2. Installation und Konfiguration des Application Servers
3. Installation der Entwicklungsumgebung

Konzeption:

1. Grobe Definition des Routingalgorithmus
2. Festlegung der zu verwendenden Technologien

Iterationsschritte 1-3:

1. Definition der Features die in dieser Iteration implementiert werden sollen
2. Implementation
3. Test
4. Dokumentation der Iteration

Test:

1. Integration
2. Gesamtheitstest

Dokumentation:

1. Integration der Iterationsdokumentationen
2. Vervollständigung
3. Präsentation

Literaturverzeichnis

Grundlagen

[1] Yves Langisch, *Anbindung eines Link-Management-Systems an einen Web-Server*; Diplomarbeit, DA-2001.13, TIK, ETHZ

[2] David Lowe and Erik Wilde, *Improving Web Linking Using XLink*; `http://dret.net/netdret/docs/OpenPublish2001.pdf`

[3] Erik Wilde, *Wilde's WWW glossary*; `http://wildesweb.com/glossary/`

Programmierhandbücher

[4] Stephen Asbury, Scott R. Weiner, *Developing Java Enterprise Applications*; John Wiley & Sons; ISBN: 0471327565

[5] Ed Roman, *Mastering Enterprise JavaBeans and the Java 2 Platform, Enterprise Edition*; John Wiley & Sons; ISBN: 0471332291

[6] Richard Monson-Haefel, *Enterprise Javabeans*; O'Reilly & Associates; ISBN: 1565928695

Spezifikationen

[7] Sun Microsystems Inc., *Enterprise JavaBean Specification, Version 2.0, Proposed Final Draft 2*; `http://java.sun.com/products/ejb/docs.html`

[8] Sun Microsystems Inc., *Java 2 Platform Enterprise Edition Specification, v1.3, Proposed Final Draft 4*; `http://java.sun.com/j2ee/download.html#platformspec`

Weblogic Server Dokumentation

[9] BEA Systems, Inc., *BEA WebLogic Server Release 6.0 Documentation*; `http://edocs.bea.com/wls/docs60/`

[10] BEA Systems, Inc., *Programming WebLogic Enterprise JavaBeans*; `http://edocs.bea.com/wls/docs60/ejb`

[11] BEA Systems, Inc., *Programming WebLogic JMS*;
`http://e-docs.bea.com/wls/docs60/jms/`

JBoss Server Dokumentation

[12] JBoss Organization, *JBoss - WORLD CLASS J2EE TECHNOLOGIES IN OPEN SOURCE*;
`http://www.jboss.org/`

[13] JBoss Organization, *JBoss 2.1+ documentation*;
`http://jboss.org/documentation/HTML/index.html`

API Dokumentationen

[14] The Apache Software Foundation, *Xerces 1.3.1 API*;
`http://xml.apache.org/apiDocs/`

[15] Sun Microsystems Inc., *Java Message Service(TM) 1.0.2 API Specification*;
`http://java.sun.com/products/jms/javadoc-102a/`

[16] Sun Microsystems Inc., *Enterprise Java Beans API, v2.0 pfd2*;
`http://java.sun.com/products/ejb/javadoc-2_0-pfd2/`

Tutorials und FAQs

[17] Sun Microsystems Inc., *JavaTM Message Service Tutorial, 1.3 Beta Release* ;
`http://java.sun.com/products/jms/tutorial/html/jmsTOC.fm.html`

[18] jGuru.com, *FAQ Entries in EJB*;
`http://www.jguru.com/faq/subtopic.jsp?topicID=328&page=1`

[19] Borland Software Corporation, *Borland Appserver 4.5 FAQ*;
`http://www.borland.com/devsupport/appserver/faq/45/`

[20] Joshua Fox, *When is a singleton not a singleton?*;
`http://www.javaworld.com/jw-01-2001/jw-0112-singleton.html`

[21] Richard Monson-Haefel, *Create forward-compatible beans in EJB*;
`http://www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-ejb1.html`