Hochschule München

Fakultät 07 Informatik und Mathematik

Aktuelle Technologien zur Entwicklung Verteilter Java-Anwendungen

---

Moderne Architekturen mit EJB 3.1 und CDI

Verfasser: Anton Bartl

Matrikelnummer: 05522309

Abgabetermin: 31.05.2013

Betreut von: Michael Theis

Erklärung:

Die oben angegebenen Verfasser erklären durch Ihre Unterschrift, dass Sie diese Seminararbeit selbstständig erstellt, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt, sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet haben.

Ort, Datum Unterschrift Student

Inhaltsverzeichnis

Moderne Architekturen mit EJB 3.1 und CDI 1

Inhaltsverzeichnis II

Abbildungsverzeichnis V

1 Einleitung 1

2 Enterprise JavaBeans 3.1 3

2.1 Die Java EE Architektur 3

2.2 Der Java EE Applikation Server 4

2.3 Der EJB-Container: 6

2.4 Deployment Deskriptor & Annotationen 9

2.5 Convention over Configuration 9

2.6 Dependency Injection & Inversion of Control 10

2.7 Session Beans: 11

2.7.1 Stateless Session Beans 11

2.7.2 Stateful Session Beans 13

2.7.2.1 Aktivierung Passivierung 15

2.7.3 Singleton Session Bean 17

3 Context and Dependency Injection 21

3.1 Das CDI Grundkonzept: 22

3.2 Das Bean Verständnis von CDI 25

3.2.1 Managed Beans 25

3.2.2 Session Beans 25

3.3 Bean Typen 29

3.4 Qualifier 30

3.5 Alternativen 31

3.6 Expression Language Name 32

4 Timer Service 33

4.1 Programmgesteuerte Timer 33

4.1.1 Grundlegende Funktionsweise programmgesteuerter Timer 34

4.1.2 Die Timeout Methode 35

4.1.3 Das Interface Timer Service Interface 35

4.1.4 Single Event Timer 36

4.1.5 Interval Timer 36

4.1.6 Calendar Schedule Based Timer 36

4.2 Automatische Timer 37

4.3 Timer und Transaktionen 39

5 Asynchrone Methodenaufrufe....40

5.1 Ablauf eines asynchronen Methodenaufrufs....41

5.2 Asynchrone Methoden und Transaktionen....43

6 Patterns:....44

6.1 Boundary....45

6.2 Control....47

6.3 Entity....49

7 Fazit....50

Anhang A Listing....51

Anhang B Literaturverzeichnis....52

Abbildungsverzeichnis

Abbildung 1: Java EE Architekturmodell, entnommen aus *(Weil)* 4
Abbildung 2: Java EE Applikationsserver, entnommen aus (Oliver Ihns, S.60) 5
Abbildung 3: Der EJB-Container, entnommen aus (Oliver Ihns, S.61) 6
Abbildung 4: Stateless Session Bean, in Anlehnung an *(tversu)* 12
Abbildung 5: Stateful Session Bean, in Anlehnung an *(tversu)* 13
Abbildung 6: Realisierung eines rudimentären Warenkorbs mit Stateful Session Bean, Managed Bean und JSF 16
Abbildung 7: Singleton Session Bean, in Anlehnung an *(tversu)* 19
Abbildung 8: Zugriffszähler für Webseite 20
Abbildung 9: Schichtenübergreifendes Zusammenspiel, entnommen aus (Oliver Ihns, S. 496) 21
Abbildung 10: Anwendungskomponentenentnommen aus (Weil, S.16) 22
Abbildung 11: Funktionsweise programmgesteuerter Timer, entnommen aus *(Oliver Ihns, S.463)* 34
Abbildung 12: Funktionsweise automatischer Timer, entnommen aus *(Oliver Ihns, S.472)* 38
Abbildung 13: Consolen Ausgabe des automatischen Timers von Listing 15 39
Abbildung 14: Ablauf eines asynchronen Session Bean Aufrufs mit Rückgabewert, entnommen aus (Oliver Ihns, S.148) 41
Abbildung 15: Aufruf der get()-Methode während andauernder Berechnung, entnommen aus *(Oliver Ihns, S.150)* 42

1 Einleitung

Enterprise JavaBeans kurz EJB's erschienen erstmals in der Version EJB 1.0 im Rahmen der Java Professional Edition im Jahr 1998. Die Idee von JPE war, die typischen Komponenten, die in verteilten Anwendungen verwendet werden, zu identifizieren, und diese Dienste bzw. Schnittstellen in einer bestimmten Umgebung zur Verfügung zu stellen. Anstatt aber eine bestimmte Implementierung dieser Komponenten vorzunehmen, hat man sich dazu entschieden, zu jedem dieser identifizierten Komponenten / Dienste eine Spezifikation zu schreiben.

Diese Spezifikationen wurden von der Firma SUN erstmals 1998 entwickelt und unter dem Namen „JPE" (Java Professional Edition) veröffentlicht. In solchen Spezifikation steht immer die Beschreibung der Schnittstellen, die in dieser Komponente angeboten werden muss, sowie eine genaue Beschreibung der Funktionalität.

Unter JPE oder Java EE, wie es heute genannt wird, versteht man ein ganzes Bündel an Spezifikationen, also kein Programm oder fertige Komponenten, die man irgendwo herunterladen und verwenden könnte. An dieser Stelle kommen die Java EE Applikation Server zum Einsatz. Solche Applikation Server sind wiederum Java Anwendungen, die die oben erwähnten Spezifikationen (Schnittstellen + Funktionalitäten) ausimplementiert haben und diese nach außen hin als Dienste anbieten.

Enterprise JavaBeans und Context and Dependency Injection sind beides Spezifikationen, die im Rahmen von JPE bzw. Java EE entwickelt wurden. Enterprise JavaBeans sind in erster Näherung, für die Bereitstellung von Geschäftslogik in Form von flexiblen Komponenten zuständig. Context and Dependency Injection ist für die Bereitstellung und Verknüpfung von Komponenten und Diensten in einer Enterprise Applikation verantwortlich. Diese Verknüpfung findet durch „Injektion" von Abhängigkeiten statt. Allerdings lässt sich CDI mittlerweile auch außerhalb von Applikation Servern im Java SE Umfeld nutzen.

Vor allem die Enterprise JavaBeans waren in der Entwicklergemeinde nicht immer so beliebt, wie sie es heute sind. Auch wenn sich durch die Verwendung von EJB's eine Menge Vorteile ergaben, so zeichneten sich frühere Versionen (vor 3.0) der EJB's durch hohen Entwicklungs- und Konfigurationsaufwand, Komplexität, höhere Fehleranfälligkeit und eine Menge an rein infrastrukturellem Code aus.

Mit Version 3.0 wurden die EJB's einer Schlankheitskur unterzogen, außerdem fand eine massive Vereinfachung der EJB Komponenten statt.

Heute liegen die EJB's in der Version 3.1 vor. Context and Dependency Injection das erstmals 2009 in Java EE 6 standardmäßig aufgenommen wurde, stammt ursprünglich aus dem Open-Source-Framework JBoss Seam, wurde aber im Java-Kontext erstmals im Spring-Framework umgesetzt. *(Oliver Ihns), (Weil)*

2 Enterprise JavaBeans 3.1

Die Idee von Enterprise JavaBeans war, dem Entwickler Komponenten zur Verfügung zu stellen, die ihm den Aufwand zur Implementierung von infrastrukturellem Code abnehmen. Kürzere Entwicklungszeiten, schlankerer, besser lesbarer Source Code und steigende Qualität waren einige Hauptziele von Enterprise JavaBeans. Ab der Version 3.0 ist diese Vision Realität geworden. Wie genau diese Ziele erreicht wurden, und welche Funktionalitäten uns Enterprise JavaBeans bieten, wird im folgenden Kapitel erläutert. *(Oliver Ihns)*

Nachfolgend soll auf die Java EE Architektur, den AS, den EJB Container, und die Session Beans, eingegangen werden. Grundsätzlich gibt es drei Arten von Bohnen: die Session Beans, die Message Driven Beans und die Persistent Entities. In dieser Seminararbeit wollen wir uns ausschließlich den Session Beans widmen. *(Oliver Ihns, S.95), (Weil, S. 237/238)*

2.1 Die Java EE Architektur

Die Architektur von Java EE Anwendungen lassen sich in eine Clientschicht, die serverseitige Implementierung und in die Schicht, in der die jeweiligen Daten persistiert oder anderweitig weiterverarbeitet werden, unterscheiden.

Wenn man in der heutigen Zeit von Clients redet, spricht man typischerweise von einem Webbrowser, es kann sich aber auch um Desktopanwendung (z. B. Swing Client) sowie mobile Endgeräte handeln. Eine Desktopanwendung, wie z. B. ein Swing Client, implementiert die Präsentationslogik selbst. Kommuniziert eine solche Anwendung mit unserer Java Enterprise Applikation, greift diese meist direkt auf die Geschäftslogik zu. Für den häufigeren Fall, dass unsere Java EE Anwendung mit einem Browser kommuniziert, muss die Präsentationslogik serverseitig implementiert werden.

In der Java EE Architektur wird die Präsentationslogik von Technologien wie Java Server Pages, Java Server Faces, Servlets und WebServices umgesetzt.

Die Geschäftslogik wird in Java EE 6 mithilfe von Context and Dependency Injection, Enterprise Java Beans, der Java Persistence API und einigen weiteren Technologien realisiert. *(Weil, S.11/12)*

Die folgende Abbildung soll einen Überblick über die Java EE Architektur geben:

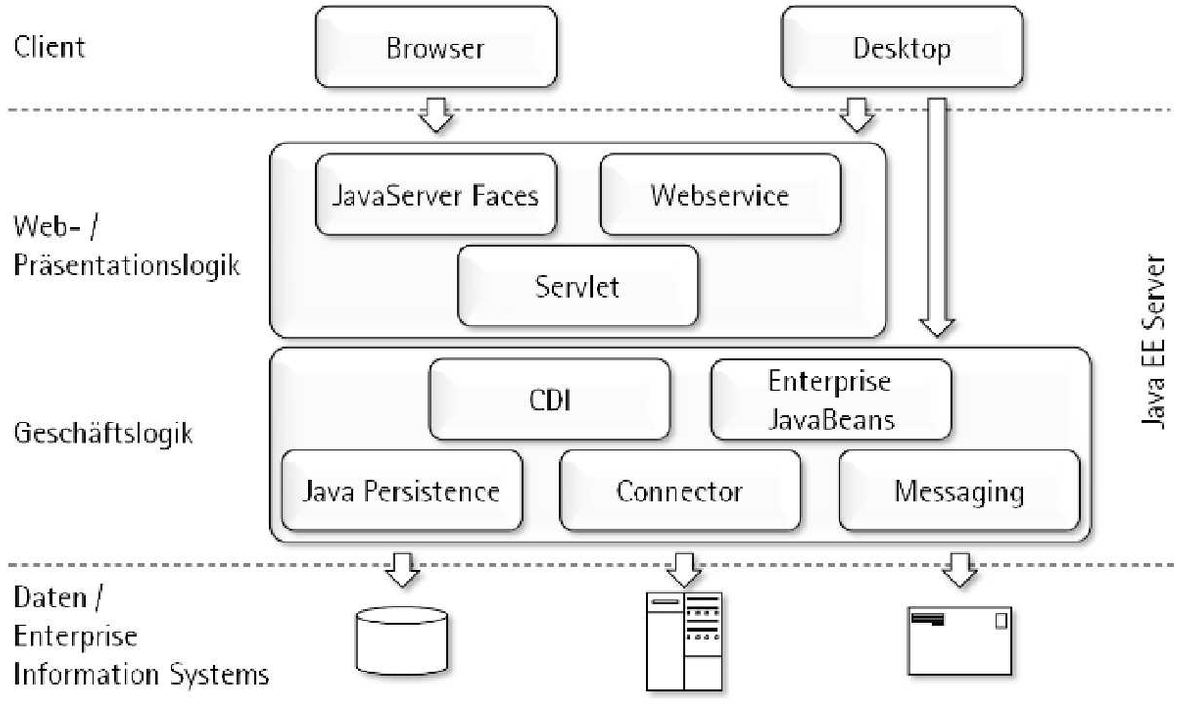

Abbildung 1: Java EE Architekturmodell, entnommen aus *(Weil)*

2.2 Der Java EE Applikation Server

Unter Java EE versteht man ein Bündel an Spezifikationen, also kein Programm oder fertige Komponenten, die man irgendwo herunterladen und verwenden könnte. An dieser Stelle kommen die Java EE Applikation Server zum Einsatz. Ein Java EE Applikation Server ist die zentrale Ablaufumgebung für Java Enterprise Applikationen. Ein Java EE Applikation Server ist wiederum selbst eine Java Anwendung, die die oben erwähnten Spezifikationen (Schnittstellen + Funktionalitäten) ausimplementiert hat und diese nach außen hin als Dienste anbietet.

Die Spezifikationen geben zwar Aufschluss darüber, wie diese Schnittstellen aussehen bzw. definiert werden müssen, sie geben aber keine Richtlinie vor, wie die Hersteller die Implementierung vornehmen müssen. Aus diesem Grund haben sich verschiedene Hersteller von Applikation Servern unterschiedlich am Markt positioniert.

Da sich aber alle Hersteller an die Spezifikationen und somit an die vorgeschriebenen Schnittstellen und Funktionalitäten halten müssen, können Java Enterprise Applikationen nicht nur auf einem beliebigen Applikation Server entwickelt werden, sondern auch bis auf ein paar kleine Konfigurationsunterschiede zum Ablaufen gebracht werden.

Einen JavaEE-Applikationserver kann man sich als umhüllenden Container vorstellen, indem andere Container (EJB-Container, Servlet-Container, Webserver-Container) enthalten sind. Desweiteren werden unterschiedlichste Dienste wie z.B. Transaktionssteuerung, Security Management, Persistenzmanagement, Monitoring, bzw. Kommunikationsprotokolle angeboten. Die Protokollpallette umfasst: HTTP, HTTPS, RMI/JRMP, RMI/IIOP, und SOAP über HTTP. *(Oliver Ihns, S59/58), (Weil, S.15/16)*

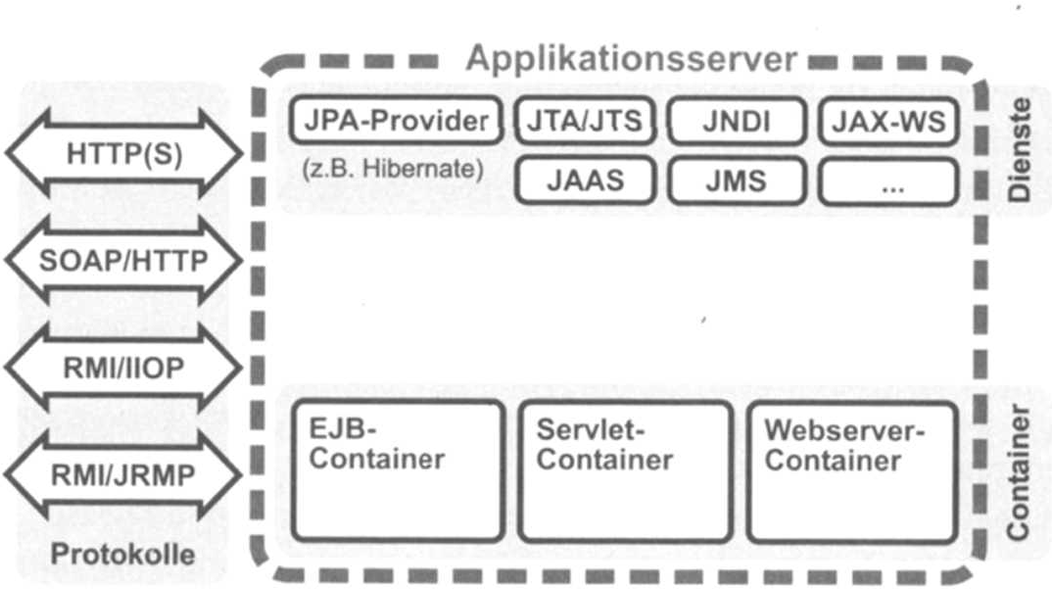

Abbildung 2: Java EE Applikationsserver, entnommen aus (Oliver Ihns, S.60)

2.3 Der EJB-Container:

Der EJB-Container ist die Laufzeitumgebung für EJB Komponenten. Er beherbergt, verwaltet, erzeugt und überwacht wenn nötig, EJB-Komponenten. Dazu stellt er die benötigten Funktionen entweder selbst bereit, oder bezieht diese vom Applikationsserver.

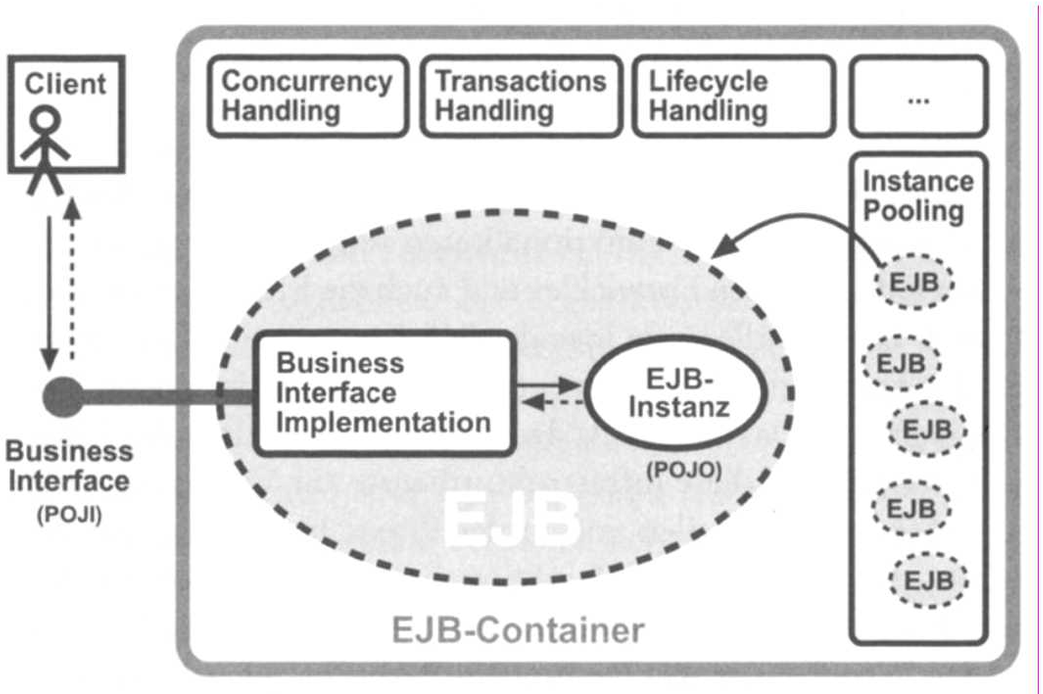

Abbildung 3: Der EJB-Container, entnommen aus (Oliver Ihns, S.61)

Persistenzmanagement:

Vor EJB 3.0 steuerte der EJB-Container die Synchronisation von EJB-Komponenten mit der Persistenzschicht. Mittlerweile übernimmt das die sogenannte Java Persistence API, die als Dienst vom Applikation Server zur Verfügung gestellt wird.

Transaktionssteuerung:

Grundsätzlich bekommen EJB-Komponenten eine komplette Transaktionssteuerung vom EJB-Container, bzw. vom Applikation Server zur Verfügung gestellt. Diese kann man entweder deklarativ über Annotationen oder programmgesteuert nutzen.

Bei der ersten Variante kann das transaktionale Verhalten entweder auf EJB Ebene, oder bis runter auf Methodenebene, deklarativ festgelegt werden. Der EJB-Container kümmert sich dann zur Laufzeit darum, dass das gewünschte Verhalten umgesetzt wird.

Beim programmgesteuerten transaktionalen Verhalten muss der Entwickler diese Logik selbst im Code implementieren. Die einzige Unterstützung, die der Entwickler hier vom EJB-Container bekommt, ist die Nutzung der standardisierten Java Transaction API kurz JTA. *(Oliver Ihns, S. 63)*

Konkurrierende Zugriffe:

Konkurrierende Zugriffe sind in der Regel schwer zu handhaben und erfordern bei der Implementierung vor allem viel Erfahrung und Weitsicht. Enterprise Anwendungen sollten gerade dafür ausgelegt sein, mit einer Vielzahl an konkurrierenden Clients zurechtzukommen. Das könnte zu einer sehr komplexen und fehleranfälligen Logik führen. Zum Glück muss man sich um die Steuerung von konkurrierenden Zugriffen nicht selbst kümmern. Der EJB-Container kümmert sich in der Regel vollständig um die Überwachung und Steuerung von konkurrierenden Zugriffen und stellt sicher, dass bei zeitgleichen Zugriffen durch Clients auf Objekte, bzw. EJB Komponenten, die entsprechenden Daten konsistent und korrekt sind. *(Oliver Ihns, S.64)*

Zugriffsrechte:

Für Benutzer in Enterprise Anwendungen stehen im Rahmen sogenannter Benutzerrollen bestimmte Funktionen zur Verfügung. Die eine Rolle (z.B. Administrator) darf mehr, bzw. fast alles, die andere Rolle (z.B.User) darf nur einen gewissen Teil der Gesamtfunktionalität nutzen.

Ein solches Verhalten lässt sich auch auf EJB-Komponenten übertragen. Dazu wird das Verhalten auf EJB oder Methodenebene deklarativ festgehalten.

Zur Laufzeit wertet der EJB-Container diese Zugriffsrechte und Sicherheitsmerkmale aus, und setzt diese entsprechend um. Dazu nutzt er den Java Authentification and Authorization Service kurz (JAAS). *(Oliver Ihns, S.65)*

Steuerung des Lebenszyklus von EJB-Komponenten:

Um den Lebenszyklus der verschiedenen EJB-Komponenten von der Erzeugung bis zu ihrer Zerstörung kümmert sich der EJB-Container. Auch das gesamte Laufzeitverhalten wird vom EJB-Container verwaltet.

Der EJB-Container erzeugt Instanzen, wenn diese vom Client benötigt werden, zerstört, bzw. löscht nicht mehr benötigte Instanzen. Um Ressourcen zu sparen, ist es dem EJB-Container erlaubt, Instanzen zu passivieren, das heißt, diese auf einen Sekundärspeicher (z. B. Festplatte) auszulagern, und wenn diese wieder benötigt werden zu aktivieren. Für diesen Vorgang müssen die entsprechenden EJB's (Stateful Session Beans) allerdings das Interface Serializable implementieren. *(Oliver Ihns, S. 65/67)*

Instanz Pooling:

Aus Performancegründen kann der EJB-Container ein sogenanntes Instanz Pooling verwenden. In Instanz Pools hält der EJB-Container Instanzen von EJB-Komponenten vor. Solange genügend EJB Instanzen im Pool vorhanden sind, um alle Client Anfragen ordnungsgemäß abzuarbeiten, müssen keine neuen erzeugt werden. Durch das Instanz Pooling kann der Container mit einer geringen Anzahl an Stateless Session Bean Exemplaren sehr viele Clients ressourcenschonend bedienen. Die gesamte Steuerung des Instanz Pooling wird vom EJB-Container übernommen. Diese Funktionalität gilt allerdings nur für EJB's, die keinen clientspezifischen Zustand über einen längeren Zeitpunkt vorhalten müssen. *(Oliver Ihns, S. 67)*

2.4 Deployment Deskriptor & Annotationen

Der Deployment Deskriptor ist eigentlich eine einfache XML Datei. Diese XML-Datei ist, bzw. war, das zentrale Element für die Festlegung von Laufzeit- und Steuerinformationen für EJB Komponenten.

In größeren Projekten wurde so ein Deployment Deskriptor allerdings ziemlich komplex und unübersichtlich. Das führte schließlich auch zu einer enormen Fehleranfälligkeit. Zu guter Letzt wurden Fehler entweder erst zum Deploymentzeitpunkt, oder noch schlimmer, erst zur Laufzeit entdeckt.

Mit EJB 3.X gab es nun die Möglichkeit Laufzeit- und Steuerinformationen mittels Annotationen direkt an die EJB's selbst zu hängen. Das erhöhte zum einen die Lesbarkeit, zum anderen können die Konfigurationsinformationen, die mithilfe von Annotationen gesetzt wurden, zur Übersetzungszeit überprüft und ausgewertet werden.

Grundsätzlich sind die hinterlegten Informationen zur Konfiguration von EJB's im Deployment Deskriptor oder mit Annotationen, nichts anderes als Meta Daten zu den entsprechenden EJB's. Mithilfe dieser Metainformationen kann man, vereinfacht ausgedrückt, das gewünschte Verhalten von EJB's dem Container mitteilen.

Trotz der Konfigurationsmöglichkeit über Annotationen hat der Deployment Deskriptor immer noch seine Daseinsberechtigung. Für einige EJB-Technologien ist er sogar unabdingbar. Außerdem übersteuern, bzw. überschreiben Angaben im Deployment Deskriptor die Angaben der Annotationen. *(Oliver Ihns, S.49/50/51)*

2.5 Convention over Configuration

Convention over Configuration oder Convention by Exception, beide beschreiben das gleiche Konzept, das mit EJB 3.1 Einzug gehalten hat. Das Ziel war, den EJB-Komponenten ein standardisiertes Verhalten mitzugeben. Dadurch sollten sich die Entwicklungs- und Konfigurationszeiten reduzieren.

Jedes EJB hat also von Anfang an ein bestimmtes Standardverhalten. Abgeleitet wurde dieses Standardverhalten aus den Best Practices, bzw. den Entwurfsmustern, die sich in der EJB-Entwicklergemeinde herauskristallisiert haben.

Der Entwickler soll jetzt also nicht jedes Verhalten selbst konfigurieren, sondern nur noch in Ausnahmefällen. Deswegen auch der Begriff Convention by Exception. *(Oliver Ihns, S.54)*

2.6 Dependency Injection & Inversion of Control

Inversion of Control wird oft mit dem sogenannten Hollywood-Prinzip „Don't call us, we'll call you" in einen Context gebracht. Dieser einfache Vergleich soll das Prinzip von Inversion of Control verdeutlichen. Grundsätzlich ist Inversion of Control eine allgemeine Eigenschaft von Frameworks. Hinter dem Begriff, der von Raplh E. Johnson in den 1980er Jahren geprägt wurde, versteckt sich ein einfaches Prinzip. Es besagt, dass die Verantwortlichkeiten bei der Programmausführung, beim benutzten Framework selbst liegen und nicht bei den damit entwickelten Komponenten. Typischerweise verlangen solche Frameworks, dass bestimmte Callback-Methoden implementiert werden. Über diese Methoden kann das Framework zur Laufzeit Informationen in Klassenattribute injizieren oder ein bestimmtes Verhalten auslösen. Dabei verwaltet das Framework auch den Lebenszyklus über die Callback-Methoden.

Dependency Injection ist eine spezielle Variante von Inversion-of-Control. Die ersten Frameworks, die Dependency Injection verwendeten, waren Spring und PicoContainer. Mithilfe dieser Frameworks war es möglich, Injektionen auf bestimmte Referenzen oder auf benötigte Ressourcen, wie Data Sources, durchzuführen. Martin Fowler unterscheidet in seinem Artikel, in dem er Inversion-of-Control und Dependency Injection erstmals in Beziehung setzte, in drei verschiedene Ausführungen:

- Constructor Injection

- Setter Injection

- Interface Injection

Anhand der Namen lässt sich ableiten, auf welcher Ebene die Injektion der Informationen erfolgt. *(Oliver Ihns, S.44/S45), (Weil, S.17/18/19)*

2.7 Session Beans:

Session Beans sind klarer Bestandteil der Geschäftslogik, das heißt, dass innerhalb der Session Beans die Geschäftslogik einer Java Enterprise Applikation implementiert wird. Die Logik kann in der Regel beliebig komplex sein, und reicht von einfachen Berechnungen, bis zu ganzen Geschäftsprozessen. Session Beans sind keine persistenten Objekte. Sie leben in der Regel im Hauptspeicher und werden nicht dauerhaft gespeichert. Um flüchtige Objekte, wie Session Beans, dauerhaft zu speichern, benutzt man im Rahmen der Java Persistent API sogenannte Persistent Entities[1]. Session Beans liegen in drei verschiedenen Ausprägungen vor: „stateful“ (zustandsbehaftete), „stateless“ (zustandslose) und „singleton“ (existieren nur einmal je Applikation). Der Unterschied der verschiedenen Session Beans liegt in der Beziehung zum Client, also zum Benutzer/Aufrufer des jeweiligen Session Beans. EJB-Methoden, also auch Session Bean Methoden werden, allerdings nie direkt aufgerufen, sondern über einen Proxy, ein sogenanntes Hüll Objekt. Der Proxy hat die volle Kontrolle über das eigentliche EJB, bietet die selbe Schnittstelle wie das EJB und ist somit für den Client nicht zu unterscheiden. Im folgenden Abschnitt werden die drei oben erwähnten Session Beans genauer beleuchtet. Besonders ausführlich wollen wir hierbei die in EJB 3.1 neu hinzugekommenen Singleton Beans behandeln. *(Oliver Ihns 2011, S.97), (Weil, S.237/238), (Werner Eberling, S.37/38)*

2.7.1 Stateless Session Beans

Stateless Session Beans sind zustandslose Beans. Das bedeutet allerdings nicht, dass eine Stateless Session Bean überhaupt keinen Zustand besitzt, sondern, dass das Stateless Session Bean keine Informationen über den Client speichert. Die Verbindung zwischen Client und dem Stateless Session Bean besteht genau für die

[1] Persistent Entities modellieren die dauerhaften (persistenten) Daten einer Java EE Applikation.

Dauer eines Methodenaufrufs. Der Zustand eines Stateless Session Beans ist also vor und nach jedem Methodenaufruf identisch. Aus der Sicht des Client sind die Stateless Session Beans nicht zu unterscheiden. Das bedeutet, dass es für den Client nicht relevant ist, immer mit dem gleichen Stateless Session Bean Object zu kommunizieren. Diese Eigenschaften eignen sich hervorragend für ein sogenanntes Instanz Pooling. Das heißt, dass Anfragen verschiedener Clients, an Instanzen aus einem Pool von Stateless Session Bean Exemplaren weitergeleitet werden und zwar an die Instanz, die gerade frei ist. Nach der jeweiligen Bearbeitung wird das Session Bean wieder in den Pool entlassen. *(Oliver Ihns S.97/98/99), (Weil, S. 238/239), (Werner Eberling, S.39/40)*

Die folgende Abbildung soll diesen Prozess verdeutlichen:

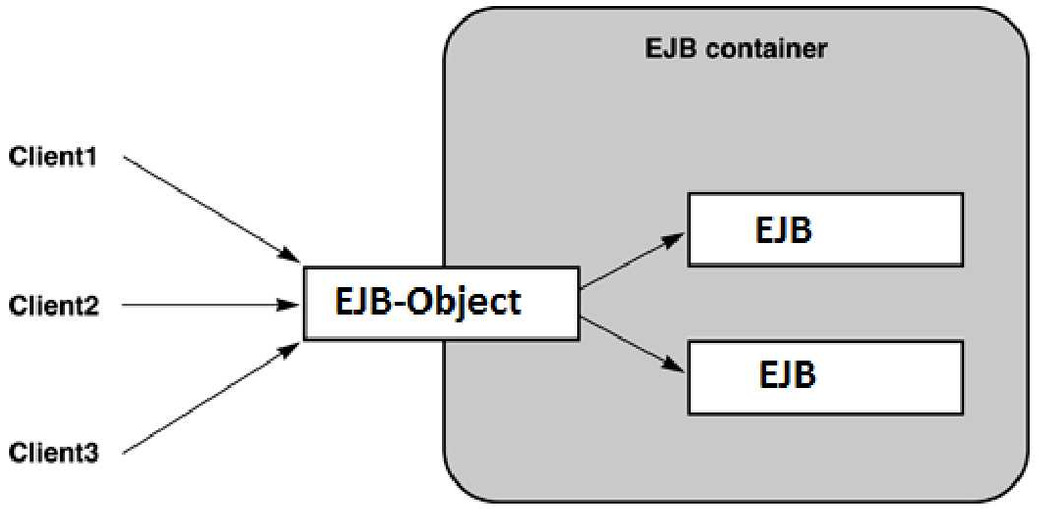

Abbildung 4: Stateless Session Bean, in Anlehnung an *(tversu)*

```java
package de.tbartl.java.verteilte.anwendung.example;

import java.io.Serializable;
import javax.ejb.Stateless;

@Stateless
public class CalculaterService implements Calculator, Serializable {

    public double calculateArea (double length, double width){
        return length * width;
    }

}
```

Listing 1: Stateless Session Bean

2.7.2 Stateful Session Beans

Der Unterschied zwischen Stateless und Stateful Session Beans ist, dass eine Stateful Session Bean einen clientspezifischen Zustand speichern kann. Die Bean merkt sich sogar den Zustand für den jeweiligen Client über mehrere Methodenaufrufe hinweg. Somit ist jedem Client ein Stateful Session Bean zugeordnet.

Die folgende Abbildung soll das verdeutlichen:

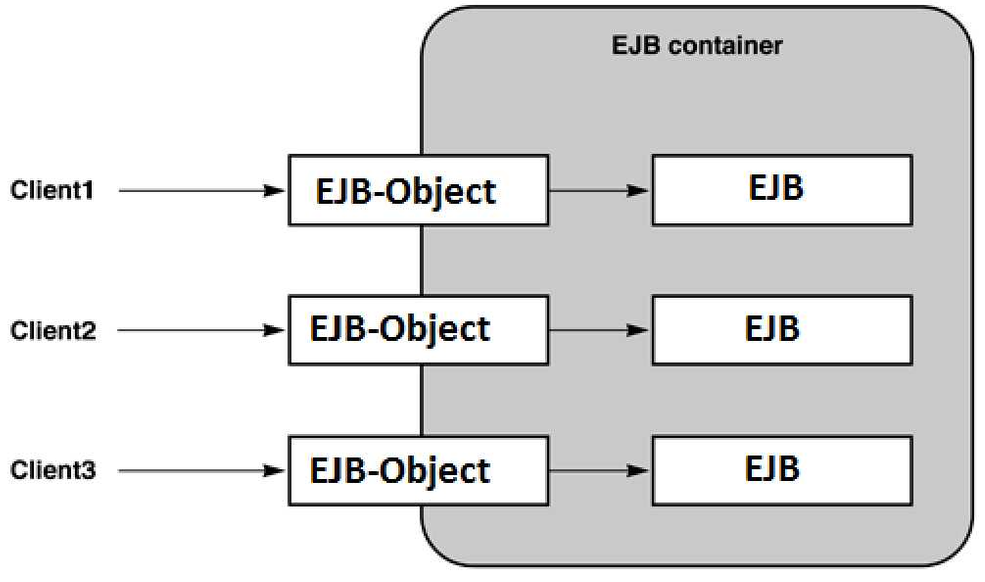

Abbildung 5: Stateful Session Bean, in Anlehnung an *(tversu)*

Dabei bestimmt der aufrufende Programmcode, also der Client, die Sitzungsdauer. Die Sitzung beginnt mit dem ersten Zugriff auf das Stateful Session Bean und endet mit einem optional definierten Timeout, der mit der @StatefulTimeout (javax.ejb.StatefulTimeout) Annotation erzwungen werden kann, oder mit einer vom Client aufgerufenen Methode, die mit @Remove (javax.ejb.Remove) gekennzeichnet ist. Mit dem Aufruf solch einer Methode wird die Verbindung zwischen dem Bean und dem Client getrennt. Danach wird die Bean vom Container zerstört. Die Methode muss an sich keine Programmlogik enthalten, sie kann sogar vollkommen leer sein. Es reicht, wenn diese mit @Remove gekennzeichnet ist.

Bei schwerwiegenden Fehlern, wie z. B. einer System Exception, wird das Stateful Session Bean ebenfalls vom Container verworfen.

Unter Conversational State versteht man alle Informationen, die den Dialog zwischen Stateful Session Bean und Client betreffen, also den clientspezifischen Zustand. Die EJB Spezifikation definiert den Conversational State wie folgt: Der Conversational State einer Stateful Session Bean ist die Menge aller Attribute einer Bean-Instanz und die transitive Hülle aller Objekte, die durch Java-Referenzen von der Bean Instanz erreichbar sind. Einfach ausgedrückt bedeutet das, dass der Conversational State nicht nur aus den einzelnen Attributen der eigentlichen Bean Instanz besteht, sondern auch aus den Attributen eines referenzierten Java Objects. Falls es noch weitere Beziehungen zu anderen Java Objekten gibt, werden die Referenzen vom Container rekursiv weiterverfolgt, bis diese vollkommen aufgelöst sind.

Stateful Session Beans werden in der Praxis häufig eingesetzt. Ein typischer Anwendungsfall wäre z.B. die Realisierung eines Warenkorbs. Allerdings sind Stateful Session Beans durch ihre speziellen Eigenschaften sehr ressourcenbelastend. Deswegen ist es dem EJB Container erlaubt den Conversational State einer Stateful Session Bean vorrübergehend auszulagern. Diesen Vorgang nennt man Aktivierung und Passivierung. Damit dies reibungslos ablaufen kann, sollten Stateful Session Beans serialisierbar sein, das heißt das Interface Serializable implementieren. *(Oliver Ihns S.99/100/101/102), (Weil S. 238/239), (Werner Eberling, S.59/60)*

2.7.2.1 Aktivierung Passivierung

In Enterprise Anwendungen, z. B. in einem Onlineshop kann es vorkommen, dass sehr viele Clients eine Verbindung zu Stateful Session Beans aufbauen. Zum Beispiel weil jeder Benutzer seinen eigenen Warenkorb zusammenstellt und die Enterprise-Applikation diesen Warenkorb für den jeweiligen Client im Hauptspeicher vorhalten muss. Dies kann allerdings sehr schnell sehr Ressourcen belastend werden. Deswegen ist es dem EJB-Container erlaubt, den Conversational State einer Stateful Session Bean zu passivieren, das heißt auf einen Sekundärspeicher, z. B. einer Festplatte auszulagern.

Wie genau, bzw. welche Bean-Instanzen ausgelagert werden, hängt vom jeweiligen Hersteller des EJB-Container ab. In der Regel wird der EJB-Container aber die Instanzen auswählen, die am längsten nicht mehr benutzt wurden. Diese ist als Least-Recently-Used (LRU-) Strategie bekannt.

Wenn die Stateful Session Bean später wieder gebraucht wird, weil sich z. B. der gleiche Benutzer zu einem späteren Zeitpunkt entscheidet, seinen Einkauf fortzusetzen, wird der Conversational State des jeweiligen Stateful Session Bean vom Sekundärspeicher zurück in den Hauptspeicher übertragen. Diesen Vorgang nennt man Aktivierung. Der EJB-Container stellt dann die Verbindung zwischen dem Client und dem Stateful Session Bean wieder her, und der Einkauf kann wie gewohnt fortgesetzt werden. *(Oliver Ihns S.101/102)*

Der gesamte Aktivierungs- / Passivierungsvorgang ist serialisiert.

```java
@Stateful
public class ShoppingCartService implements ShoppingCart, Serializable {

    private List<String> shoppingCart;

    @PostConstruct
    private void initialize(){
        shoppingCart = new ArrayList<String>();
    }

    public void addArticel(String articel){
        shoppingCart.add(articel);
    }

    public List<String> getAllArticels (){
        return shoppingCart;
    }

}
```

Listing 2: Stateful Session Bean

In Zusammenhang mit JSF, einer Managed Bean, CDI und der Stateful Session Bean lässt sich ein rudimentärer Warenkorb programmieren:

Abbildung 6: Realisierung eines rudimentären Warenkorbs mit Stateful Session Bean, Managed Bean und JSF

2.7.3 Singleton Session Bean

Mit der Singleton Session Bean hält eines der meistbekannten Entwurfsmuster aus der Java SE Welt in die Java EE Welt Einzug. Das Singleton Pattern besagt, dass von einer bestimmten Klasse höchstens eine Instanz erzeugt werden kann.

Vor EJB 3.1 mussten Entwickler, die dieses Funktionalität nutzen wollten, auf proprietäre Lösungen einzelner EJB-Container-Hersteller zurückgreifen oder aber selber nach dem Java-SE-Singleton Pattern implementieren. Der Nachteil ist, dass man dadurch die vom EJB-Container bereitgestellten Funktionalitäten und Dienste, wie Zugriffssteuerung, Transaktionalität und Skalierbarkeit verliert.

Seit Java EE 6 bzw. EJB 3.1 wurden die Session Beans um die Singleton Bean erweitert. Damit lässt sich sicherstellen, dass innerhalb einer Applikation genau eine Instanz dieser Session Bean vorhanden, bzw. vom EJB-Container zugelassen wird. Laufen EJB-Container geclustert, wird eine Instanz einer Singleton Session Bean pro JVM erzeugt.

Singleton Session Beans werden in der Regel dazu verwendet, um einen applikationsweiten Zustand herzustellen. Ergebnisse von häufigen oder aufwendigen Abfragen lassen sich so in entsprechenden Caches oder Konstanten zentral speichern und der gesamten Applikation zur Verfügung stellen.

Die Verbindung zwischen Client und einer Session Bean gilt nur für die Dauer eines Methodenaufrufs.

Außerdem ist es möglich, eine Singleton Session Bean beim Starten der Applikation zu instanziieren. Dadurch kann man die Singleton Session Bean gut dafür nutzen, Aufgaben beim Laden und Beenden einer Applikation durchführen zu lassen. Mit gewissen Abhängigkeiten, die sich innerhalb einer Singleton Session Bean definieren lassen, ist es möglich abhängige Beans erst dann instanziieren zu lassen, wenn die Singleton Session Bean bereits erzeugt wurde. *(Oliver Ihns, S.103/104), (Weil, S.238/239)*

Singleton Session Beans und Transaktionen:

Singleton Session Beans können an Transaktionen teilnehmen. Optionale Callback-Methoden, von Singleton Session Beans, bei der, der Container die Transaktionssteuerung verwaltet, haben laut dem Configuration by Exception Ansatz dasselbe Transaktionsverhalten wie die Geschäftsmethode der aufrufenden Session Bean. Hier stehen allerdings nur die Transaktionstypen „Required“, „Requires-New“ „NotSupported“ zur Verfügung.

Das Transaktionsverhalten kann per Annotation @TransactionAttribute (javax.ejb.TransactionAttribute) oder im Deployment-Deskriptor angegeben werden. *(Oliver Ihns, S.103/104), (Weil, S.238/239)*

Singleton Session Beans und Nebenläufigkeit:

Per Default, können Geschäftsmethoden einer Singleton Bean von mehreren Clients gleichzeitig durchlaufen werden. Dieses Standardverhalten lässt sich aber entweder für die gesamte Singleton Bean, oder je Methode, über die Annotationen, @Lock (javax.ejb.Lock) und @LockType (javax.ejb.LockType) oder mit einem entsprechendem Eintrag im Deployment-Deskriptor ändern. *(Oliver Ihns, S.103/104), (Weil, S.238/239)*

Die folgende Abbildung soll das verdeutlichen:

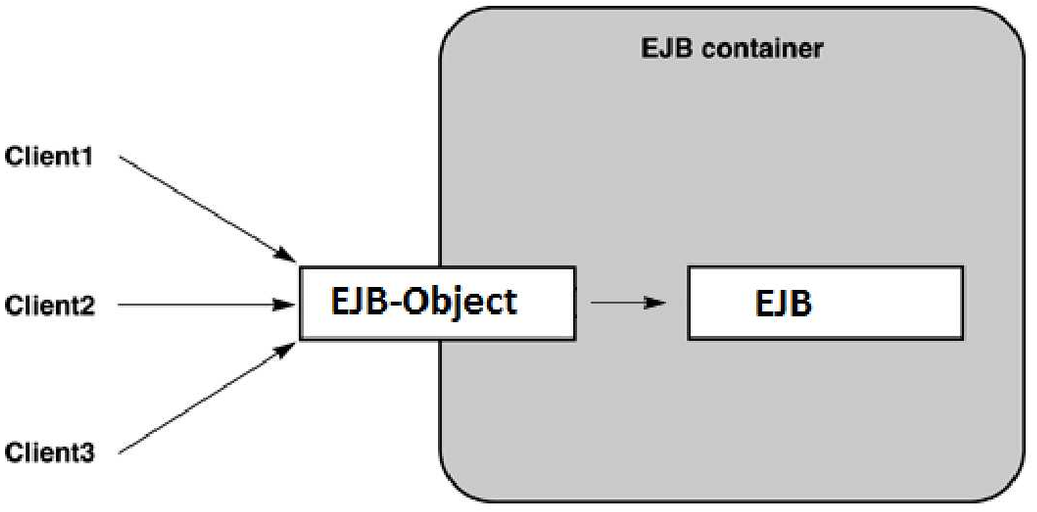

Abbildung 7: Singleton Session Bean, in Anlehnung an *(tversu)*

```java
package de.tbartl.java.verteilte.anwendung.example.accessCounter;

import java.io.Serializable;

@Singleton
@Startup
public class AccessCounterService implements AccessCounter, Serializable {

    private int accessHits = 1;

    public int getAccessHits() {
        return accessHits++;
    }


}
```

Listing 3: Singleton Session Bean die direkt beim Applikationsstart erzeugt wird.

Mithilfe dieser Singleton Session Bean, einer Managed Bean, und einer JSF Datei lässt sich z.B. ein Zähler für Webseitenzugriffe implementieren.

Abbildung 8: Zugriffszähler für Webseite

3 Context and Dependency Injection

Context and Dependency Injection, das unter der Spezifikation JSR 299 bekannt ist, ist eine schichtenübergreifende typsichere Art von Dependency Injection mit kontextbasiertem Lebenszyklusmanagement. Das Ziel von CDI, bzw. die Hauptaufgabe, ist die Standardisierung und Vereinfachung des Zusammenspiels zwischen der Präsentationsschicht (JSF) und der EJB-Technologie mit der implementierten Geschäftslogik. Wie genau CDI funktioniert und wie man das Zusammenspiel mit EJB's realisiert schauen wir uns im nächsten Kapitel an. *(Oliver Ihns, S. 495), (Weil, 17/18/19)*

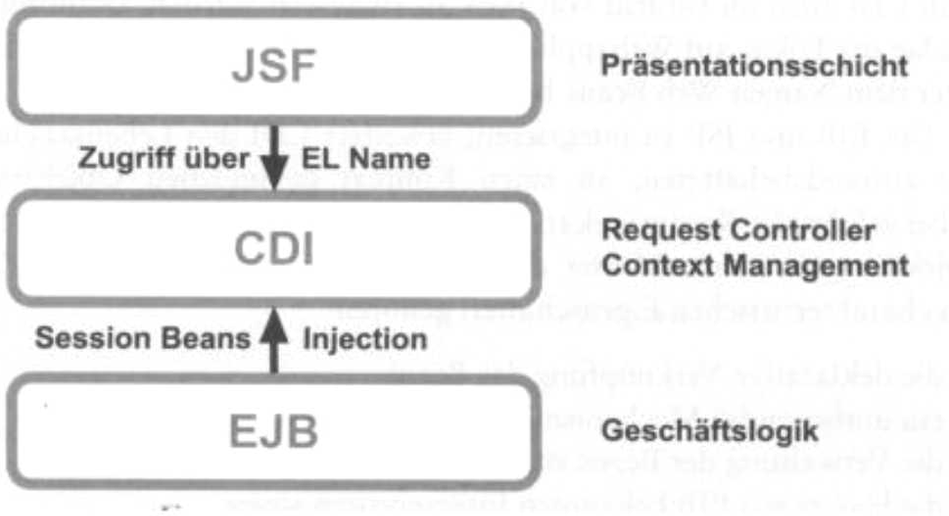

Abbildung 9: Schichtenübergreifendes Zusammenspiel, entnommen aus (Oliver Ihns, S. 496)

3.1 Das CDI Grundkonzept:

Java EE Applikationen sind nicht monolithisch aufgebaut, sondern bestehen in der Regel aus verschiedenen Komponenten, die sich gegenseitig nutzen.

Die folgende Abbildung soll das Ganze verdeutlichen:

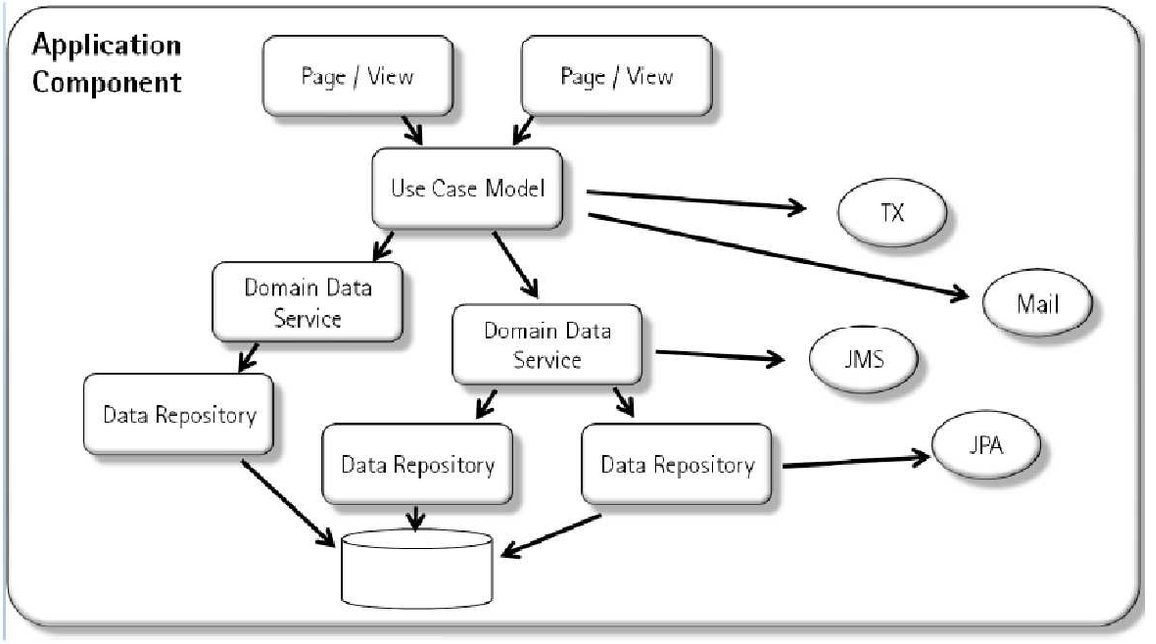

Abbildung 10: Anwendungskomponenten entnommen aus (Weil, S.16)

Java EE Applikationen bestehen in der Regel sogar aus verwalteten Komponenten, sogenannten Managed Components (Servlets, JSPs, JSF Managed Beans, Enterprise Java Beans). Um die Verwaltung dieser Komponenten kümmert sich der Container, das heißt, der Container erzeugt/zerstört benötigte oder nicht mehr benötigte Komponenten. Der Container kümmert sich auch während der Lebenszeit um die Kommunikation der Komponenten untereinander. Er informiert die Komponenten über ein- und ausgehende Ereignisse oder stellt ihnen bestimmte Dienste bereit. Auf der anderen Seite gibt es in Java EE noch sogenannte POJO's (Plain Old Java Objects = ganz normale Java Objekte). Diese Objekte werden nicht vom Container verwaltet.

Damit die verwalten Komponenten untereinander, sowie die verwalteten Komponenten und die POJO's miteinander kommunizieren können, müssen diese

untereinander verknüpft werden. Normalerweise würde eine bestimmte Komponente mit dem Schlüsselwort „new“ eine andere Komponente instanziieren. Das bedeutet allerdings, dass wir genau darüber Bescheid wissen müssen, was das für ein Objekt ist, bzw. wo das Objekt zu bekommen ist und wie es erzeugt werden muss. Das führt zu einer starken Kopplung der Komponenten, wodurch Änderungen wesentlich fehleranfälliger und aufwendiger zu realisieren sind.

Mit Context and Dependency Injection, kurz CDI haben wir jetzt die Möglichkeit, durch das Inversion of Control Konzept die Verantwortung für diese Referenzen auf benötigte Objekte auf den Container abzuschieben.

Die zentrale Komponente in Context and Dependency Injection, kurz CDI, ist der Container. Ein Container im Java EE Umfeld ist ein Behälter für Objekte, die vom Container selbst verwaltet werden. Diese Objekte können dem Container hinzugefügt und wieder entfernt werden. Der Kontext beschreibt die Beziehungen der Objekte, die in einem Container verwaltet werden. Dependency Injection ermöglicht dem Container, Objekte zur Laufzeit zu erzeugen, entsprechende Referenzen dieser Objekte zu injizieren und wenn nötig diese Objekte wieder zu zerstören. Dependency Injection kurz DI ist an sich nichts Neues. CDI bietet allerdings etwas mehr als manch andere Dependency Injection Frameworks, z. B. eine typsichere Auflösung. Mithilfe des implementierten Typesafe-Resolution Algorithmus wird ein Entwickler sehr früh, spätestens aber zur Deployment Zeit informiert, wenn sich ein Injektionspunkt nicht auflösen lässt. Außerdem kann man nun grenzübergreifend innerhalb einer Java EE Applikation zwischen Frontend und Backend auf dieses Konzept zurückgreifen. Die Beans lassen sich direkt in ein Attribut oder als Parameter eines Konstruktors oder Setters injizieren.

Um aus der Präsentationsschicht (z. B. JSF) direkt mit den Beans sprechen zu können, wurde die sogenannte Unified Expression Language entwickelt. Dadurch ist es möglich, einfach über den Bean-Namen mit diesen zu kommunizieren.

Scopes definieren im Rahmen von CDI die Sichtbarkeit von Variablen und Klassen sowie die Lebensdauer von Objekten innerhalb des Containers. *(Oliver Ihns, S.497), (Weil, 17/18/19)*

CDI unterstützt vier Scopes:

Scope	Beschreibung
Request Scope	Lebt nur für die Dauer eines HTTP Requests.
Conversation Scope	Lebt für eine vom Entwickler festgelegte „Gesprächsdauer“, die mehrere HTTP-Requests umfassen kann. Beispielsweise kann in einem Shopsystem die Conversation mit dem ersten Einlegen eines Artikels in den Warenkorb gestartet und mit dem Abschließen des Bestellvorgangs beendet werden.
Session Scope	Lebt für die Dauer einer eindeutig einem Benutzer zugeordneten HTTP-Session.
Application Scope	Lebt, solange die (Web-)Applikation läuft.

Tabelle 1: CDI Scopes, entnommen aus *(Oliver Ihns S.497)*

Die Beans werden entsprechend ihrem Scope, der durch simple Annotationen realisiert werden kann, an den gleichnamigen Kontext gebunden. *(Oliver Ihns, S.497)*

Nachfolgend sehen Sie eine Auflistung der entsprechenden Annotationen:

- @RequestScoped (javax.enterprise.context.RequestScoped),
- @ConversationScoped (javax.enterprise.context.ConversationScoped)
- @SessionScoped (javax.enterprise.context.SessionScoped)

- @ApplicationScoped (javax.enterprise.context.ApplicationScoped)

3.2 Das Bean Verständnis von CDI

Unter Beans im Zusammenhang mit CDI fallen Managed Beans und EJB Session Beans. Message Driven Beans, Interzeptoren, Servlets, JAX WS Services, JSP Tag Handler oder Tag Library Event Listener, sind keine Beans im Sinne von CDI, allerdings können diese Beans injiziert bekommen. *(Oliver Ihns, S.497), (Weil, 20/21/22)*

3.2.1 Managed Beans

Beans die nicht speziell, mittels Annotationen gekennzeichnet sind, werden von CDI automatisch als Managed Bean interpretiert. Allerdings sollte es eine konkrete Klasse darstellen, oder mit der @Decorator (javax.Decorator) Annotation ausgewiesen sein, bzw. einen parameterlosen oder einen mit @Inject (javax.inject.Inject) annotierten Konstruktor besitzen. Außerdem sollte es keine statische innere Klasse sein. Falls die Bean, aus dem Enterprise Java Beans Umfeld stammt, wird diese ebenfalls nicht als Managed Bean interpretiert. Und zu guter Letzt sollte keine SPI-Erweiterung vorliegen. *(Oliver Ihns, S.501), (Weil, 184/185)*

3.2.2 Session Beans

Session Beans entsprechen im Aufbau exakt CDI Beans. Allerdings verwaltet der EJB-Container bereits den Lebenszyklus der Session Beans. Deswegen versteht CDI, Session Beans als eigenen Bean Typ.

Session Beans haben eine Scope Besonderheit. Bei Stateless und Singleton Session Beans muss man den Scope nicht explizit angeben, da der EJB-Container den Lebenszyklus sowieso schon verwaltet, bzw. steuert. Bei den Stateful Session Beans hingegen gibt es die Möglichkeit, die Verbindung zum Client durch den Aufruf einer mit @Remove annotierten Methode vorzeitig, oder ordnungsgemäß zu unterbrechen. Im CDI Kontext funktioniert dieses Verhalten allerdings erst wieder, wenn die Stateful Session Bean zusätzlich mit der Annotation @Dependent

(javax.enterprise.context.Dependent) annotiert wurde. *(Oliver Ihns, S.501/502), (Weil, 238/239)*

Source Code Beispiel ohne CDI:

```
package de.tbartl.java.verteilte.anwendung.example.cdi;

import java.io.Serializable;


public class ActualDateService implements ActualDate, Serializable {

    public String dateToString (){
        Date date = new Date();
        return date.toString();
    }
}
```

Listing 4: „ActualDateService“ – POJO (Ganz normales Java Object)

```
package de.tbartl.java.verteilte.anwendung.example.cdi;

import java.util.List;

public interface ActualDate {

    public String dateToString ();

}
```

Listing 5: „ActualDate“ Interface

Der „ActualDateService“ (POJO) wird über das Interface „ActualDate“ angesprochen und instanziiert.

```
package de.tbartl.java.verteilte.anwendung.example.cdi;
import java.io.Serializable;

@Named
@SessionScoped
public class ActualDateManager implements Serializable {

    private ActualDate acutalDateService = new ActualDateService();

    private String date;

    public String getDate() {
        return date;
    }

    public void setDate(String date) {
        this.date = date;
    }

    public String getActualDate(){
        return acutalDateService.dateToString();
    }

}
```

Listing 6: CDI Bean/Managed Bead, ohne CDI.

Da es sich bei dem „ActualDateService“ um ein POJO, also ein ganz normales Java Object handelt, wird es standardmäßig nicht vom Container verwaltet. Deshalb müssen wir es mit dem „new“ Operator instanziieren. Wir müssen also genau wissen, welche Klasse das Interface „Actutal Date“ implementiert. Dadurch entsteht eine starke Kopplung zwischen den Komponenten, also dem Managed Bean und dem „ActualDateService“. Falls sich der Name der Klasse irgendwann mal ändern sollte, oder durch eine andere ersetzt wird, steigt der Wartungsaufwand und dadurch die Fehleranfälligkeit.

Source Code Beispiel mit CDI:

Das Interface „ActualDate“ und die Klasse „ActualDateService“ sind unverändert.

```
package de.tbartl.java.verteilte.anwendung.example.cdi;
import java.io.Serializable;

@Named
@SessionScoped
public class ActualDateManager implements Serializable {

    @Inject
    private ActualDate acutalDateService;

    private String date;

    public String getDate() {
        return date;
    }

    public void setDate(String date) {
        this.date = date;
    }

    public String getActualDate(){
        return acutalDateService.dateToString();
    }

}
```

Listing 7: CDI Bean/Managed Bead, mit CDI.

Um CDI zu verwenden, genügt es einfach, unsere Variable „actualDateService“ vom Typ „ActualDate“ mit @Inject zu annotieren. Damit geben wir die Verantwortung für die Instanziierung und den gesamten Lebenszyklus des Objects an den Container ab. Der Container sucht dann zur Laufzeit nach einer passenden Klasse, die das Interface „ActualDate“ implementiert, instanziiert diese, und injiziert uns eine Referenz auf dieses Object in die entsprechende Variable. Durch CDI erreichen wir also eine lose Kopplung unter den Komponenten. Der Client muss lediglich noch wissen, wie das Interface heißt. Um den Rest kümmert sich der Container. Wenn die Klasse „ActualDateService“ irgendwann umbenannt oder ersetzt wird, müssen wir an der CDI Bean/Managed Bean (Table 4) nichts ändern. CDI funktioniert nicht nur mit POJOS, sondern auch mit EJB’s.

3.3 Bean Typen

Über Bean Typen lassen sich Klassen einschränken, die an einem bestimmten Injektionspunkt überhaupt injiziert werden können. Ein BeanTyp ist z. B. eine Klasse oder ein Interface das für den Client sichtbar ist. Es handelt sich also um die vom Client nutzbare Schnittstelle. *(Oliver Ihns, S.503)*

Eine einzelne Bean kann mehrere BeanTypen haben. Das folgende Listing soll das veranschaulichen.

```
public class BandverwaltungImpl extends Verwaltung
implements Bandverwaltung{

}
```

Listing 8: Bean Typen einer Managed Bean entnommen aus *(Oliver Ihns, S.503)*, *(Weil, S.24)*

Die Bean Typen aus dem Listing 7 sind „BandverwaltungImpl“, „Verwaltung“, Bandverwaltung und der implizite Typ „java.lang.Object“

Bei Session Beans zählen grundsätzlich nur die lokalen Interfaces und die Bean Class Local View zu den BeanTypen.

```
import javax.ejb.Stateless;


@Stateless
public class BandverwaltungBean
implements BandverwaltungLocal{

}
```

Listing 9: Bean Typen einer Stateless Session Bean entnommen aus (Oliver Ihns, S.503)

Die BeanTypen aus Listing 9 beschränken sich deshalb auf BandVerwaltungLocal und java.lang.Object.

Allerdings lassen sich mithilfe der Annotation @Typed (javax.entprise.inject.Typed) die BeanTypen einschränken.

```
import javax.enterprise.inject.Typed;


@Typed(Bandverwaltung.class)
public class BandverwaltungImpl extends Verwaltung
implements Bandverwaltung{

}
```

Listing 10: Typed Annotation entnommen aus *(Oliver Ihns, S. 504)*

Somit sind die BeanTypen für Listing 10 auf Bandverwaltung und java.lang.Object beschränkt. *(Oliver Ihns, S. 504)*

3.4 Qualifier

BeanTypen reichen in manchen Fällen nicht aus, z. B. wenn es sich bei dem BeanTyp um ein Interface handelt, zu dem es mehrere verfügbare Implementierungen gibt. Der Container kann dann spätestens zur Deployzeit den Injektionspunkt nicht auflösen, und wirft eine entsprechende Fehlermeldung.

Der Container braucht also genauere Informationen, welche Implementierung er für die Injizierung verwenden soll. Für diesen speziellen Fall gibt es sogenannte Qualifier. Qualifier sind definierbare Annotationen, die mit der Annotation @Qualifier (javax.inject.Qualifier) deklariert sein müssen. Mithilfe dieser Qualifier wird es dem Container ermöglicht, die richtige Bean zu identifizieren.

```
@Qualifier
@Target({TYPE,METHOD,PARAMETER,FIELD})
@Retention(RUNTIME)
public @interface Invoice {}
```

Listing 11: Definition des Qualifier Invoice

Mithilfe der selbst definierten Annotation wird die gewünschte Bean und der Injektionspunkt gekennzeichnet.

```
@Invoice
public class InvoicePaymentTransaction implements PaymentTransaction{

}
```

Listing 12: Bean mit Qualifier Invoice spezifiziert entnommen aus *(Oliver Ihns, S. 504), (Weil, 26)*

```
@Inject
@Invoice
private PaymantTransaction paymentTransaction;
```

Listing 13: Nutzen des Qualifiers Invoice am Injektionspunkt entnommen aus *(Oliver Ihns, S. 504)*

Findet der Container jetzt nur eine Bean, auf den diese selbst definierte Annotation passt, wird der Injektionspunkt ordnungsgemäß aufgelöst und die Anwendung deployt.

Wird keine passende oder werden mehrere passende Beans gefunden, gibt der Server eine entsprechende Fehlermeldung aus. *(Oliver Ihns, S. 504)*

3.5 Alternativen

Mit der Annotation @Alternative (@javax.enterprise.inject.Alternative) lassen sich, abhängig von der Laufzeitumgebung, unterschiedliche Implementierungen eines Interfaces nutzen.

Per Default ist diese Funktionalität deaktiviert. Wenn man diese für gewisse Anwendungen (z. B. Tests) dennoch benötigt, muss man diese erst im CDI-Deployment-Diskriptor „beans.xml“ aktivieren. *(Oliver Ihns, S. 505/506), (Weil, 28/29)*

3.6 Expression Language Name

Um auf eine Bean außerhalb des Java-Codes zuzugreifen z. B. aus einer JSP- oder JSF Seite, muss man auf die sogenannte Expression Language Name zurückgreifen. Mit dieser ist es möglich, die entsprechende Bean über den EL-Namen anzusprechen.

Per Default ist der EL-Name einer Managed Bean immer der Klassenname mit kleingeschriebenen Anfangsbuchstaben. Will man den Default EL-Namen ändern, ist es möglich ihn bei der @Named (javax.inject.Named) Annotation mit anzugeben z. B. @Named(„Beispiel-EL-Name“). *(Oliver Ihns, S. 510)*

Da Interzeptoren und Dekoratoren nicht zum Inhalt dieser Seminararbeit zählen, werden die Themengebiete Interzeptoren und CDI, sowie Dekoratoren und CDI hier nicht betrachtet.

4 Timer Service

In Unternehmensapplikationen ist die zeitgesteuerte, also regelmäßige und automatische Ausführung von Geschäftsprozessen, enorm wichtig. Typische Beispiele sind die regelmäßige Erstellung von Unternehmensreports, oder eine stete Überprüfung von Aktienkursen, oder das anstoßen von bestimmten Backup- oder Archivierungsprozessen. Wenn man solch wiederkehrende Aufgaben realisieren wollte, ohne irgendeine Client-Interaktion, konnte man entweder, auf das aus der UNIX-Welt bekannte „cron“ Tool zurückgreifen, oder eigens angefertigte Batch Skripte direkt in den betriebssystemeigenen Scheduling Mechanismus einhängen.

Mit der Java Enterprise Edition lässt sich so ein Verhalten auch über den Java EE internen Timer Service realisieren. Der Timer Service ist ein Dienst, der über den EJB-Container bereitgestellt wird. Diesen Timer Dienst kann man als Java EE Entwickler entweder programmgesteuert über die entsprechende Schnittstelle des Timer Service ansprechen, oder man kann automatische Timer verwenden. Automatische Timer lassen sich rein deklarativ über entsprechende Annotationen realisieren.

Mithilfe des Java EE internen Timer Service ist es also möglich, bestimmte Aufgaben zu benutzerdefinierten Zeitpunkten einmalig oder periodisch auszuführen. Die Funktionen des Timer Service im Detail schauen wir uns im folgenden Kapitel genauer an. *(Oliver Ihns, S. 461), (Weil, S.250/251)*

4.1 Programmgesteuerte Timer

Bei programmgesteuerten Timern kann man weitaus mehr Einfluss auf die gewünschten Ausführungszeitpunkte nehmen, als bei automatischen Timern. Auch sonstige Kontext-Informationen lassen sich zur Laufzeit besser mit programmgesteuerten Timern ermitteln.

4.1.1 Grundlegende Funktionsweise programmgesteuerter Timer

Wenn eine EJB das Interface TimedObject (@javax.ejb.TimedObject) mit der Methode ejbTimeout() implementiert, oder eine Methode der EJB mit @Timeout (javax.ejb.Timeout) versehen wird, wird die EJB zu einem Timed Object.

Die Bean-Instanzen sind dadurch in der Lage den Timer-Service zu nutzen. Vorher müssen Sie aber über das Interface TimerService (javax.ejb.TimerService) einen neuen Timer (javax.ejb.Timer) erzeugen.

Der TimerService ist ein Dienst, der über den EJB-Container/Applikation Server zur Verfügung gestellt wird.

Wenn der Timer für eine Bean-Instanz abgelaufen ist, informiert der Timer die Bean-Instanz, indem er die Timeout Methode aufruft. *(Oliver Ihns, S. 463)*

Die folgende Abbildung soll das Szenario verdeutlichen:

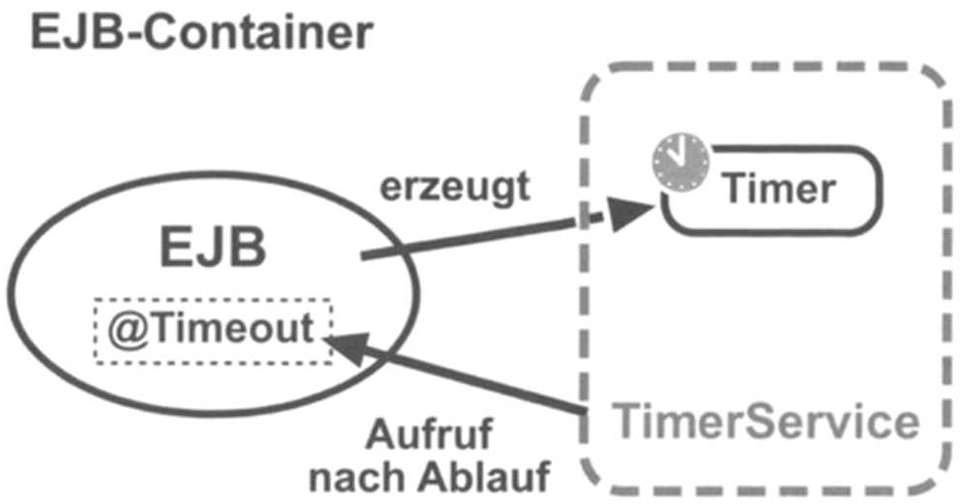

Abbildung 11: Funktionsweise programmgesteuerter Timer, entnommen aus *(Oliver Ihns, S.463)*

4.1.2 Die Timeout Methode

Laut Spezifikation müssen EJB's, die programmgesteuerte Timer nutzen, eine entsprechende Callback-Methode bereitstellen. Dies ist in der Regel eine Methode innerhalb der EJB, die mit der Annotation @Timeout (javax.ejb.Timeout) versehen ist. Solche Methoden dürfen weder als final noch als static deklariert sein, dürfen keinen Rückgabewert besitzen, müssen also vom Typ „void" sein und dürfen nur Parameter vom Typ Timer (javax.ejb.Timer) entgegennehmen.

Methoden, die auf diese Weise gekennzeichnet wurden, werden vom Timer-Service automatisch aufgerufen, sobald ihr Timer abgelaufen ist. Das entspricht einem konkreten Zeitpunkt oder einem definierten Zeitintervall, das nach der Erzeugung des jeweiligen Timers abgelaufen ist. *(Oliver Ihns, S. 465)*

4.1.3 Das Interface Timer Service Interface

Das Interface TimerService (javax.ejb.TimerService) ermöglicht es ebenfalls das EJB zu einem TimedObject zu machen, bzw. auf den Timer-Service des EJB-Container/Applikation Server zuzugreifen.

Eine Bean-Instanz bekommt eine Referenz auf dieses Interface über den Dependency Injection Mechanismus, oder über die getTimerService() Methode der Klasse EJBContext.

In der EJB erzeugt man dann sogenannte Timer-Objekte mit der Methode createTimer() die vom Timer Interface angeboten wird. Mit diesen Timer Objekten kann man sich dann beim Timer-Service registrieren. Der Unterschied zur Timeout Methode ist, dass man hier beliebig viele Timer-Objekte erzeugen kann, das heißt, eine einzige EJB kann sich für eine mehrfache Ausführung zu verschiedenen Zeitpunkten registrieren. *(Oliver Ihns, S. 466)*

Es wird zwischen drei verschiedenen Timer Arten unterschieden:

4.1.4 Single Event Timer

Die Lebenszeit eines Single Event Timer, auch Single Action Timer genannt, beschränkt sich auf die einmalige Ausführung seiner Aufgabe. Single Event Timer werden über die Methode create-SingleActionTimer() erzeugt. Dabei wird entweder der gewünschte Ausführungszeitpunkt als Zeitintervall in Millisekunden oder als absoluter Zeitpunkt (java.util.Date) mitgegeben. *(Oliver Ihns, S. 467)*

4.1.5 Interval Timer

Wie der Name schon sagt, handelt es sich hier um Timer, die stetig aufgerufen und ausgeführt werden. Der Intervall Timer erwartet allerdings zusätzlich zum Ausführungszeitpunkt des Single Event Timer beim Erzeugen durch die createIntervallTimer() Methode einen Parameter, der das Wiederholungsintervall in Millisekunden angibt. Intervall Timer sind aktiv, bis sie explizit mit der cancel() Methode beendet werden. *(Oliver Ihns, S. 468)*

4.1.6 Calendar Schedule Based Timer

Mit EJB 3.1 neu hinzugekommen sind die sogenannten Calendar Schedule Based Timer. Diese ermöglichen es, Zeitpläne für die Ausführung festzulegen. Es lassen sich z. B. spezielle Tageszeiten oder Wochentage definieren. Solche Timer werden mit der Methode createCalendarTimer() erzeugt. Der Zeitplan wird über ein spezielles Objekt vom Typ ScheduleExpression (javax.ejb.ScheduleExpression) übergeben.

Das folgende Listing soll das Ganze verdeutlichen:

Hier wird ein Timer innerhalb einer bestimmten Saison jeden Samstag um 15:30 aufgerufen.

```
ScheduleExpression schedule = new ScheduleExpression();
schedule.month("Aug-May".dayOfWeek("Sat").hour(15).minute(30);
Timer timer = timerService.createCalendarTimer(schedule);
```

Listing 14: Erzeugung eines Timers nach Zeitplan entnommen aus *(Oliver Ihns, S.468)*

Alle Varianten sowohl Single Event Timer, Intervall Timer oder Calendar Schedule Based Timer, benötigen zusätzlich ein weiteres Objekt vom Typ TimerConfig (javax.ejb.TimerConfig). Mithilfe dieses Objekts ist es möglich, über eine bestimmte Methode, ein serialisierbares Objekt zu setzen. Zum einen ist dies wichtig da der EJB-Container dieses Objekt als Ausführungskontext mit dem Timer Objekt assoziiert, zum anderen, da alle Timer per Default persistent sind. Seit EJB 3.1 ist es aber möglich dieses Defaultverhalten mit der Methode setPersistent(false) zu ändern. *(Oliver Ihns, S. 468/469), (Weil, S.250/251)*

4.2 Automatische Timer

Automatische Timer sollten immer dann verwendet werden, wenn man nicht direkt darauf angewiesen ist, den Zeitpunkt der Erzeugung eines Timers bestimmen zu müssen. Um Automatische Timer zu verwenden muss man lediglich eine oder mehrere Methoden mit der @Schedule (javax.ejb.Schedule) Annotation versehen. Zusätzlich muss man noch den Ausführungszeitplan angeben. Den Rest erledigt der EJB-Container.

Der EJB-Container registriert die Bean beim Deployment beim Timer-Service, installiert alle benötigten Timer und ruft die Callback-Methoden beim Ablauf eines Timers auf. *(Oliver Ihns, S. 471/472)*

Die folgende Abbildung soll das Ganze verdeutlichen:

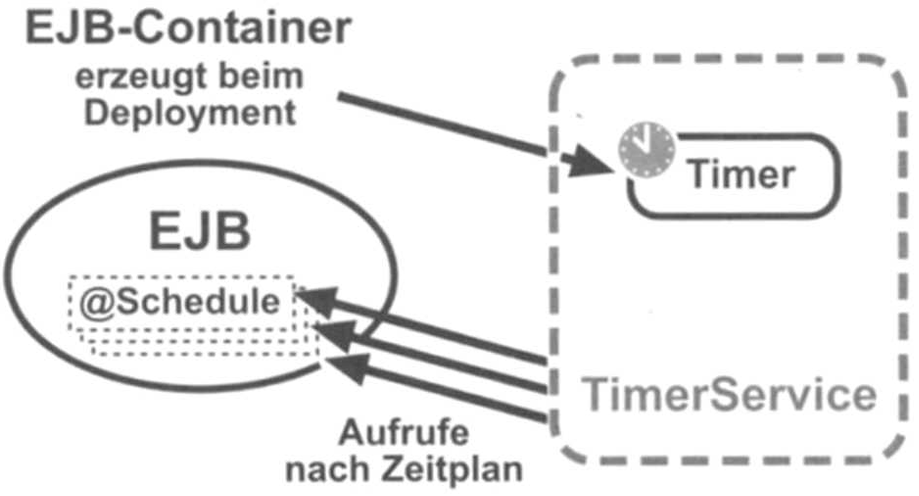

Abbildung 12: Funktionsweise automatischer Timer, entnommen aus *(Oliver Ihns, S.472)*

Die Regeln für die Callback Methoden sind die gleichen wie die, die in „4.1.2 Die TimeoutMethode“ beschrieben sind. Allerdings können hier beliebig viele Methoden mit der Annotation @Schedule markiert werden.

Automatische Timer sind grundsätzlich Calendar Schedule Based Timer, wobei der gewünschte Ausführungszeitplan über die @Schedule Annotation mitgegeben wird.

Auch automatische Timer sind per Default persistent. Dieses Verhalten kann aber über das optionale Attribut persistent=“false“ in der @Schedule Annotation geändert werden. *(Oliver Ihns, S. 472/473)*

```java
package de.tbartl.java.verteilte.anwendung.example.reports;

import java.util.Date;

@Stateless
public class ReportService implements Report{

    public String getTimeReport(){
        Date date = new Date();

        return "Date from ReportService: " + date.toString();
    }

    @Schedule(second="0,10,20,30,40,50", minute="*", hour ="*")
    public void displayReport(Timer timer){

        System.out.println(getTimeReport());
    }


}

```

Listing 15: Automatischer Timer der alle 10 Sekunden ausgeführt wird.

C:\glassfishv3\glassfish\domains\domain1\logs\server.log

```
INFO: Date from ReportService: Thu May 30 19:07:10 CEST 2013

INFO: Date from ReportService: Thu May 30 19:07:20 CEST 2013

INFO: Date from ReportService: Thu May 30 19:07:30 CEST 2013

INFO: Date from ReportService: Thu May 30 19:07:40 CEST 2013

INFO: Date from ReportService: Thu May 30 19:07:50 CEST 2013
```

Abbildung 13: Consolen Ausgabe des automatischen Timers von Listing 15.

4.3 Timer und Transaktionen

Werden Timer innerhalb von Transaktionen erzeugt, und es erfolgt ein Rollback, so ist das Timer-Objekt nach dem Rollback nicht mehr vorhanden. Falls ein Timer aber innerhalb einer Transaktion gelöscht wird und die Transaktion zurückgerollt wird, existiert der Timer nach wie vor. *(Oliver Ihns, S. 479)*

5 Asynchrone Methodenaufrufe

Bis jetzt wurde in dieser Seminararbeit immer nur synchrone Kommunikation behandelt. In vielen Enterprise Applikationen kann es aber sinnvoll, bzw. unabdingbar sein, asynchrone Kommunikation zu nutzen. Im nächsten Kapitel wollen wir uns deshalb mit asynchronen Methodenaufrufen beschäftigen.

Sinnvolle Einsatzgebiete währen z. B. das Anstoßen von Hintergrundprozessen, die keine weitere Interaktion erfordern, das Ausführen komplexerer Aufgaben, während der Benutzer mit anderen Teilen der Applikation weiterarbeiten kann.

Früher, also vor EJB 3.1, musste für jede Art von asynchrone Kommunikation, immer die Message Driven Bean Technologie eingesetzt werden. Das bedeutete aber oft zusätzlichen unnötigen Konfigurations- und Implementierungsaufwand. Mit EJB 3.1 können asynchrone Methodenaufrufe einfach innerhalb einer Session Bean neben normalen synchronen Methoden realisiert werden. Alles, was dazu nötig ist, ist die entsprechende Methode mit @Asynchronus (@javax.ejb.Asynchronus) zu kennzeichnen. Die @Asynchronus Annotation lässt sich auf Klassen sowie auf Methodenebene anwenden.

Grundsätzlich gibt es zwei verschiedene Arten von asynchronen Methodenaufrufen. Methodenaufrufe, die keinen Rückgabewert haben also „void“ sind, und Methodenaufrufe mit Rückgabewert. Letztere bekommen einen speziellen Rückgabetyp, ein sogenanntes Future (java.util.concurrent.Future<V>). Dabei ist das „V“ der eigentliche Rückgabetyp der Methode. *(Oliver Ihns, S.147/148), (Weil, S.249)*

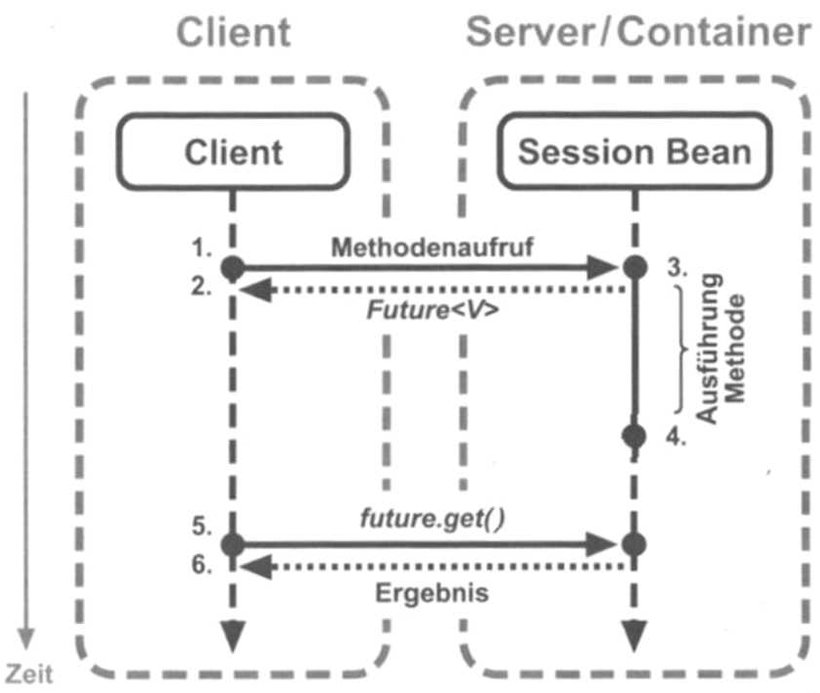

Abbildung 14: Ablauf eines asynchronen Session Bean Aufrufs mit Rückgabewert, entnommen aus (Oliver Ihns, S.148)

5.1 Ablauf eines asynchronen Methodenaufrufs

Für Methodenaufrufe, die keinen Rückgabewert haben, ändert sich im Vergleich zu einem synchronen Methodenaufruf nichts, außer dass die Methode mit der @Asynchronus Annotation versehen wird. Der Unterschied ist, dass der Client nicht bis zum Ende der Abarbeitung dieser Methode warten muss. Er kann also direkt mit der Programminteraktion fortfahren, insofern das Ergebnis der Methode dafür nicht relevant ist.

Bei asynchronen Methodenaufrufen mit Rückgabewert wird anstelle des normalen Rückgabewerts ein typisiertes Stellvertreterobjekt, ein sogenanntes Future, zurückgeliefert. Dieses Stellvertreterobjekt steht sofort beim Aufruf der asynchronen Methode zur Verfügung, auch wenn die Methode noch gar nicht mit Ihrer Verarbeitung fertig ist. Der Client kann zu einem späteren Zeitpunkt auf das Ergebnis dieser asynchronen Methode über das Stellvertreterobjekt, also das Future zugreifen, insofern das Ergebnis mittlerweile schon bereitsteht.

Wenn der Client zu einem späteren Zeitpunkt auf das Ergebnis der Methode zugreifen will, dann muss er sich erst mal vergewissern, dass das Ergebnis schon bereitsteht, bzw. schon im Future hinterlegt ist. Das kann er mit der „get()“ Methode realisieren. Sobald der Client, beim Future mit der get() Methode anfragt, ob das Ergebnis der Methode bereits vorhanden ist, wird das Ergebnis entweder sofort zurückgeliefert, oder falls es noch nicht bereitsteht, wird der Client solange blockiert, bis das Ergebnis zur Verfügung steht. Falls man vermeiden möchte, dass nach dem Aufruf der get() Methode der Client blockiert ist, kann man alternativ mit der Methode .isdone() überprüfen, ob die Berechnung bereits abgeschlossen ist, und ein Ergebnis im Future hinterlegt worden ist.

Der Client ist außerdem in der Lage, über das Stellvertreterobjekt die Berechnung abzubrechen, oder kann nachvollziehen, ob während der Berechnung Fehler aufgetreten sind. Dieses Verhalten kann man mit den Methoden .cancel() und .getCause() realisieren. *(Oliver Ihns, S.148), (Weil, S.249)*

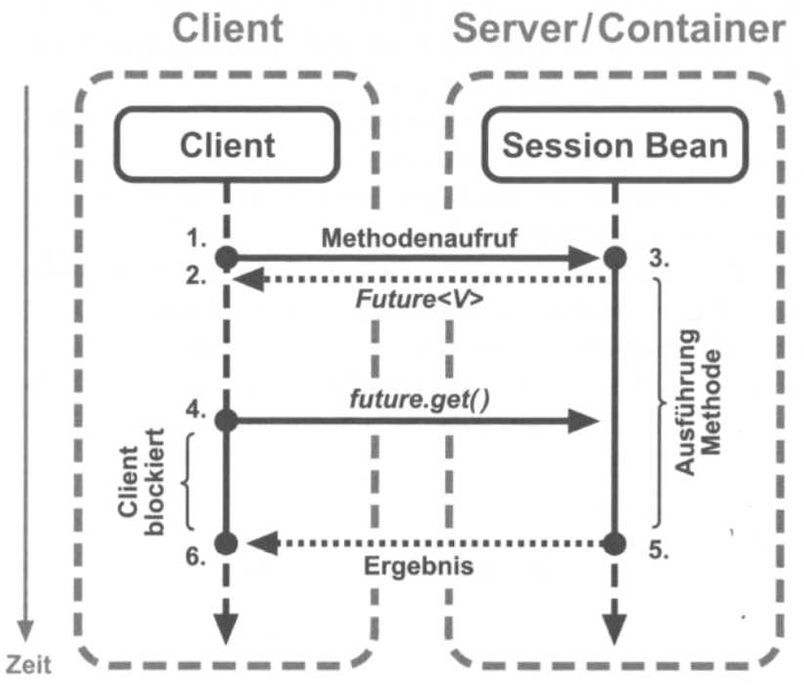

Abbildung 15: Aufruf der get()-Methode während andauernder Berechnung, entnommen aus *(Oliver Ihns, S.150)*

```java
package de.tbartl.java.verteilte.anwendung.example.waitAndWrite;

import java.util.concurrent.Future;

@Singleton
public class WaitAndWriteService implements WaitAndWrite{

    @Asynchronous
    public Future<String> waitAndWriteString(){
        // wait 10 sec.
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            System.out.println("Error in: WaitAndWriteService.waitAndWrite()");
        }

        return new AsyncResult<String>("Fertig");
    }

}
```

Listing 16: Asynchrone Methode mit Future<String> Rückgabewert

5.2 Asynchrone Methoden und Transaktionen

Für asynchrone Methoden einer Session Bean ist es nicht erlaubt, mit Bean-Managed Transaction Demarcation einen Transaktionskontext vom Client zu verwenden. Container Managed Transactions können allerdings wie gewohnt verwendet werden. *(Oliver Ihns, S.151), (Weil, S.249)*

6 Patterns:

Patterns, bzw. Entwurfsmuster, sind bereits erprobte und regelmäßig eingesetzte allgemeingültige Verfahren zur Lösung eines Entwurfsproblems in verschiedenen Programmiersprachen. Die Popularität von Entwurfsmustern entstand durch das in den 90ern erschienene Buch, „Design Patterns – Elements of Reusable Object Oriented Software“, von der sogenannten Gang of Four. Die Mitglieder der Gang of Four kurz GOF sind Erich Gamma, Richard Helm, Raplh Johnson und John Vlissides. Es gilt bis heute als eines der Standardwerke im Bereich der Softwaretechnik.

Die Java 2 Enterprise Edition hat sich schnell zu einer der bedeutendsten Plattformen unserer Zeit entwickelt. Sie war maßgeblich daran beteiligt, die Entwicklung der Applikationssysteme vom Großrechner weg, wieder zurück auf den Server zu bringen. Das Ziel war die Verteilung, Skalierbarkeit und Transaktionssicherheit so einfach wie möglich zu gestalten, bzw. diese Dienste einfach vom Applikation Server zur Verfügung stellen zu lassen. Entwickler konnten sich so auf die Modellierung der Geschäftslogik konzentrieren. Leider war die Enterprise Applikation Entwicklung in den früheren Java EE Versionen noch nicht so ausgereift wie heute. Gerade das Enterprise-Umfeld stellte enorme Anforderungen an das Softwaredesign, da sich ansonsten schnell Laufzeit- oder Wartungsprobleme ergaben. Gerade in der Anfangszeit der J2EE scheiterte eine Vielzahl an Projekten. Nach einiger Zeit ergaben sich aber verschiedene Entwurfsmuster, die diese Probleme umgehen, bzw. lösen konnten. Diese Entwurfsmuster wurden von der Firma Sun in sogenannten Blue Prints veröffentlicht bzw. empfohlen.

Durch die Einführung von Java EE 6 wurden viele dieser Probleme gelöst. Viele der Patterns wurden in Java EE 6 schon standardmäßig und transparent für den Entwickler implementiert. Deshalb gelten viele der alten Entwurfsmuster in Java EE 6 nicht mehr, oder müssen ggf. entsprechend abgeändert werden. Adam Bien drückt es folgendermaßen aus: „Wenn man in Java EE 6, Design Pattern nutzt, versucht man bereits gelöste Probleme der Java-Spezifikation zu umgehen“.

Im folgenden Kapitel sollen neue Entwurfsmuster für Java EE 6 vorgestellt werden. Diese Entwurfsmuster stammen aus dem 2012 veröffentlichten Buch „Real World Java EE Patterns Rethinking Best Practices - Adam Bien“. *(Bien), (Amazon)*

Im folgenden Abschnitt sollen die drei Patterns „Boundary“, „Control“, „Entity“ kurz vorgestellt werden.

6.1 Boundary

Die grundsätzliche Idee hinter dem Boundary Pattern ist, Schnittstellen von externen Komponenten zu vereinfachen. Die Veröffentlichung von feingranularen Methoden der Geschäftslogik, im Netzwerk, über Remote Interfaces, ist allein schon aus Performancegründen nicht sinnvoll. Außerdem ist es schwierig, feingranulare Methoden in einem Transaktionskontext auszuführen. Das Boundary Pattern koordiniert nun die entsprechenden Services und Objekte einer Komponente, erhöht dadurch die Granularität und bietet diese als externe Schnittstelle nach Außen an.

Ein Boundary ist in der Regel ein Stateless, in Ausnahmefällen auch ein Stateful Session Bean, ohne Interface Implementierung. Ein Remote Interface sollte immer nur dann explizit implementiert werden, wenn das Boundary außerhalb der JVM zugänglich gemacht werden soll.

Das Boundary repräsentiert die Geschäftslogik einer Komponente, es ist der erste Ansprechpartner eines externen Clients. Deshalb sollten die externen Schnittstellen also die nach außen veröffentlichten Methoden möglichst grob granular, bzw. so einfach und fachlich wie möglich, gewählt werden. *(Bien, S.57/58/59)*

Konventionen nach *(Bien, S.59/60)*:

- Das Boundary befindet sich in einer Komponente, das als Java Package mit einem fachlich spezifischen Namen realisiert ist.

- Die Realisierung des Boundary (das Geschäftsinterface und die Bean Implementierung) sollte immer in einem untergordneten Package „Boundary“ liegen.

- Protokoll spezifische Adapter, z.b. JAX-RS, JAX-WS, Hessian, und IIOP, liegen im selben Package wie das Boundary. Implementierungen die mehrere Klassen umfassen, sollten allerdings in ein untergeordnetes Subpackage verschoben werden.

- Alle injizierten Felder haben keinen speziellen Zugriffsmodifizierer. Diese Felder sind also packageweit sichtbar.

- Die Bean sollte nicht den Namen „Boundary“ enthalten. Durch die Namensgebung des Unterpackages „Boundary“ wurde die Verwendung des Boundary Patterns schon deutlich gemacht.

- Das Boundary startet immer eine neue Transaktion, wird also üblicherweise mit dem „REQUIRES_NEW“ Transaktions Atttribut deployt. Die „REQUIRES_NEW“ Option ist die beste Wahl für die Boundary Pattern Technologie.

```
@Stateless
public class BookOrdering{

    @PersistenceContext
    EntityManager em;

    @Inject
    Delivery delivery;

    public void order(Book book) {
        this.em.persist(book);
        delivery.deliver(book.getIsbn());
    }

    public void cancelOrder(String id) {
    }
}
```

Listing 17: Boundary Beispiel, entnommen aus (Bien, S.58)

6.2 Control

Ein Control ist das Resultat, das aus einem Boundary Refactoring hervorgeht. Wenn das Boundary durch stetige Funktionalitätserweiterung zu komplex wird, kapselt man man Funktionalität in einem Control. Ein Control ist immer ein POJO. Das Boundary veröffentlicht nur die wichtigsten grobgranularen Funktionen und koordiniert die jeweiligen Controls. Da das Boundary immer eine neue Transaktion beginnt, und die verschiedenen Controls koordiniert, macht das jeweilige Control immer diejenige Transaktion mit, die vom Boundary initialisiert wurde.

Die Implementierung eines Controls ist nicht zwingend notwendig, sondern nur erforderlich, wenn das jeweilige Boundary zu komplex wird. In den meisten einfachen Fällen verschmelzen Boundary und Control. Die Funktionalität des Controls geht dann in das Boundary über. *(Bien, S.83/84/85)*

Konventionen nach *(Bien, S.85/86)*:

- Ein Control ist eine Klasse ohne irgendwelche Annotationen, es sei denn, es wird EJB 3.1 Funktionalität benötigt wie @Schedule oder @Asynchronus.

- Ein Control hat kein Interface. Seine Public Methoden ersetzen das Interface.

- Ein Control liegt in einem Java Package mit einem fachlich spezifischen Namen.

- Ein Control (Control realisiert als einfaches POJO) liegt in einem untergordneten Package, das den Namen „Control“ trägt.

- Ein Control Name sollte die Geschäftslogik konkret wiedergeben. Es sollten keine technischen oder infrastrukturelle Silben wie „Impl“ „control“ oder weiteres verwendet werden.

- Nutzlose und nichtsaussagende Silben, wie „Mgmt“ oder „Service“, sollten vermieden werden.

- Ein Control ist immer an der Transaktion das vom Boundary initialisiert wurde, beteiligt/gebunden.

6.3 Entity

Das Entity hat mit die wichtigste Rolle in einer „Domain-Driven Architecture“. Es repräsentiert das Schlüsselkonzept der fachlichen Umsetzung. Das Entity ist dafür verantwortlich, die Anforderungen an die Geschäftslogik so klar wie möglich zu beschreiben. Grundsätzlich kann der Zustand eines Entitys mithilfe des EntityManagers persistiert werden. Ein Entity ist eher ein passives Element, das von einem Control oder Boundary erzeugt und verwaltet werden muss. *(Bien, S.91/92/93)*

Konventionen nach *(Bien, S.98):*

- Entities sind JPA (Java Persistence API) Entities, der Schwerpunkt liegt allerdings auf der Realiserung von Geschäftslogik.

- Entities liegen in einer Komponente die als Java Package mit einem fachlich spezifischen Namen realisiert ist.

- Das Entity an sich liegt in einem untergeordnetem Package, das mit dem Namen „entity“ versehen ist.

- Getter und Setter müssen nicht zwingend, sondern je nach Anforderung implementiert werden.

- Es sollte methodenspezifisches Verhalten implementiert werden, das den Zustand der Geschäftsobjekte verändern kann.

- Es ist nicht notwendig, die Silbe „Entity“ im eigentlichen Klassennamen zu verwenden, da mit der Namensbezeichnung des untergeordneten Package die Verwendung des Patterns schon deutlich gemacht wurde.

7 Fazit

Mit Java EE 6, bzw. EJB 3.1 und der Einführung von Context and Dependency Injection, hat die Java Enterprise Edition endlich die Leichtgewichtigkeit erreicht, die schon lange angestrebt war. Mittlerweile verbreiten sich Java EE überall. Das liegt vor allem daran, dass die Entwicklung weg, von schwerfällig zu konfigurierbaren Komponenten, hin zu leichtgewichtigen, mit Annotationen versehenen Komponenten den Java Entwicklern sehr entgegenkommt. Die in Java EE 6 neu eingeführten Singleton Session Beans und Context and Dependency Injection sowie Asynchrone Methodenaufrufe beweisen, dass Oracle sich sehr bemüht, den entsprechenden Wünschen und Anforderungen der Community gerecht zu werden.

Grundsätzlich ist es möglich mit Java EE 6 und der EJB 3.1 Technologie schnell und einfach komplexe Unternehmensapplikationen zu programmieren. Dabei wird man in fast allen Bereichen vom Applikationsserver und EJB-Container unterstützt. Selbstverständlich hat man immer die Freiheit alles wie gewohnt selbst zu programmieren. Wer sich aber einmal an die vom EJB-Container bereitgestellten Dienste und die Unterstützung gewöhnt, wird diese, wann immer möglich mit Freude einsetzen.

Wer allerdings noch nicht so viel Erfahrungen mit Java EE 6 gesammelt hat, sollte sich nicht gleich dazu hinreißen lassen, diese Technologie im nächsten Projekt einzusetzen. Die Einarbeitungszeit sollte auf keinen Fall unterschätzt werden. Gerade die verschiedenen EJB-Konzepte und Context and Dependency Injection sind am Anfang nicht leicht zu verstehen. Für Entwickler, die im Java Umfeld tätig sind, würde sich eine gewissenhafte Einarbeitung in Java EE allerdings in jedem Fall lohnen.

Anhang A Listing

Listing 1: Stateless Session Bean 13
Listing 2: Stateful Session Bean 16
Listing 3: Singleton Session Bean die direkt beim Applikationsstart erzeugt wird. ... 19
Listing 4: „ActualDateService“ – POJO (Ganz normales Java Object) 26
Listing 5: „ActualDate“ Interface 26
Listing 6: CDI Bean/Managed Bead, ohne CDI. 27
Listing 7: CDI Bean/Managed Bead, mit CDI..... 28
Listing 8: Bean Typen einer Managed Bean entnommen aus *(Oliver Ihns, S.503), (Weil, S.24)*..... 29
Listing 9: Bean Typen einer Stateless Session Bean entnommen aus (Oliver Ihns, S.503) 29
Listing 10: Typed Annotation entnommen aus *(Oliver Ihns, S. 504)* 30
Listing 11: Definition des Qualifier Invoice 30
Listing 12: Bean mit Qualifier Invoice spezifiziert entnommen aus *(Oliver Ihns, S. 504), (Weil, 26)* 31
Listing 13: Nutzen des Qualifiers Invoice am Injektionspunkt entnommen aus *(Oliver Ihns, S. 504)* 31
Listing 14: Erzeugung eines Timers nach Zeitplan entnommen aus *(Oliver Ihns, S.468)* 37
Listing 15: Automatischer Timer der alle 10 Sekunden ausgeführt wird. 39
Listing 16: Asynchrone Methode mit Future<String> Rückgabewert 43
Listing 17: Boundary Beispiel, entnommen aus (Bien, S.58) 47

Anhang B Literaturverzeichnis

(Bien) Bien, Adam. Java EE Patterns - Rethinking Best Practices. Real World, 2012.

(Oliver Ihns) Oliver Ihns, Stefan M.Heldt, Holger Koschek, Joachim Ehm, Casten Sahling, Roman Schlömmer. EJB 3.1 proffessional - Grundlagen- und Expertenwissen zu Enterprise JavaBeans 3.1. Heidelberg: dpunkt.verlag, 2011.

(Weil) Weil, Dirk. Java EE 6 - Enterprise-Anwendungsentwicklung leicht gemacht. Frankfurt am Main: Software & Support Media GmbH, 2012.

(Werner Eberling) Werner Eberling, Jan Lessner. Enterprise Java Beans 3 - Das EJB 3 Praxisbuch. München: Hanser, 2007.

(Amazon) Amazon. http://www.amazon.de/J2EE-Patterns-Entwurfsmuster-f%C3%BCr-J2EE/dp/382731903X, aufgerufen am 12.05.2013.

(Java Community Process)

Java Community Process. http://jcp.org/en/jsr/detail?id=299, aufgerufen am 09.05.2013

Java Community Process. http://jcp.org/aboutJava/communityprocess/final/jsr318/, aufgerufen am 06.05.2013

(tversu) http://edc.tversu.ru/elib/inf/0052/0201914662_ch04lev1sec2.html, aufgerufen am 19.05.2013

(Wikipedia) Wikipedia. https://de.wikipedia.org/wiki/Dependency_Injection, aufgerufen am 13. 05 2013