2

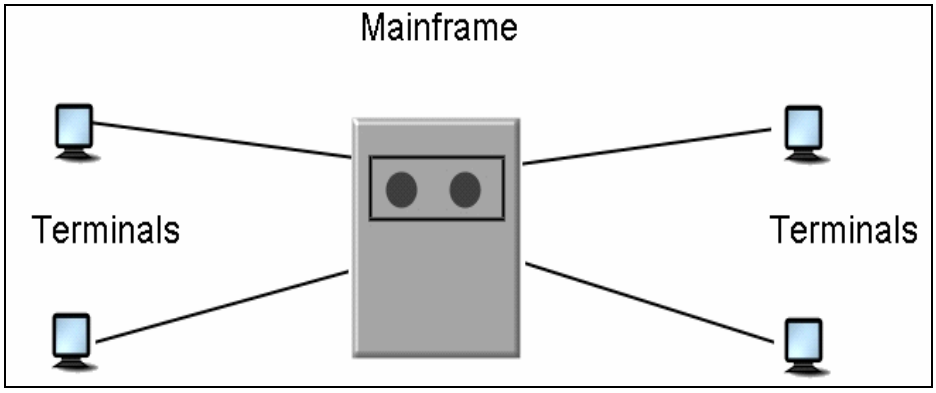
Die Java Enterprise Architekturen

2 Die Java Enterprise Architekturen

2.1 Architekturen im Überblick

2.1.1 Monolithische Architekturen

Hier befinden sich sowohl die Daten als auch die Geschäftslogik auf dem Mainframe:



Die typischen Eigenschaften einer Mainframe-Architektur sind:

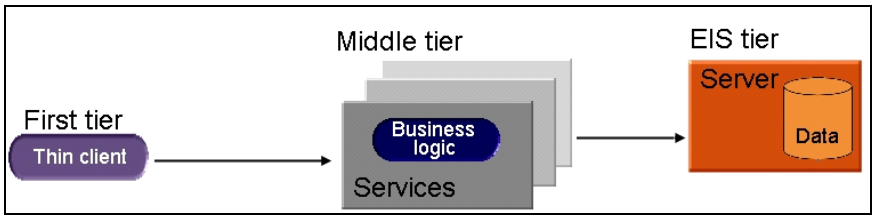
- Auf den Clients laufen oft nur Terminalanwendungen
- Die Datenhaltungs- und Geschäftslogikschichten werden oft vermischt und sind schlecht wartbar
- Die Geschäftslogik wird oft mittels "Hostprogrammen" (PL1) oder Datenbank-Triggern implementiert
- Die einzelnen Schichten lassen sich nicht physikalisch trennen, deswegen ist diese Architektur nicht sehr gut skalierbar
- Heterogene Anwendungen können durch die enge Kopplung zwischen Client und Host nur unzureichend integriert werden.

2.1.2 Client-Server Architektur

Der „Fat“ Client stellt die Präsentation und die Businesslogik dar. Auf dem Server befinden sich lediglich die Daten. Bei der Änderung der Geschäftslogik, müssen jedoch auch die Clients angepasst werden.

Die Client-Server Architektur leidet insbesondere unter langsamen Netzwerkverbindungen, da die Clients auf eine schnelle Verbindung mit der Datenbank angewiesen sind. Jede Abfrage wird auf der Datenbank durchgeführt.

2.1.3 3-Schichtige Architektur



Hier wird der Komponenten-basierte Ansatz deutlich: Die Anwendung wird in wieder verwendbare Komponenten aufgeteilt, wobei natürlich jede Komponente für bestimmte Aufgaben zuständig ist. Die einzelnen Komponenten kommunizieren miteinander über (mehr oder weniger) frei definierbare Schnittstellen.

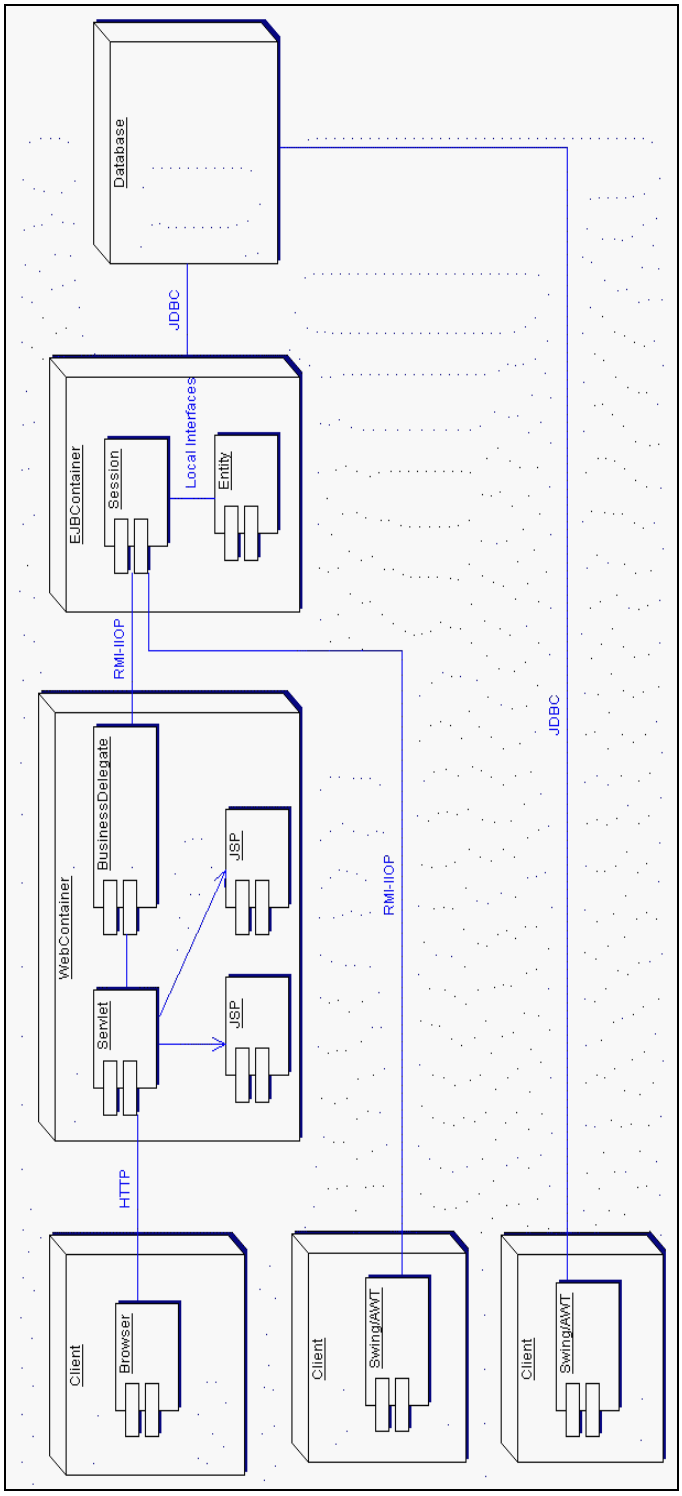
Die Middleware ist im Vergleich zur Client-Server-Architektur bereits wesentlich schlanker und präziser definiert, muss jedoch immer noch drei Aufgaben gleichzeitig übernehmen:

- Fachliche Ablaufsteuerung
- Technologische Adaptionen
- Dienste

2.1.4 n-schichtige e-Business Plattformen

Der Komponenten-Ansatz wird nun in kompletten n-schichtigen e-Business-Plattformen konsequent weiter verfolgt. JEE oder .NET führen noch weitere Schichten mit neuen Aufgaben ein, Beispiel JEE:

- Präsentationsschicht für heterogene Clients
- Geschäftslogikschicht
- Integrationsschicht für heterogene Backend-Systeme



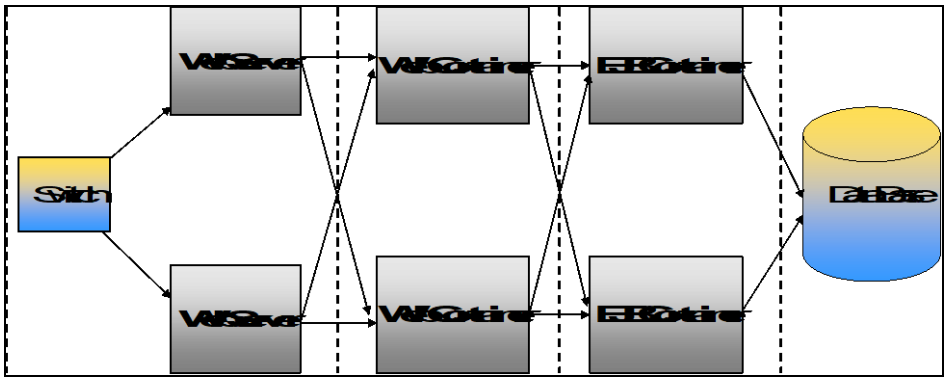
Die neu eingeführten Schichten bieten eine klare Aufgabenverteilung und motivieren damit zu einem strikten Komponenten-Ansatz. Damit

wird die unabhängige Modellierung fachlicher und technologischer Komponenten ermöglicht.

Insgesamt ist das System damit jedoch wesentlich komplexer. Erforderlich ist die Administration verschiedener Server-Produkte. Für die Anwendungsprogrammierer erfordert die Kopplung der einzelnen Schichten die Einführung zusätzlicher Schnittstellen und Datenhaltender Objekte, also ebenfalls ein deutlich erhöhter Aufwand.

2.1.5 Cluster

Die einzelnen Schichten einer e-Business-Plattform laufen auf Grund der benötigten Ausfallsicherheit redundant innerhalb eines Clusters



Ein Cluster liefert höchste Ausfallsicherheit für die darin enthaltenen Anwendungen.

Damit wird jedoch auch eine weitere Ebene der Komplexität eingeführt:

- Session Verwaltung bei Ausfall eines Cluster-Bestandteils
- Über den Cluster verteilte Transaktionen
- Hoher Administrationsaufwand für effizienten Clusterbetrieb notwendig
- Anwendungen müssen „Clusterfest“ sein (Sessions-Migration!).

2.2 Moderne IT-Architekturen

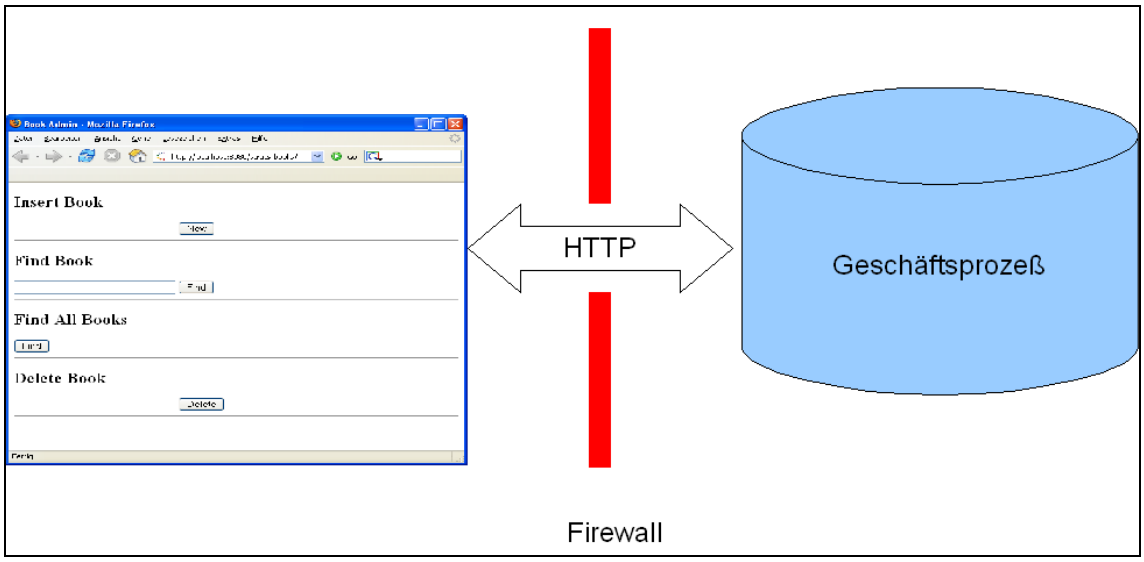
2.2.1 Architektur-Vorgaben

Die IT-Landschaft hat sich mit der rasanten Globalisierung der Unternehmen und deren elektronische Kopplung durch standardisierte Protokolle wie das bekannte HTTP innerhalb weniger Jahre massiv verändert.

Früher waren die Unternehmens-internen Prozesse komplett abgeschottet und konnten deshalb relativ problemlos monolithisch und proprietär realisiert werden. Klassisches Beispiel hierfür ist eine Großrechner-Anwendung mit interner Datenbank und Terminal-basierten Benutzer-Interaktionen. Ein elektronischer Datenaustausch mit den Kunden oder Geschäftspartnern erfolgte entweder gar nicht oder durch spezielle aufgebaute Netzwerke mit jeweils gesondert vereinbarten Protokollen.

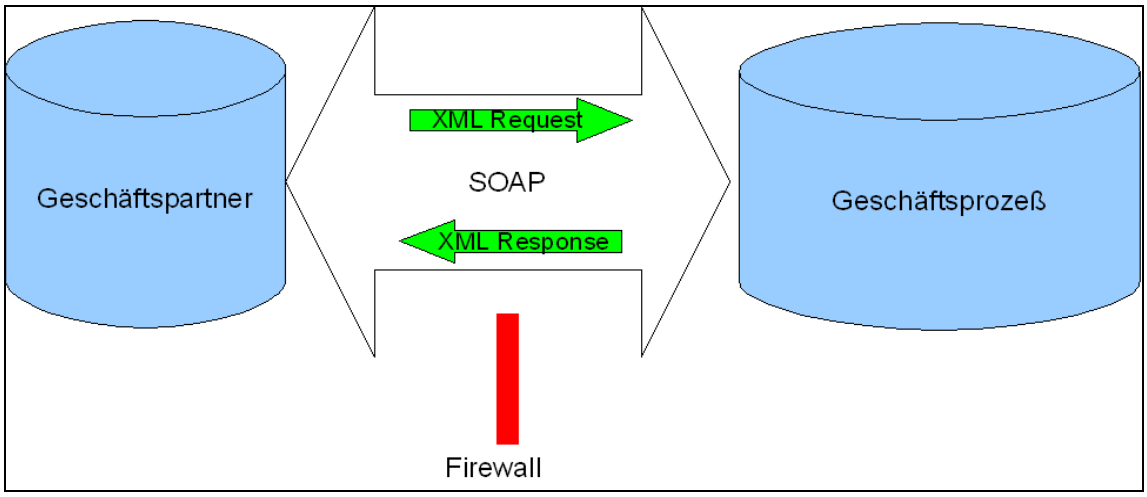
In der heutigen Zeit sind jedoch andere Anforderungen zu erfüllen:

- Business-to-Consumer (B2C): Bestimmte Geschäftsprozesse werden vom Kunden des Unternehmens direkt aufgerufen. Ein Beispiel hierfür ist ein Hersteller, der über Internet seine Produktpalette anbietet und einem Kunden eine Registrierung und anschließend das Bestellen ermöglicht. Die gebräuchlichste Umsetzung ist eine Web-Anwendung: Der Kunde sitzt zu Hause vor seinem Internet Browser und ruft die URL des Unternehmens auf. Die gelieferte HTML-Seite enthält ein Formular, das der Kunde ausfüllt, die eingegebenen Informationen werden über das Internet-Protokoll http zum Unternehmen geschickt und dort von einem Prozess entgegen genommen und verarbeitet. Zum Schutz vor Hacker-Attacken wird das Unternehmen wahrscheinlich noch eine Firewall einsetzen, die Versuche, über andere Protokolle in das Firmen-interne Netzwerk einzudringen verhindert und auch die Kommunikation über http überwacht. Die gesamte Kommunikation kann natürlich bei Bedarf verschlüsselt werden und es kann eine Benutzer-Authentifizierung verlangt werden.

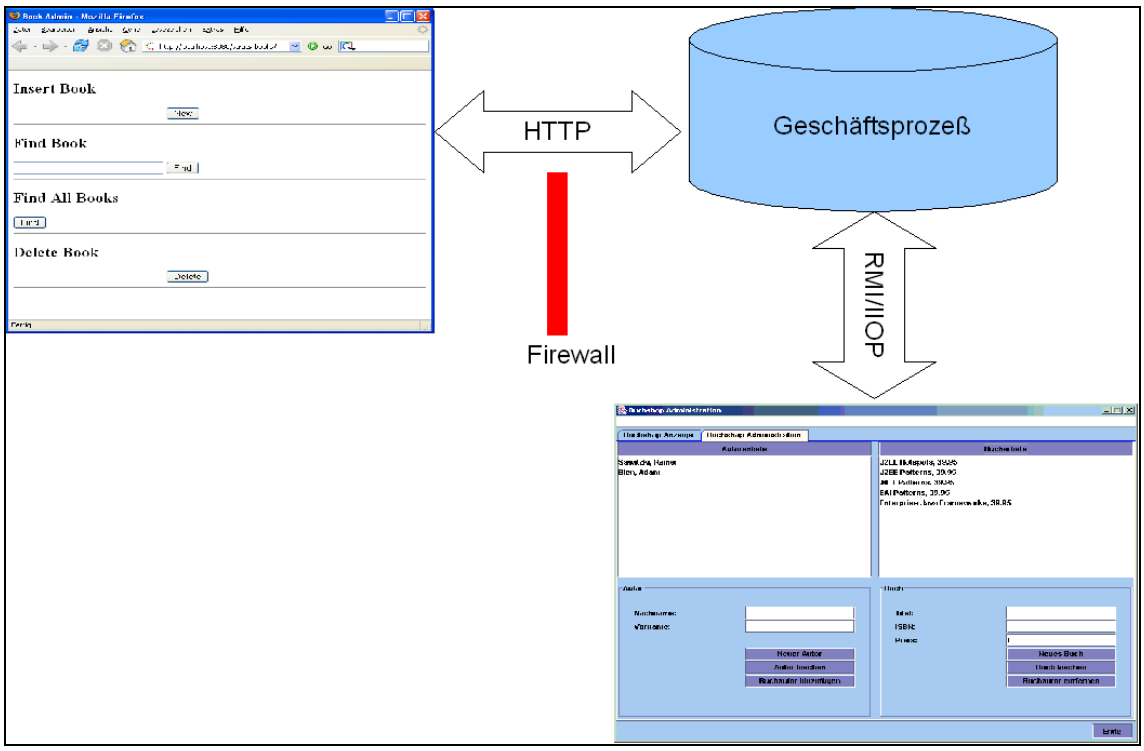


- Business-to-Business (B2B): Geschäftsprozesse können auch Unternehmens-übergreifend sein. Nun ist es erforderlich, die eigenen Prozesse genau zu spezifizieren und damit automatisiert aufrufbar zu machen. So kann beispielsweise eine Flugreservierung (Unternehmen A) automatisch einen Leihwagen anfordern (Unternehmen B), eine Hotelreservierung durchführen (Unternehmen C) und eine Reiseversicherung abschließen (Unternehmen D). Für die Umsetzung dieser Problemstellung bietet es sich an, statt HTML, das für die Präsentation von Daten für einen menschlichen Benutzer verwendet wird, das XML-Format einzusetzen, das perfekt für die automatisierte elektronische Datenverarbeitung geeignet ist. Auch die Beschreibung der Geschäftsprozesse kann sehr detailliert über ein XML-Dokument erfolgen. Dieser Ansatz ist die Basis einer so genannten „Service Oriented Architecture“ (SOA). Dabei erfolgt die Service-Definition in der „Web Service Description Language“ (WSDL), Parameter und Ergebnisse werden durch das SOAP[1]-Protokoll () übertragen.

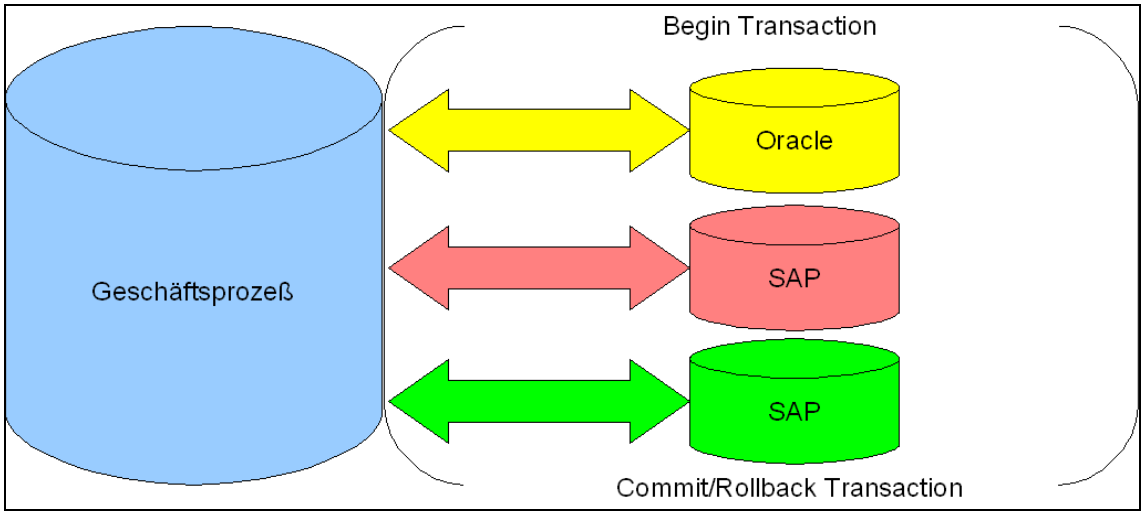
[1] Diese Abkürzung entstammte der ursprünglichen Interpretation als „Simple Object Access Protocol“.



- Flexible Protokollunterstützung: Sollen zwei Geschäftsprozesse miteinander kommunizieren und Daten austauschen so sind dafür aktuell viele verschiedene Protokolle möglich, die sich bezüglich Geschwindigkeit, produzierter Netzwerklast und Verlässlichkeit der Verbindung unterscheiden. Bei Änderungen der Voraussetzungen sollte es einfach möglich sein, ein nun besser geeignetes Protokoll verwenden zu können. So kann ein Geschäftsprozess für eine Anwendung, die innerhalb der Firmen-Firewall läuft, problemlos und effizient über eine „Remote Method Invocation“ (RMI) aufgerufen werden. Soll der gleiche Prozess aber von Außerhalb aufgerufen werden, wird die Firewall nur für http-Aufrufe durchlässig sein. Wird besonderer Wert auf „Quality of Services“ wie Ausfallsicherheit und Garantie der Übermittlung gelegt so bietet sich eher eine Messaging-Variante an.



- Verteilte Transaktionen und Enterprise Application Integration (EAI): Innerhalb eines Unternehmens werden die verschiedensten Betriebssysteme und Software-Systeme parallel betrieben. Verschiedene Teilprozesse sind oft in verschiedenen Systemen realisiert und müssen von einem zentralen Steuerungssystem angesprochen werden. So kann beispielsweise das Bestellen eines Produktes im Hintergrund im Rahmen einer großen Transaktion nacheinander ein Datenbanksystem (z.B. Oracle), SAP und Großrechner-Prozeduren aufrufen. Für die Umsetzung werden jetzt mehrere Dinge benötigt:
  - Ein Transaktions-Manager (auch geläufig als Transaktionsmonitor), der in der Lage ist, für die verschiedenen System die Transaktion zu kontrollieren.
  - Das zentrale Steuerungssystem führt selber wiederum einen Geschäftsprozess aus, der die Aufrufe der anderen Systeme koordiniert.



2.2.2 Die Rolle des Applikationsservers

Die eben aufgeführten Anforderungen sind prinzipiell dadurch umsetzbar, dass jedes im Unternehmen eingesetzte System alle geforderten Fähigkeiten besitzt. So müsste beispielsweise die Oracle-Datenbank ein Web Frontend und eine XML-Schnittstelle besitzen und über RMI, http und Messaging aufrufbar sein. Das gleiche müsste auch für SAP, den IBM Großrechner usw. gelten.

Die Hersteller dieser Systeme erweitern tatsächlich ihre Systeme immer weiter und bieten zumindest teilweise die oben genannten Eigenschaften an. Allerdings sind diese Lösungen nur auf das Ansprechen ihres eigenen Systems ausgelegt.

Es ist sicherlich kein Problem, eine rein Datenbank-zentrierte Anwendung alleine mit Oracle Web Forms zu erstellen. Allerdings ist dieser Ansatz eben nicht flexibel:

- Keine einfache Integrationsmöglichkeit. Insbesondere ist eine System-übergreifende Transaktion so nicht zu realisieren.
- Proprietärer Ansatz, somit ist ein Wechsel der Datenbank und Wiederverwendung der Logik nicht möglich.

Deshalb wird statt dieser Zweischichten-Architektur häufig durch Einführung einer Middleware eine n-Schichten-Architektur eingesetzt. Diese Middleware ist der Applikationsserver und dieser übernimmt die „neu“ eingeführten Aufgaben:

- Er ist über die verschiedensten Netzwerkprotokolle ansprechbar.
- Er bietet eine Plattform für die Erzeugung von HTML-basierten Web-Anwendungen.
- Innerhalb des Applikationsservers können Geschäftsprozesse ablaufen.
- Für eine SOA werden diese Prozesse über WSDL definiert.
- Die Authentifizierung eines Benutzers wird vom Applikationsserver durchgeführt.
- Die wichtigste Aufgabe des Applikationsservers ist aber die Orchestrierung der verschiedensten Systeme insbesondere im Zusammenhang mit verteilten Transaktionen.

Für Applikationsserver existieren eine Reihe von fertigen Plattformen und Frameworks, die die Anwendungsentwicklung durch Komponenten- und Programmiermodelle erleichtern sollen:

- Die Java Enterprise Edition unter Federführung von Sun Microsystems
- .NET von Microsoft
- CORBA von der unabhängigen Object Management Group
- Im Java-Umfeld eine Reihe weiterer Plattformen, insbesondere das Open Source Framework „Spring“
- Web Services Plattformen

Welche dieser Plattformen für die Umsetzung von Geschäftsprozessen für ein bestimmtes Unternehmen geeignet ist, ist nicht eindeutig bestimmbar, sondern hängt von vielen Nebenbedingungen ab. Systemlandschaft (Hardware und Betriebssysteme), Ausbildungsstand der Programmierer (Objektorientiert oder nicht), Kosten für den Betrieb der Middleware usw.

2.2.3 Kommunikationsprotokolle

Die Forderung nach Plattform-unabhängigen Aufrufen und der beliebigen Verteilung der Anwendung führt in der Praxis zur Einführung verschiedener Kommunikationsprotokolle:

- Technologie-abhängige APIs (Datenbankzugriff, LDAP)
- Speziallösungen (Native Socket-Verbindungen)
- Remote Method Invocation (RMI)
- http mit HTML-Dokumenten und HTML-Formularen
- Simple Object Access Protocol (SOAP)
- Messaging

Jedes dieser Protokolle hat bestimmte Stärken und Schwächen, die einen Kompromiss zwischen idealer und technisch machbarer Implementierung darstellen.

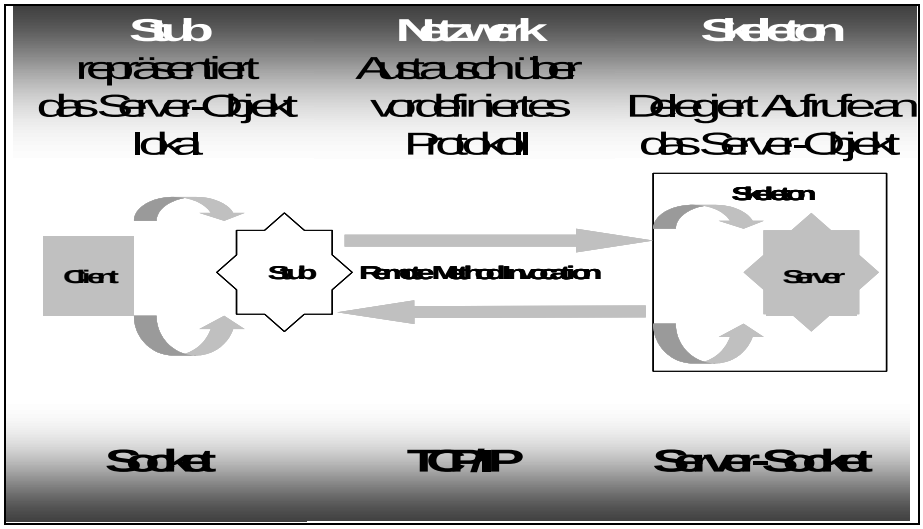
Gemeinsam ist allen Protokollen jedoch, dass Objekte über ein Netzwerk miteinander kommunizieren:

- Client-Objekt bzw. Nachrichten-Produzent
- Server-Objekt bzw. Nachrichten-Empfänger

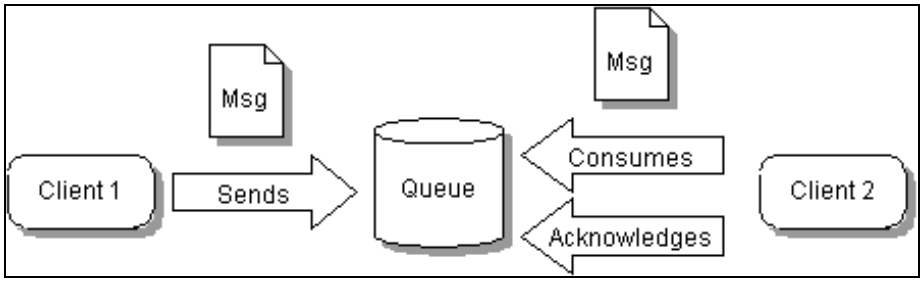
Das Server-Objekt stellt eine Reihe von Methoden für Client-Aufrufe zur Verfügung. Diese Methoden werden durch „Remote Method Invocation“ aufgerufen. Gängige Protokolle sind

- Java RMI
- IIOP als Bestandteil von CORBA
- SOAP

Die eigentliche Netzwerk-Kommunikation im TCP/IP-Netzwerk übernehmen Stubs und Skeletons:



Beim asynchronen Messaging tritt zwischen dem Nachrichten-Produzenten und dem Nachrichten-Empfänger eine vermittelnde „Destination“ auf:



Eigentlich sollte das zu verwendende Kommunikationsprotokoll zwischen Objekten in verteilten Anwendungen im Idealfall das Design der Anwendung nicht beeinflussen. Dies ist jedoch in der Realität leider nicht der Fall:

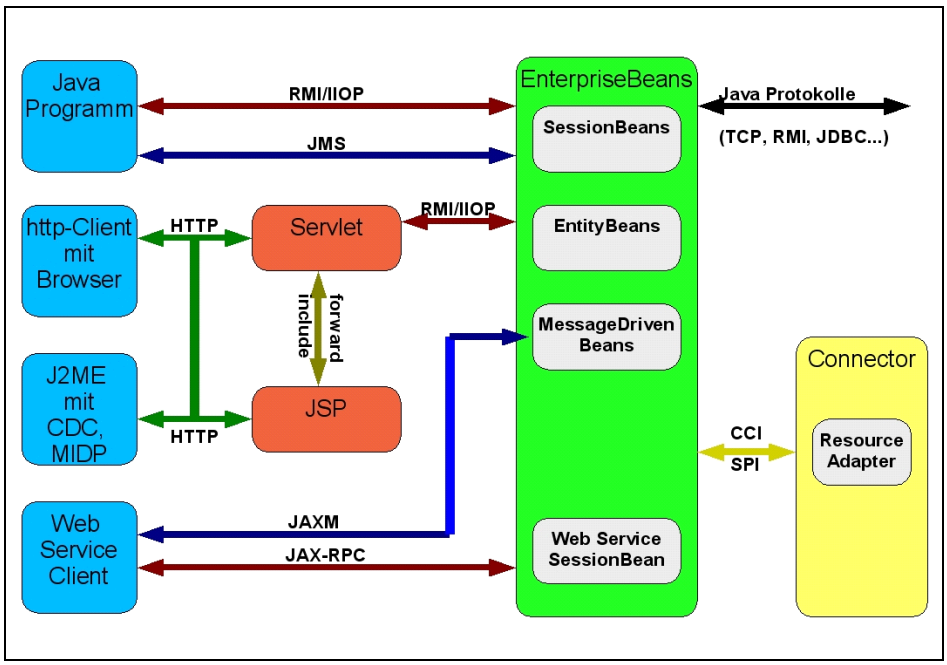
- Call by Value oder Call by Reference
- Synchrone Aufrufe mit Rückgabewerten und Ausnahmebehandlung oder asynchrones Messaging
- Unterstütze Datentypen
- Komplexität und Größe der übertragenen Daten

Insbesondere die unterschiedliche Aufrufsemantik wird eine Austauschbarkeit der Technologien zumindest erschweren. Ebenso problematisch sind die Datentypen. Der Zugriff auf Datenbestände liefert je nach eingesetzter Kommunikations-Technologie unterschiedliche Ergebnis-Typen:

- Serialisierte Objekte bei RMI
- HTML-Dokumente bei Web-Seiten
- XML-Dokumente bei SOAP
- Tabellarische ResultSets bei JDBC
- Byteströme, einfache Zeichenketten…

Der Geschäftsprozess alleine kann keinesfalls die Aufbereitung der Daten für alle möglichen Client-Anforderungen bereitstellen.

Um einen Eindruck der Komplexität der verschiedenen typischerweise eingesetzten Protokollen zu geben, hier ein Bild der JEE-Plattform mit eingetragenen Netzwerk-Technologien:



Eine Zusammenfassung der elementaren Eigenschaften der verschiedenen Protokolle ist der folgenden Tabelle zu entnehmen:

Name	Java RMI	http/HTML	SOAP	Messaging	SOAP mit Messages
Methoden-parameter	Objekte	Einfache Datentypen	XML-Typen	Einfache Datentypen, evtl. Objekte	XML-Datentypen
Rückgabetyp	Objekt	HTML-Dokument	XML-Datentyp	Objekt	XML-Datentyp
Aufruf	Synchron	Synchron	Synchron	Asynchron	Asynchron
Interoperabel	Nein	Eingeschränkt	Ja	Ja, aber nur bei einfachen Datentypen	Ja

Leider ist weder das Übersichtsbild noch die Tabelle vollständig. Alle paar Jahre werden neue Technologien propagiert, die „entscheidende Verbesserungen“ bringen sollen. Letztes Beispiel dafür waren die „Web Services“ und das darin enthaltene SOAP-Protokoll.

All diese Technologien habe ihre Stärken, aber auch Schwächen, die sich im Laufe der Zeit ändern. Ein Umschalten zu besser geeigneten Technologien muss deshalb möglich sein. Dies ist aber bei der Konzep-

tion der Anwendung noch völlig unklar. Was das Design der Anwendung vorsehen muss ist ein flexibles Auswechseln dieser Technologien. Ebenso muss die potenzielle Existenz eines nicht-perfekten Netzwerks, das zum Aufruf von Methoden verwendet werden wird, berücksichtigt werden.

2.3 Java Applikationsserver und die Java Enterprise Edition

2.3.1 Implementierungen

Die Umsetzung von Applikationsservern kann auf unterschiedliche Weise erfolgen. So bietet die Firma Microsoft eine auf dem .NET-Framework basierende Lösung. Auch der IBM Großrechner weist alle Eigenschaften eines Applikationsservers auf. Die auf Grund ihrer offenen Spezifikation und der daraus resultierenden Vielfalt an Herstellern aktuell sicherlich am weitesten verbreitete Technologie ist die Java Enterprise Edition, JEE (früher: J2EE). Die erste Spezifikation entstammt aus dem Jahre 1998, so dass in der kurzlebigen IT-Landschaft bereits von einem etablierten Ansatz gesprochen werden kann. Die Spezifikation wird von den Herstellern unter der Federführung von Sun Microsystems ständig erweitert, verbessert und vereinfacht, so dass alle Anzeichen dafür sprechen, dass die Java Applikationsserver eine zukunftssichere Lösung darstellen.

Hersteller von Java Applikationsservern sind:

- IBM Websphere
- BEA Weblogic
- JBoss
- Oracle iAS
- SAP NetWeaver

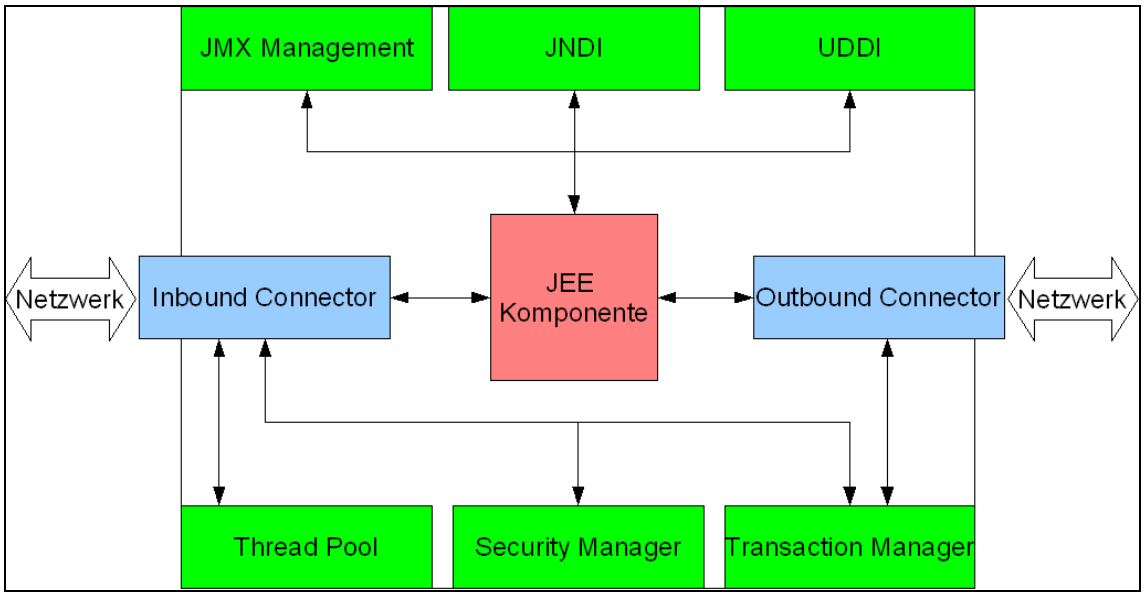
Eine vollständige Produkt-Matrix ist unter http://www.theserverside.com zu finden.

Im Folgenden wird unter „Applikationsserver“ stets der Java Applikationsserver gemeint.

Die Aufgaben des Applikationsservers sind im wahrsten Sinne des Wortes vielschichtig. Deshalb wird dies Middleware intern nochmals in weitere Schichten bzw. Komponenten und Dienste aufgeteilt. Diese Aufteilung wird im Folgenden knapp gegeben, wobei die meisten erwähnten Technologien und Begriffe erst in den folgenden Kapiteln bzw. im Anhang detailliert eingeführt werden.

2.3.2 Interne Dienste

- Jede Aufgabe, die innerhalb des Applikationsservers ausgeführt wird (ein so genannter „Request“), wird von einem Thread (siehe das folgende Kapitel) exklusiv ausgeführt. Aus Effizienzgründen hält der Applikationsserver einen Thread Pool.
- Ein Security Manager übernimmt die Authentifizierung eines Requests und verknüpft die Ausführung mit dem authentifizierten Benutzer.
- Der Transaction Manager startet, koordiniert und beendet Ressourcen-übergreifende verteilte Transaktionen.
- Alle Komponenten und Dienste werden von einem Management-Dienst (die Java Management Extension, JMX) administriert bzw. überwacht. So kann über diese Konsole beispielsweise die Größe des Thread Pools eingestellt und seine Auslastung überwacht werden.
- Der Applikationsserver dient als Container für Inbound und Outbound Connectors sowie verschiedene Implementierungen von JEE-Komponenten.
- Ein Namensdienst für Java-Objekte, das Java Naming and Directory Interface (JNDI)
- Eine Registry für WSDL-Dienste (UDDI)



2.3.3 Inbound Connectors

Inbound Connectors ermöglichen die Kommunikation mit dem Applikationsserver. Laut JEE-Spezifikation sind die folgenden Inbound Connectors vorhanden:

- http/https
- Java Remote Method Invocation (RMI) bzw. die interoperable Variante RMI/IIOP[2]
- Empfangen von Messages mit dem Java Messaging Service (JMS)
- SOAP

Für Protokolle, die nicht im Standard enthalten sind, können spezielle Implementierungen für Inbound Connectors als Plug-In in den Applikationsserver installiert werden. Dafür existiert auch ein breiter Markt von Herstellern und Produkten.

Die Inbound Connectors führen selber keine Geschäftsprozesse aus. Laut Konzeption wandeln sie eine einkommende Netzwerkverbindung in einen Request um der, wie oben bereits angesprochen, von einem Thread aus dem Thread Pool ausgeführt wird. Die Geschäftslogik wird in einer JEE-Komponente definiert und ausgeführt. Der Request selber enthält die vom Connector aus dem Netzwerk-Datenstrom gelesene Informationen aufbereitet als Java-Objekt. Dieses Java-Objekt wird dann entweder direkt der verarbeitenden Komponente übergeben (z.B. der `javax.servlet.HttpServletRequest` oder die `javax.jms.Message)` oder als Methodenaufruf mit Parameterübergabe interpretiert.

Ein Inbound Connector auch kann im Rahmen einer Verteilten Transaktion aufgerufen werden.

[2] IIOP ist das so genannte „Internet Inter ORB Protocol“, das Kommunikationsprotokoll der Plattform-unabhängigen CORBA-Spezifikation

2.3.4 JEE Komponenten

JEE Komponenten bestehen aus zwei Komponenten: Einem Container, der vom Applikationsserver zur Verfügung gestellt wird und der eigentlichen Komponente, die als Bestandteil eines Anwendungsprogramms aufzufassen ist.

Die JEE-Spezifikation enthält die folgenden Typen von Komponenten:

- Servlets werden vom http Inbound Connector aufgerufen. Die Entscheidung, welches Servlet (und damit welche Anwendungslogik) aufgerufen werden soll erfolgt über die URL des http-Aufrufs. Ein spezieller Typ von Servlet kann auch über SOAP aufgerufen werden.
- Für die Erzeugung von speziellen Dokumenten-Formaten für verschiedene Client-Typen gibt es eine spezielle Skript- und Template-Sprache, die JavaServer Pages (JSP) und die JavaServer Pages Standard Tag Library (JSTL). Zur Definition von Browser-basierten Oberflächen gibt es die UI-Bibliothek der JavaServer Faces /JSF).
- Session Beans sind eine von drei unterschiedlichen Ausprägungen von Enterprise Java Beans (EJBs). Session Beans werden entweder über RMI/IIOP oder aber lokal von anderen JEE-Komponenten aufgerufen. Es gibt zwei Subarten von Session Beans: Stateless und Stateful. Die Stateless Session Beans können auch über SOAP angesprochen werden. Verschiedene Session Bean Implementierungen werden über einen eindeutigen Alias-Namen unterschieden.
- Message Driven Beans können über JMS angesprochen werden. Welche Message Driven Bean vom Inbound Connector für JMS aufgerufen wird hängt davon ab, bei welcher JMS-Destination (einer Art Adresse) die Message Driven Bean registriert wurde.
- Entity Beans werden meistens nur lokal von anderen Komponenten angesprochen. Sie dienen zum Auslesen von Daten aus einem Repository, meistens einer SQL-Datenbank. Ab der J EE 5 Sprachversion können EntityBeans auch in serialisierter Form zu einem entfernten Client übertragen werden. Hierbei geht natürlich die Anbindung an die Datenbank verloren.

Hinweis: EntityBeans sind zwar noch Bestandteil der Spezifikation, werden aber auf Grund der unzeitgemäßen Konzeption in der Praxis de facto nicht mehr verwendet.

2.3.5 Outbund Connectors

Outbound Connectors werden von JEE-Komponenten lokal benutzt. Ihre Aufgabe ist es, einen Geschäftsprozess innerhalb eines Backend-Systems aufzurufen. Dazu hält der Outbound Connector einen Connection Pool. Damit sind sie die Hauptkomponenten für die Enterprise Application Integration.

Die Outbound Connectors werden auch vom Transaktionsmanager (für die Koordinierung der Verteilten Transaktionen) und vom Security Manager (Propagierung des angemeldeten Benutzers) benutzt.

Die JEE-Spezifikation verlangt die folgenden Outbound Connectors:

- Die Data Sources verbinden zu SQL-Datenbanksystemen.
- Versenden von JMS Messages
- Anbinden an ein Email-System

Alle weiteren Systeme sind wiederum durch Plug-Ins installierbar, für die ein breiter Markt existiert.

2.4 Programmiermodell

2.4.1 Allgemeines

Seit der ersten Spezifikation für die Java Enterprise Edition (JEE) wurden bei jedem Versionswechsel neue bzw. geänderte Features eingeführt. So ergänzten beispielsweise MessageDrivenBeans die ursprünglich konzipierten SessionBeans und Servlets. Die Datenzugriffstechnologien wurden insgesamt drei Mal komplett umdefiniert und haben sich endlich auf dem Java Persistence API stabilisiert. Mit Einführung der JEE 5 wurde das ursprünglich extrem technisch gehaltene Programmiermodell, das auf einer Mischung aus Schnittstellen-Implementierung, Schnittstellen-Definition und automatischer Codegenerierung basierte, durch einen konsequent Annotations-basierten Ansatz ersetzt.

Komplett gleich geblieben ist aber das Verhalten der JEE Komponenten zur Laufzeit: Der Applikationsserver stellt für jeden Komponenten-Typ einen JEE Container zur Verfügung, der die innerhalb der eigentlichen Implementierung ablaufenden Sequenzen durch konfigurierbare Service-Aufrufe ergänzt.

2.4.2 Stateless SessionBeans

In der JEE wird eine Klasse durch eine simple Annotation zur EJB:

```
@Stateless
public class DemoBean implements Demo{
    public String echoMessage(String message){
        System.out.println("Receiving message " + message);
        return "Back from Server: " + message;
    }
}
```

Diese Klasse implementiert eine einzige Schnittstelle, die wiederum über eine Annotation den möglichen Zugriff, hier lokal, definiert:

```
@Local
public interface Demo{
    public String echoMessage(String message);
}
```

Warum wird eine Klasse als EJB definiert? Die Antwort ist einfach:

Eine EJB ermöglicht eine Annotations-basierte Transaktionssteuerung.

2.4.3 Context and Dependency Injection

Die Objekte der Anwendung werden zur Laufzeit zu einem Objekt-Geflecht verbunden. Dies übernimmt ein Context.

Der JEE-Applikationsserver besitzt aus historischen Gründen mehrere fast gleichwertige Context-Implementierungen:

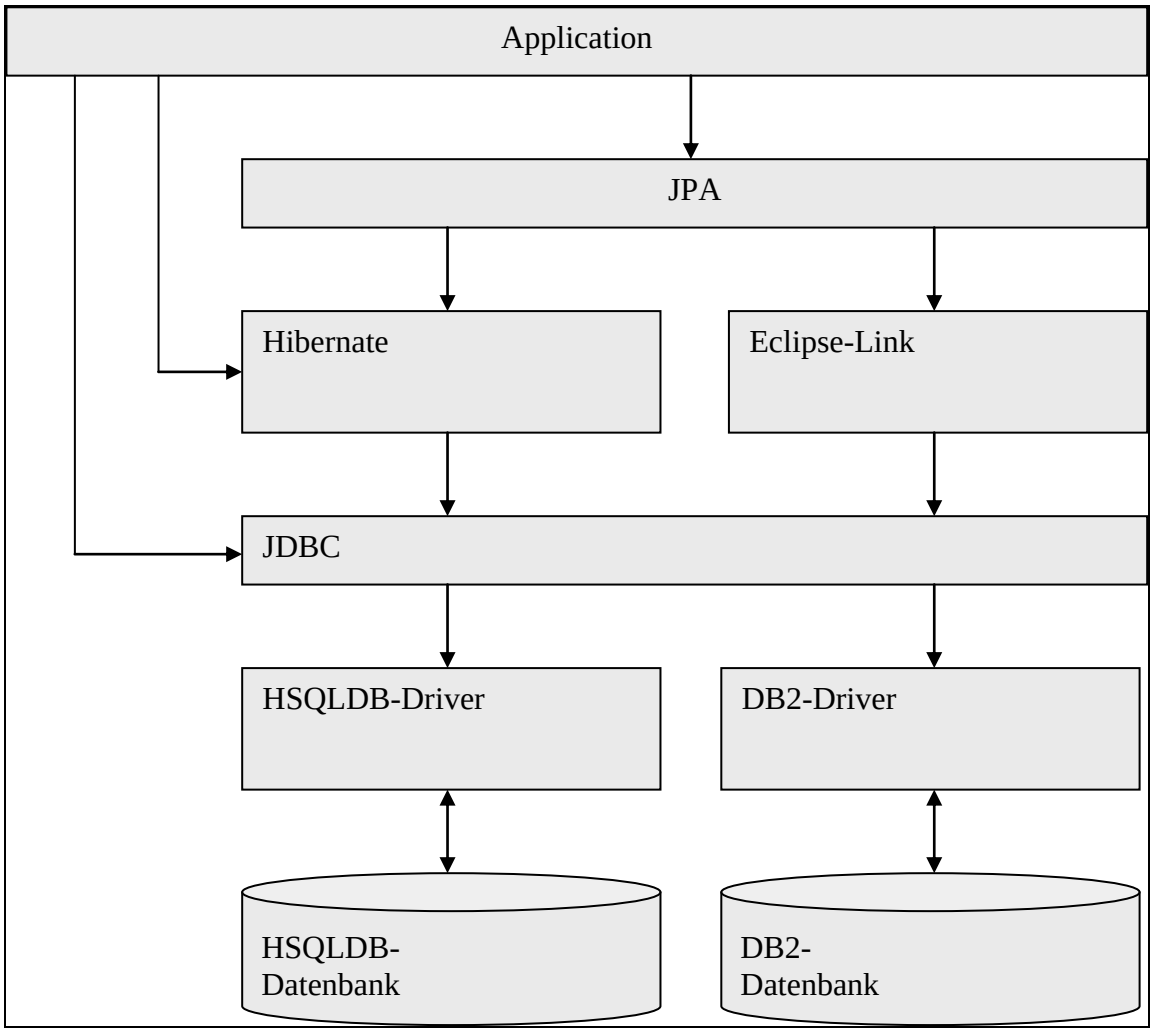
- Dependencies werden innerhalb einer EJB-Anwendung mit den Annotationen `@EJB`, `@Resource und @PersistenceContext` definiert. Komponenten werden als `@Stateless`, `@Stateful` oder `@MessageDriven` gekennzeichnet.
- Innerhalb einer JSF-basierten Web-Anwendung werden `@ManagedBean` und `@ManagedProperty` benutzt.
- In allen anderen Umgebungen, bei Bedarf sogar vollkommen unabhängig vom Applikationsserver existiert CDI mit den Annotationen `@Named` und `@Inject`.

Das ebenfalls sehr verbreitete Spring-Framework war übrigens an dieser Stelle ein äußerst inspirierender Konkurrent der Java Enterprise Edition. Viele Ideen wurden im CDI-Framework übernommen, allerdings im Detail doch differenziert übernommen. Spring stellt sich in der aktuellen Version der Java Enterprise Edition (JEE 6) auf Grund einer Vielzahl vorhandener Hilfsklassen[3] sowie der definitiv eleganteren Unterstützung Aspekt-orientierter Ansätze immer noch als überlegen dar. Jedoch ist dies in der Praxis nicht mehr sonderlich relevant. Weiterhin beginnt sich die Spring Community immer mehr mit den neu eingeführten Standards anzufreunden[4] und diese zu unterstützen. Ob Spring irgendwann einmal als „simpler“ JEE-Provider anzusehen sein wird, wird die Zukunft erweisen.

[3] Hilfsklassen sind auch definitiv nicht im Fokus der JEE-Spezifikation!

[4] Was natürlich keine Überraschung ist: Die neuen Standards sind ja wie erwähnt auch von Spring mitgeprägt.

2.4.4 Java Persistence API



Ähnlich wie über JDBC unterschiedliche Datenbanken weitgehend transparent angesprochen werden können, können über JPA unterschiedliche OR-Mapper angesprochen werden. Bei JDBC setzt dies voraus, dass für die jeweilige Datenbank ein entsprechender Treiber existiert (also: Implementierungen der JDBC-Interfaces!). Ein OR-Mapper, der als "JPA-Provider" fungieren kann, muss ebenfalls entsprechende Interfaces (die in JPA spezifiziert sind) implementieren.

Mit Hilfe des EntityManagers werden @Entity-annotierte Klassen mit der Datenbank synchronisiert.

2.4.5 JavaServer Faces

JSF2 bietet ein modern konzipiertes und anerkanntes Framework für die Erstellung einer Web Anwendung.

Wichtige Features umfassen:

- Einfache Konfiguration
- Generelle Benutzung von XHTML
- Templating mit Facelets
- Reichhaltiger Satz von Scopes für Anwendungsdaten
- Verwendung von HTTP-GET-Requests
- Integration von AJAX
- Verwendung von Annotations
- Event-basiertes Programmiermodell
- Konvertierungs- und Validierungsframework
- Einfache Erstellung eigener Komponenten

2.4.6 Jax-WS

Die Unterstützung von SOAP-basierten WebServices ist mittlerweile Bestandteil der Java Standard-Edition. Applikationsserver unterstützen häufig zusätzlich noch einen Satz der WS-*-Spezifikationen wie beispielsweise WS-Security oder WS-Adressing. Ebenso ist die MTOM-Optimierung gebräuchlich.

Web Services werden in der gebräuchlichen Ausprägung Document/literal bzw. Document/literal-wrapped realisiert. Sowohl der Code First- als auch der Contract-First-Ansatz werden von Java unterstützt.

Der Web Service wird auf Folgende Art schrittweise entwickelt.

- Es wird ein „Plain Old Java Object“, ein POJO programmiert. Einige Web Services Plattformen verlangen, dass das POJO eine Schnittstelle implementiert. Dies ist jedoch konzeptuell optional.

```
import java.util.Random;


public class RandomKeyGenerator{
private Random random;
{
  random = new Random(this.hashCode() +
  System.currentTimeMillis());
}
public String next() {
```

```
  System.out.println("calling next using " + this);
  return Integer.toString(random.nextInt());
}
}
```

- Das POJO wird annotiert. Dazu dienen die Annotationen aus der JAX-WS-Bibliothek.

```
import java.util.Random;

import javax.jws.WebResult;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;

@WebService(name = "KeyGenerationWebService", target-
  Namespace =
  "http://org.javacream.ws/samples/keygeneration", service-
  Name = "KeyGeneratorService")
@SOAPBinding(parameterStyle = SOAPBind-
  ing.ParameterStyle.WRAPPED)
public class RandomKeyGenerator{

private Random random;

{
  random = new Random(this.hashCode() + Sys-
  tem.currentTimeMillis());
}
public @WebResult(name = "key")
String next() {
  System.out.println("calling next using " + this);
  return Integer.toString(random.nextInt());
}
}
```

- Falls vorhanden werden die benutzerdefinierten Datentypen mit JAXB[5]-Annotationen versehen. Eine, für diese einfache Anwendung nicht benutzte Klasse sieht wie folgt aus:

```
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "StoreEntry",
  namespace="http://javacream.org/store/")
@XmlRootElement(namespace="http://javacream.org/store/",
  name = "StoreEntry")
public class StoreEntry {

@XmlElement(namespace="http://javacream.org/store/",
  required = true)
private String category;
@XmlElement(namespace="http://javacream.org/store/",
  required = true)
private String item;

public StoreEntry(){}
public StoreEntry(String category, String item) {
  super();
  this.category = category;
  this.item = item;
}
protected String getCategory() {
  return category;
}
@Override
public String toString() {
```

[5] Das Java API for XML Binding stellt ein API mit Annotationen zur Verfügung, mit denen Java-Klassen nach in ein XML-Schema umgewandelt werden können und umgekehrt. Das API stellt Encoder und Decoder zur Verfügung, mit denen Objekt-Bäume in ein XML-Dokument umgewandelt werden können. Dabei werden auch die Constraints berücksichtigt. Ein Code- bzw. Schema-Generator ist Bestandteil der Implementierungen.

```
  return "StoreEntry [category=" + category + ", item=" +
  item + "]";
}
protected String getItem() {
  return item;
}
@Override
public int hashCode() {
//…}
@Override
public boolean equals(Object obj) {
//…
}
```

Hilfsklassen und WSDL sowie Schema-Definitionen werden generiert.

2.4.7 Jax-RS

REST (Representional State Transfer) ist prinzipiell ein reduzierter Web Services Stack. Grundidee ist das vereinfachte Senden und Empfangen der Nutzdaten. Diese Nutzdaten werden als Ressource interpretiert. Damit können Web Services sehr einfach über Aktionen und Links angesprochen werden. Weiterhin unterstützen RESTful Web Services das Konzept der MIME-Types: Der Client kann einen bestimmten Datentyp anfordern.

Um einen RESTful Web Service im Java-Umfeld zu realisieren kann die JAX-RS-Bibliothek benutzt werden[6].

[6] Diese ist zwar noch nicht notwendige Bestandteil der JEE-Umsetzung, wird aber de facto von den Herstellern zur Verfügung gestellt. Oracle stellt mit Jersey eine Servlet-basierte Referenz-Implementierung zur Verfügung.

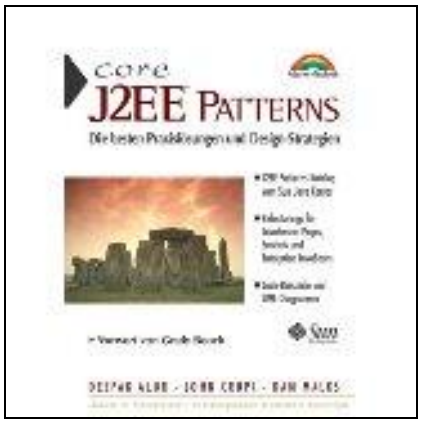
Ein konkreter RESTful Web Service ist erstaunlich einfach zu entwickeln: Im Wesentlichen ist ein Mapping zwischen einer http-URL und einer Java-Methode mit Parametern notwendig. Das ist mit Annotationen aus javax.ws.rs möglich:

```
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("demo")
@Produces(MediaType.APPLICATION_XML)
public class SimpleRestBean {

 @GET
 @Path("/{id}")
 public String findResource(@PathParam("id") String id){
  System.out.println("Finding resource for id " + id);
  return "Resource " + id;
 }
}
```

2.5 Patterns und Idiome

2.5.1 Was ist mit den J2EE-Pattern?

Als Bestandteil der Blueprints wurden von Sun schon seit den Urzeiten der JEE eine Sammlung von Best Practices und Referenzimplementierungen gepflegt. Unter anderem wurde daraus ein Satz von Design Patterns propagiert, der unter dem Namen „J2EE-Patterns“ vermarktet wurde[7].



Bei genauerer Betrachtung stellt sich jedoch heraus, dass diese Patterns entweder

- nur eine minimal veränderte Formulierung eines bereits vorhandenen Patterns sind (Beispiel: „Session Facade“ im Vergleich zum Gamma-Pattern „Facade“ oder Martin Fowlers „Remote Facade“), oder
- eine Unzulänglichkeit der Spezifikation behebt (der „Value Object Assembler“, der aus den EntityBeans Schichten-übergreifend verwendbare Objekte macht) und damit bestenfalls ein Idiom darstellt oder
- schlicht und ergreifend eine simple Utility-Klasse beschreibt (der „ServiceLocator“)

Die J2EE-Patterns hatten ihre Zeit und haben die JEE-Spezifikation mit Sicherheit erheblich weiter gebracht, haben aber im Rückblick bis auf wenige Ausnahmen die hohen Anforderungen an Design Patterns nicht erfüllt. Stattdessen sollten besser allgemeine Patterns verwendet werden, die auch außerhalb der JEE eine Bedeutung haben.

[7] Und früher ehrlicherweise auch die Grundlage für dieses Seminar darstellte.

2.5.2 GoF Patterns

Der Standard-Katalog der Design Patterns von Erich Gamma und Kollegen, der „Gang of Four“, ist zum Großteil auch in heutigen Anwendungen aktuell gültig. Auch hier sind jedoch einige Patterns weggefallen (Factory, Singleton) bzw. sind modifiziert und umbenannt worden (so ist der in einem folgenden Kapitel angesprochene „Business Delegate“ eigentlich nichts anderes als ein Adapter).

2.5.3 Fowler-Patterns

Martin Fowler veröffentlichte in seinen Büchern des Öfteren diverse Design-Pattern-Kataloge:

- „Patterns of Enterprise Application Architecture“

„Refactoring Patterns“

…

Die ersten beiden Werke haben einen deutlichen Bezug zur Anwendungsentwicklung mit JEE. Im Vergleich zu den J2EE-Patterns sind diese jedoch Technologie-unabhängig formuliert und deshalb allgemeiner gültig und verwendbar. Allerdings sind auch hier einige Patterns bei Verwendung der JEE nicht mehr relevant: Grund hierfür ist, dass diese Pate standen für Frameworks wie beispielsweise das Java Persistence API („ActiveRecord“-Pattern)oder die JavaServer Faces. Damit sind diese Entwurfsmuster immanent vorhanden und müssen deshalb nicht mehr im Rahmen der Programmierung vom Entwickler direkt benutzt werden.