

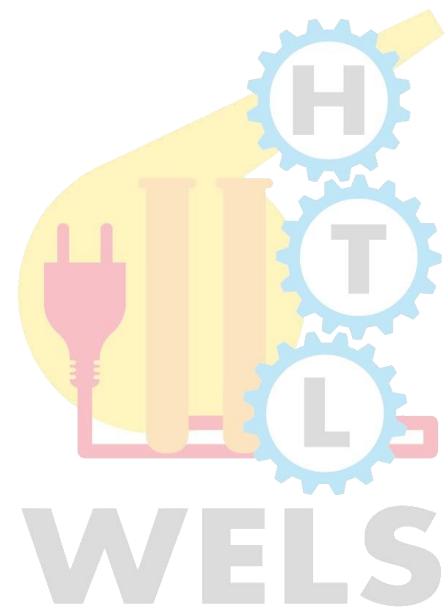
Datenbanksysteme 2

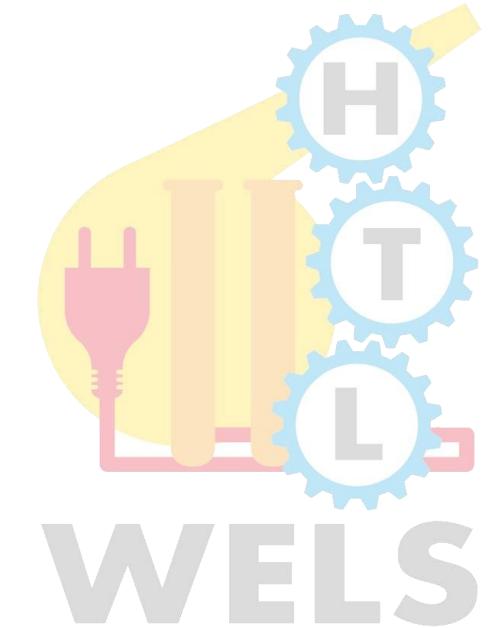
Java Persistence API

- Java Platform
- JEE
- Persistence
- JPA 2.1

Inhalt

- Java & Java Plattform
- Java Enterprise Edition (JEE 8.0)
 - Historie
 - Infrastruktur
 - Application Server
 - API's
 - Unterschiedlichen JEE Container
 - Packaging
- Servlets
- JSP / JSF
- EJB (JPA)
 - Exkurs: O/R-Mapping
 - Enterprise JAVA Beans
 - JPA
- Tomcat (Konfiguration)
- Glassfish (Konfiguration)





Java Enterprise Edition (JEE 8.0)



Java Enterprise Edition



- Die Java Platform, Enterprise Edition, abgekürzt **Java EE** oder früher **J2EE**, ist die **Spezifikation einer Softwarearchitektur** für die **transaktionsbasierte** Ausführung von in Java programmierten Anwendungen und insbesondere Web-Anwendungen.
- Salopp: „**JEE ist Java auf Serverseite für Unternehmensanwendungen.**“
- JEE basiert auf Java Platform, Standard Edition (Java SE)
- Sie ist eine der großen Plattformen, die um den **Middleware-Markt** kämpfen. Größter Konkurrent ist dabei die .NET-Plattform von Microsoft.
- In der Spezifikation werden **Softwarekomponenten** und **Dienste** definiert, die primär in der Programmiersprache **Java** erstellt werden. Die **Spezifikation** dient dazu, einen **allgemein akzeptierten Rahmen zur Verfügung zu stellen**, um auf dessen Basis aus modularen **Komponenten verteilte, mehrschichtige Anwendungen** entwickeln zu können.





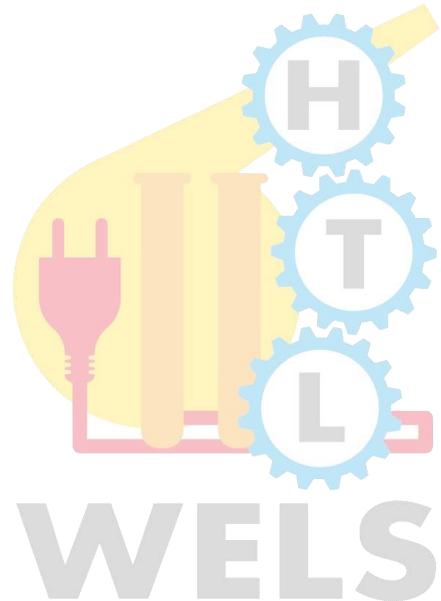
Certified for
IBM WebSphere.
software

Java Enterprise Edition

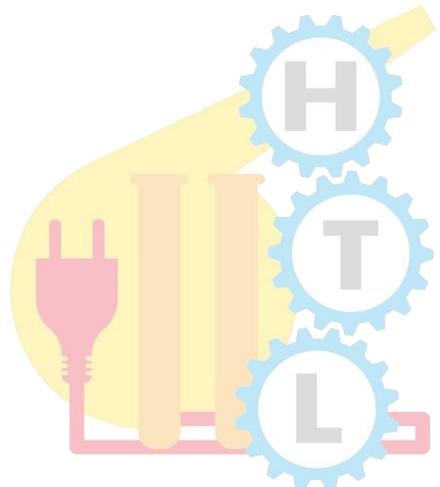
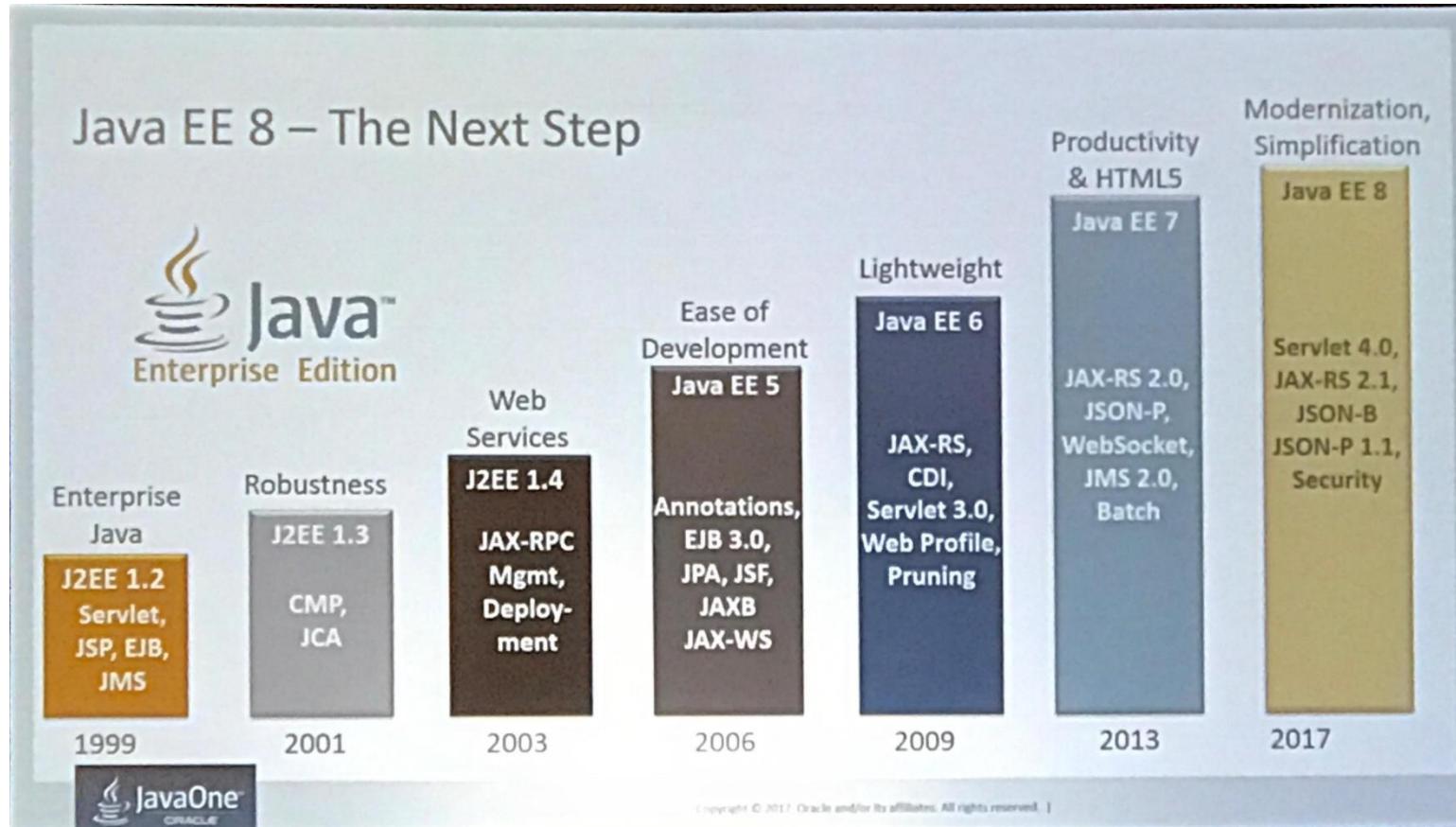


APACHE
GERONIMO

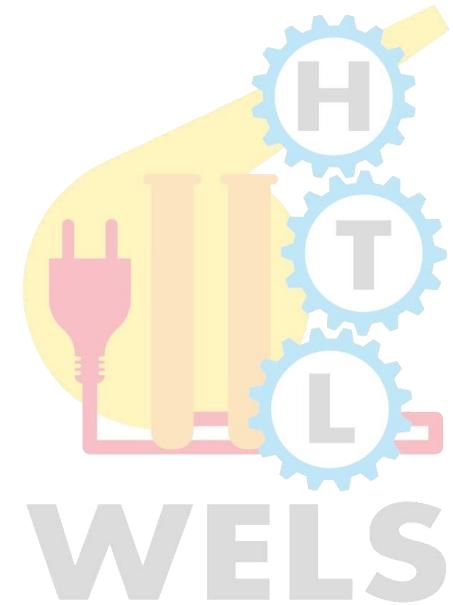
- Klar definierte **Schnittstellen** zwischen den **Komponenten** und **Containern** sollen dafür sorgen, dass Softwarekomponenten unterschiedlicher Hersteller **interoperabel** sind, wenn sie sich an die Spezifikation halten, und dass die verteilte Anwendung gut **skalierbar** ist.
- **JEE ist kein Produkt oder fertiges System**
- Bestandteile der Spezifikation werden innerhalb des **Java Community Process** von diversen Unternehmen erarbeitet und schließlich der Öffentlichkeit in Form eines Dokuments und einer Referenzimplementierung zur Verfügung gestellt.
- Die aktuelle Version JEE-Spezifikation ist die Version 6.0
- <http://www.oracle.com/technetwork/java/javaee/tech/index.html>



History J2EE / Java EE

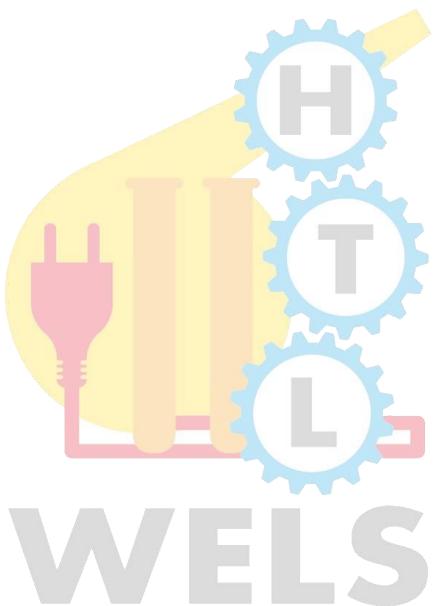


JEE - Hauptthemen



J2EE/JEE Releases

Version	Ausführlicher Name	Veröffentlichungs-datum	Release-Status
1.0	Java 2 Platform Enterprise Edition, v 1.0	Dezember 1999	Final Release
1.2	Java 2 Platform Enterprise Edition, v 1.2	Jänner 2000	Final Release
1.2.1	Java 2 Platform Enterprise Edition, v 1.2.1	23. Mai 2000	Final Release
1.3	Java 2 Platform Enterprise Edition, v 1.3	24. September 2001	Final Release
1.4	Java 2 Platform Enterprise Edition, v 1.4	24. November 2003	Final Release
5.0	Java Platform, Enterprise Edition, v 5.0	11. Mai 2006	Final Release
6.0	Java Platform, Enterprise Edition, v 6.0	10. Dezember 2009	Final Release
7.0	Java Platform, Enterprise Edition, v 7.0	12. Mai 2013	Final Release
8.0	Java Platform, Enterprise Edition v 8.0 SJR366	18. September 2017	Final Release



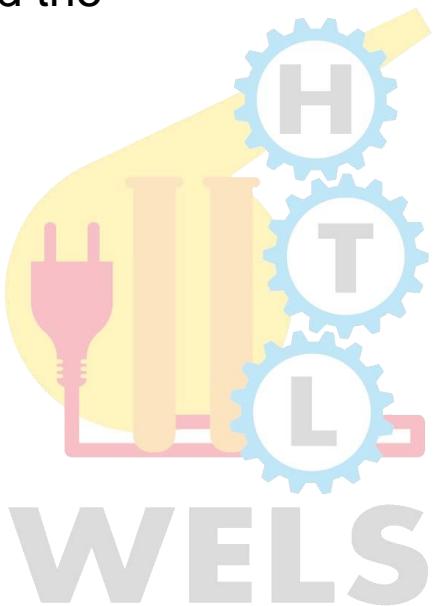
JEE-Infrastruktur

- Java-EE-Komponenten erfordern als **Laufzeitumgebung** eine spezielle Infrastruktur, einen sogenannten **Java EE „Application Server“**.
- Dieser Server stellt technische Funktionalitäten zur Verfügung
 - **Sicherheit (Security)**,
 - **Transaktionsmanagement**,
 - **Namens- und Verzeichnisdienste**,
 - **Kommunikation zwischen Java-EE-Komponenten**,
 - **Persistenzdienste zum langfristigen Speichern von Daten**,
 - **Management der Komponenten über den gesamten Lebenszyklus (inklusive Instanziierung)**,
 - **Unterstützung für die Installation (Deployment)**
- Des Weiteren kapselt der Server den Zugriff auf die Ressourcen des zugrundeliegenden Betriebssystems (Dateisystem, Netzwerk, ...).



Application-Server (1)

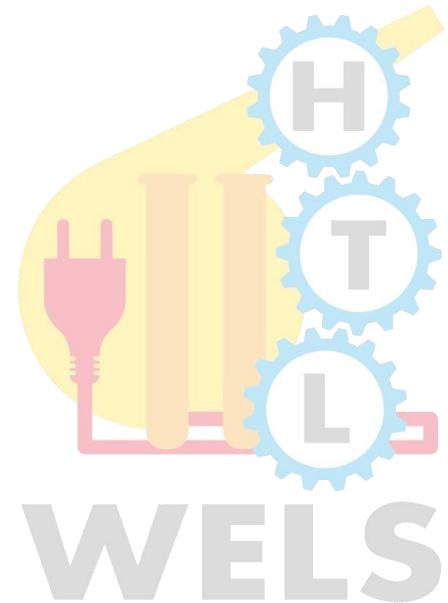
- Implementierungen der JEE-Spezifikation heißen **Application-Server** (Anwendungsserver, JEE-Server)
- Bietet eine Laufzeitumgebung für den Server-Teil einer Client-Server-Anwendung
- Hersteller können sich von SUN (Oracle) ihre Konformität zur JEE-Spezifikation bestätigen lassen
 - „The J2EE Compatibility Test Suite (CTS) ist available for the J2EE platform. The J2EE CTS contains over 5.000 test for J2EE 1.4 and will contain more for later versions. This test suite test compatibility by performing specific application functions and checking results“
 - „Look for the J2EE brand which signifies that the specific branded product has passed the Compatibility Test Suite (CTS) and is compatible.“



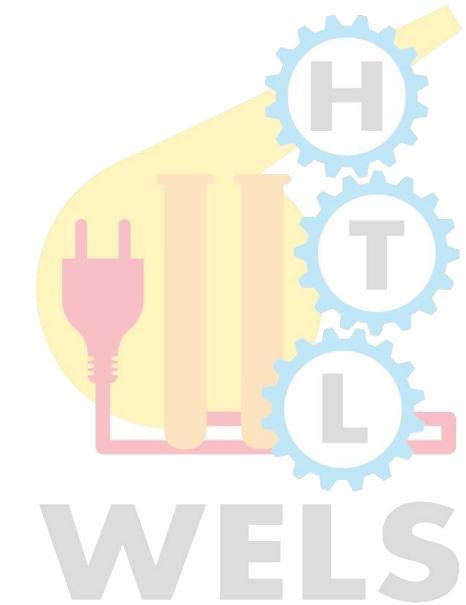
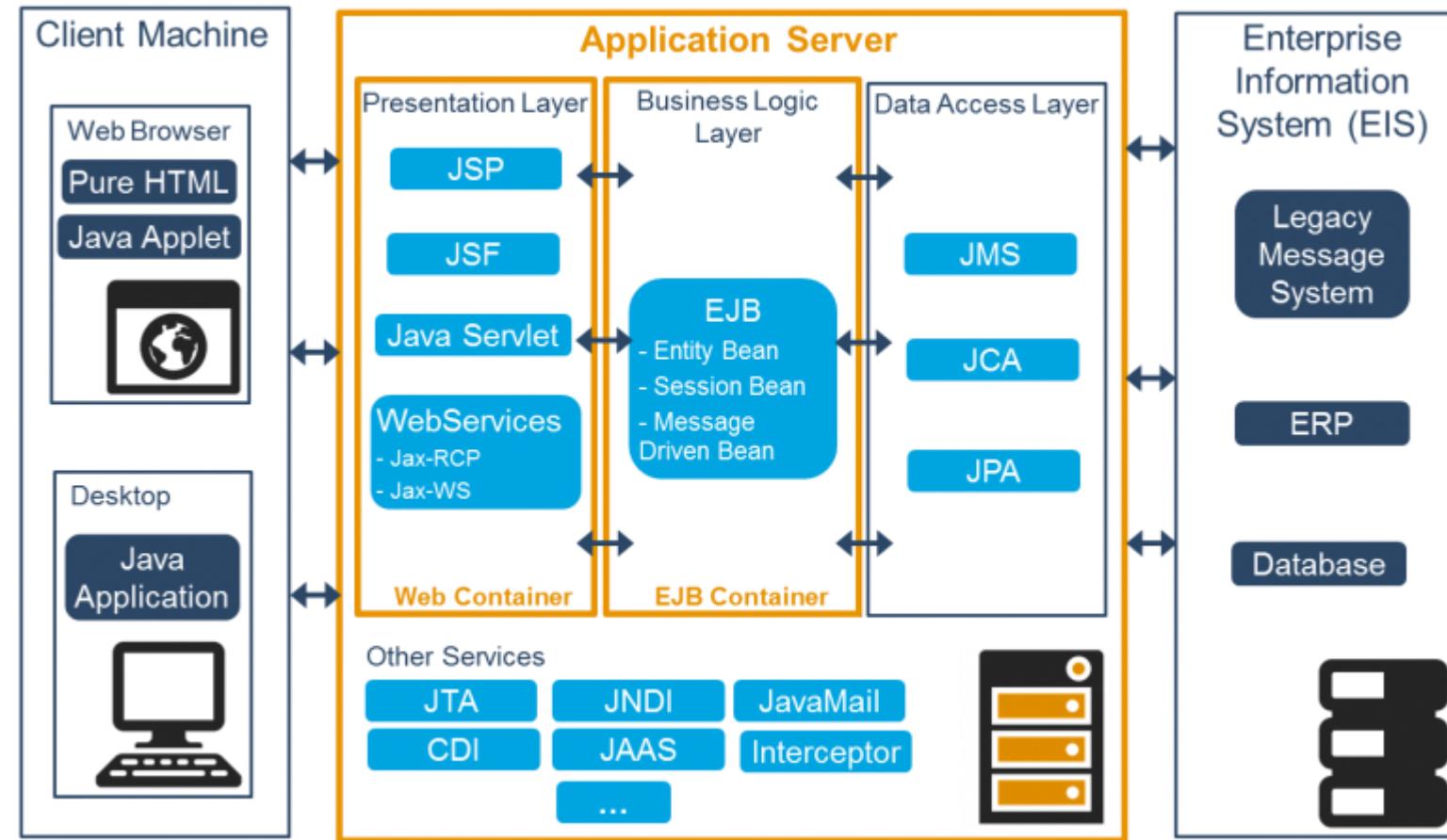
Application-Server (2)

- Die wichtigsten Aufgaben eines Application-Servers:

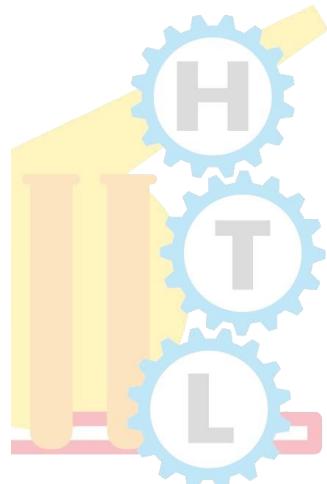
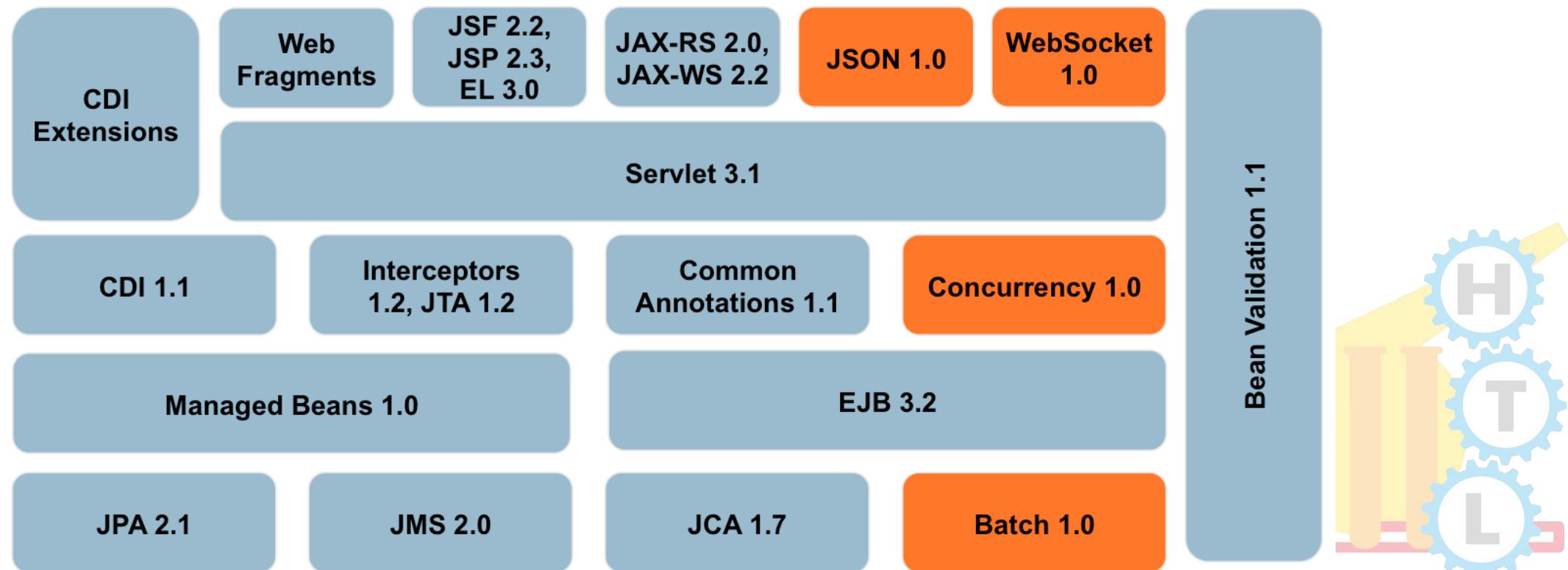
- Thread- und Prozess-Management
- Unterstützung von Clustering (Verbindung mehrerer Instanzen eines JEE-Servers auf einem oder mehreren Rechnern)
- Gleichmäßige Verteilung von Anfragen an die verschiedenen JEE-Server-Instanzen
- Verwaltung der Systemressourcen (CPU, Speicher)
- Verwaltung und Überprüfung von Sicherheitsanforderungen
- Integration von Middleware-Diensten (Messaging-, Namens- und Verzeichnisdienst)



JEE-Architektur



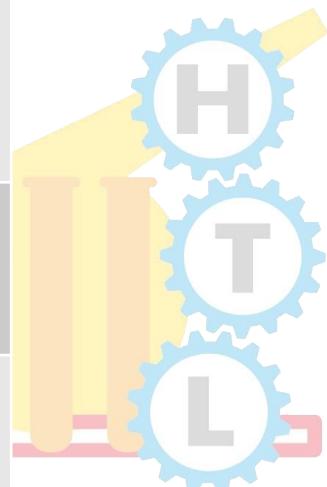
JEE-API's - Übersicht



WELS

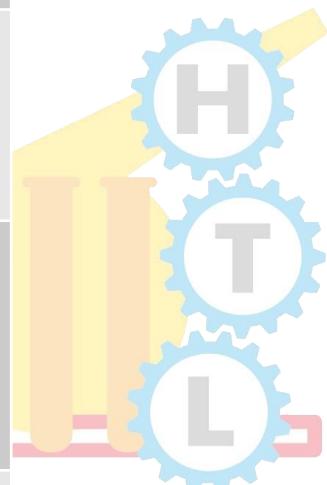
JEE-API's #1

Abkürzung	Name	Beschreibung	1.4	5.0	6.0	7.0
EJB	Enterprise Java Beans	<p>beinhalten die Geschäftslogik einer Enterprise Anwendung oder gestatten Zugriff auf persistente Daten. Die Beans laufen in einem EJB-Container ab. Es gibt drei unterschiedliche Typen von EJBs:</p> <ul style="list-style-type: none"> • Session-Beans, sowohl zustandsbehaftet als auch zustandslos, implementieren die Geschäftslogik und sind meistens vom Client zugreifbar • Message-Driven-Beans, kurz MDB, für die Verarbeitung von JMS-Nachrichten, wurden in Version 2.1 neu eingeführt • Entity-Beans für die Abbildung von persistenten Datenobjekten 	2.0	3.0	3.1	3.2
JSS	Java Servlet API	Im Allgemeinen erlaubt die Java-Servlet-API die Erweiterung von Servern, deren Protokoll auf Anfragen und Antworten basiert. Primär werden Servlets im Zusammenhang mit dem Hypertext Transfer Protocol (HTTP) verwendet, wo sie in einem Web-Container leben und Anfragen von Webbrowsern beantworten.	2.4	2.5	3.0	3.1
JSP	Java Server Pages	sind Textdokumente, die zum einen aus statischem Text und zum anderen aus dynamischen Textelementen – den JSP-Elementen – bestehen. Die JSP-Seiten werden transparent vom Web-Container in ein Servlet umgewandelt.	2.0	2.1	2.2	2.3
JSF	Java Server Faces	Mit Hilfe von JSF kann der Entwickler auf einfache Art und Weise Komponenten für Benutzerschnittstellen in Webseiten einbinden und die Navigation definieren.	X	1.2	2.0	2.2
JPA	Java Persistence API	stellt eine einheitliche und datenbankunabhängige Schnittstelle für Object-Relational-Mapping und das Arbeiten mit Entitäten bereit.	X	1.0	2.0	2.1



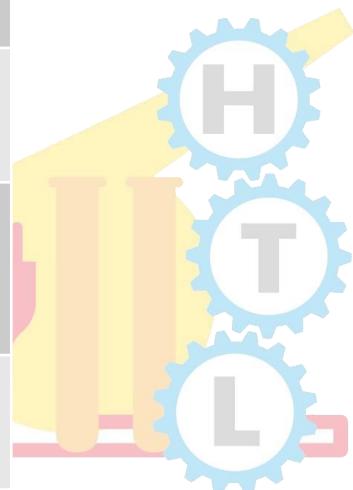
JEE-API's #2

Abkürzung	Name	Beschreibung	1.4	5.0	6.0	7.0
WS	Webservices	definieren Schnittstellen zu EJBs, die mit einem Uniform Resource Identifier (URI) eindeutig identifizierbar sind und deren Schnittstellen als XML-Artefakte definiert, beschrieben und gefunden werden können.	1.0	1.2	1.3	1.4
JNDI	Java Naming and Directory Interface	ist eine gemeinsame Schnittstelle mit der alle Java-Klassen auf Namens- und Verzeichnisdienste zugreifen können. Über JNDI wird insbesondere der Zugriff auf Java-EE-Komponenten sichergestellt.	1.2	1.2	1.2 SE	1.2 SE
JMS	Java Messaging Service	ist eine API für die asynchrone Nachrichtenverarbeitung.	1.1	1.1	1.1	2.0
JTA	Java Transaction API	erlaubt der Anwendung die Steuerung der Transaktionsverwaltung. JTA ist die Java-Schnittstelle zu Transaktionsmonitoren . Standardmäßig wird diese Schnittstelle implementiert vom Java Transaction Service (JTS), welcher eine Schnittstelle zu CORBA Object Transaction Service (OTS) bietet..	1.0	1.1	1.1	1.2
JAAS	Java Authentication and Authorization Service	Der "Java Authentication and Authorization Service" (JAAS) ist eine Java-API, die es ermöglicht, Dienste zur Authentifikation und Zugriffsrechte in Java-Programmen bereitzustellen. JAAS implementiert ein Standard-"Pluggable Authentication Module" (PAM) und unterstützt durch dieses Modul eine einfache Authentifizierung und benutzerbasierte Autorisierung .	1.0	1.0	1.0	1.0
JavaMail	JavaMail	erlaubt den Zugriff auf Mail Services, wie z.B. SMTP, POP3, IMAP oder NNTP.	1.2	1.4	1.4	1.5



JEE-API's #3

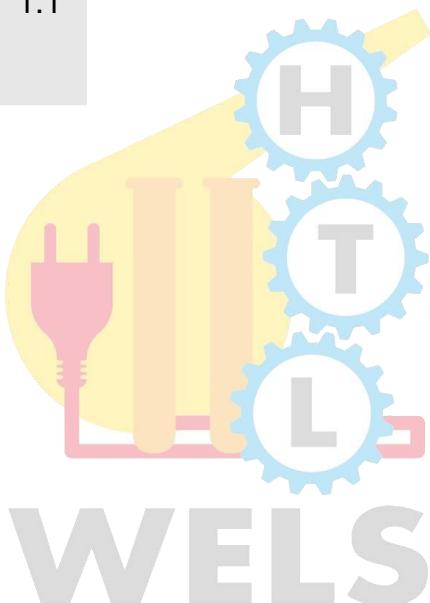
Abkürzung	Name	Beschreibung	1.4	5.0	6.0	7.0
JAXB	Java Architecture for XML Binding	ermöglicht es, ein XML-Schema direkt an Java-Klassen zu binden. Wurde offiziell erst seit Java EE Version 1.5 gefordert, wird jedoch evtl. schon vorher unterstützt.	X	2.0	2.2	2.2
JAXP	Java API for XML Processing	hilft dem Entwickler bei der Bearbeitung von XML-Dokumenten .	1.2	1.3	1.4	1.4
JAX-RPC	Java API for XML-Based Remote Procedure Calls	ermöglicht den entfernten Zugriff auf RPC-Dienste .	1.0	1.1	1.1	1.1
JAXR	Java API for XML Registries	dient dazu, einen transparenten Zugriff auf so genannte Business-Registries, wie beispielsweise ebXML oder ein UDDI basiertes Verzeichnis sicherzustellen.	1.0	1.0	1.0	1.0
JACC	Java Authorization Contract for Containers	dient dazu, andere Systeme transparent zu integrieren (Stichwort: EAI).	1.0	1.0	1.4	1.5
JAF	JavaBeans Activation Framework	bietet die Möglichkeit, verschiedene Daten anhand des MIME-Headers zu erkennen.	1.0	1.1	1.1	1.1



WELS

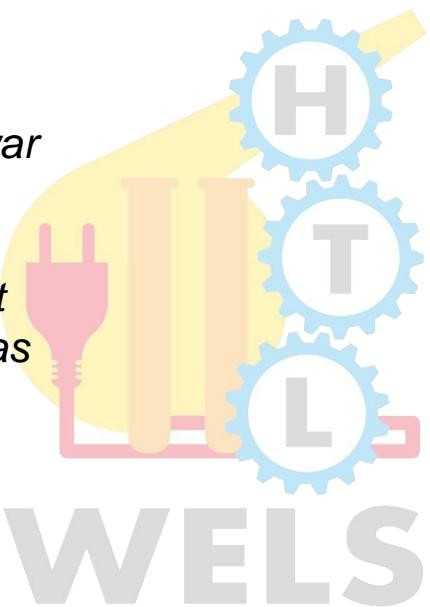
JEE-API's #4

Abkürzung	Name	Beschreibung	1.4	5.0	6.0	
JAX-WS	Java API for XML Web Services	hilft bei der Erstellung von Web Services und zugehörigen Clients , die über XML kommunizieren, z. B. über SOAP .	X	2.0	2.2	2.2
StAX	Streaming API for XML	Cursor-basierte XML-Verarbeitung in Ergänzung der DOM - und SAX-Parser .	X	1.0	1.0	1.0
JSTL	Java Server Pages Standard Tag Library	Sammlung von JSP-Tags für die Strukturierung, XML, SQL, Internationalisierung usw.	X	1.2	1.2	1.2
CDI	Context and Dependency Injection	CDI ist eine Technik um Felder nach dem Inversion of Control-Prinzip zu setzen. Es verbindet JSF mit EJB..	X	X	1.0	1.1

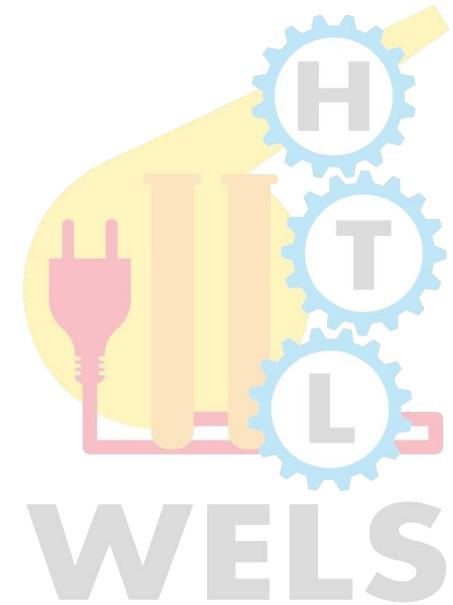
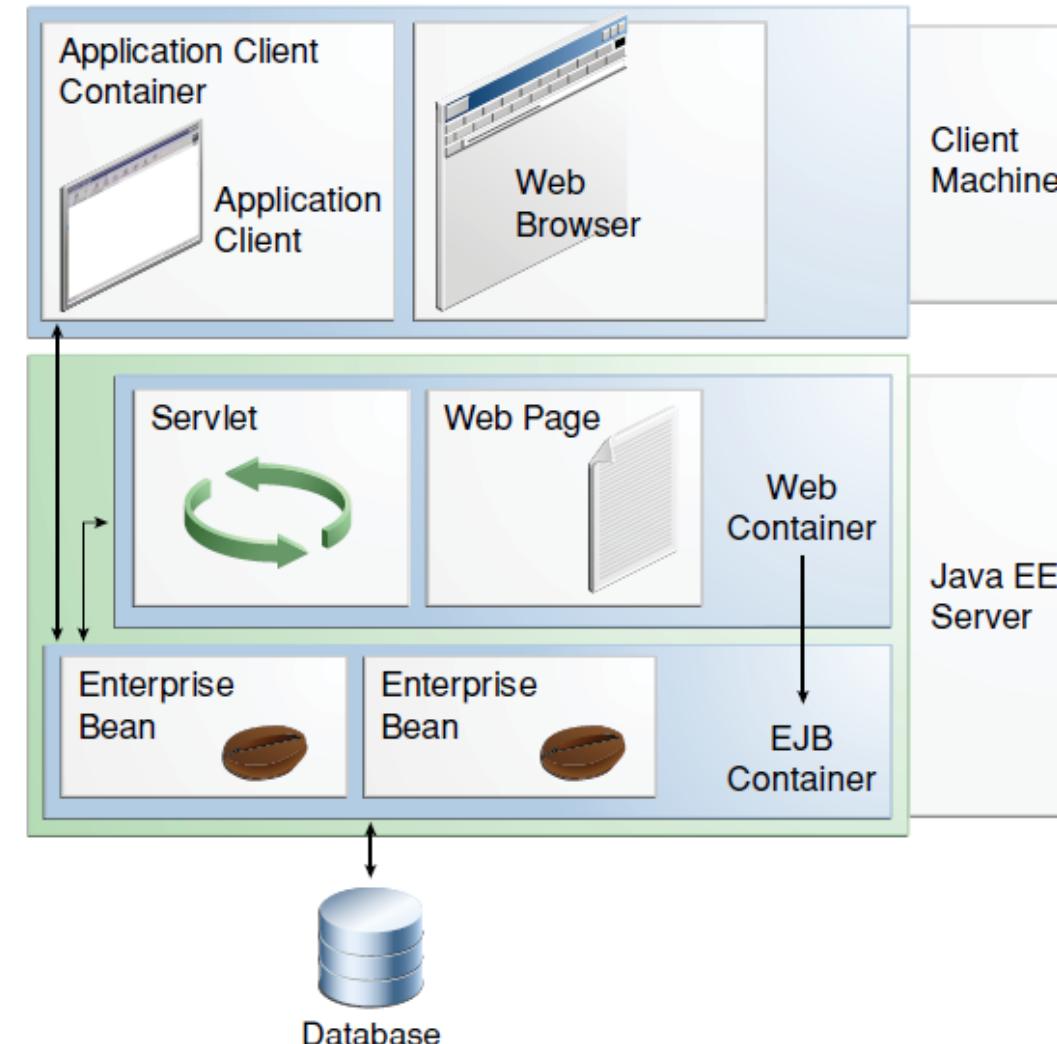


Systeme eines Java-EE-Servers

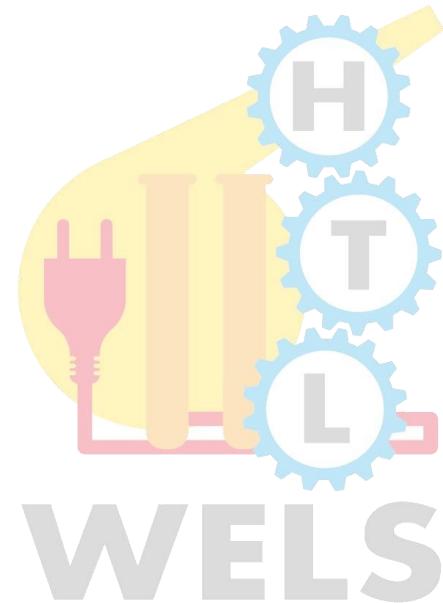
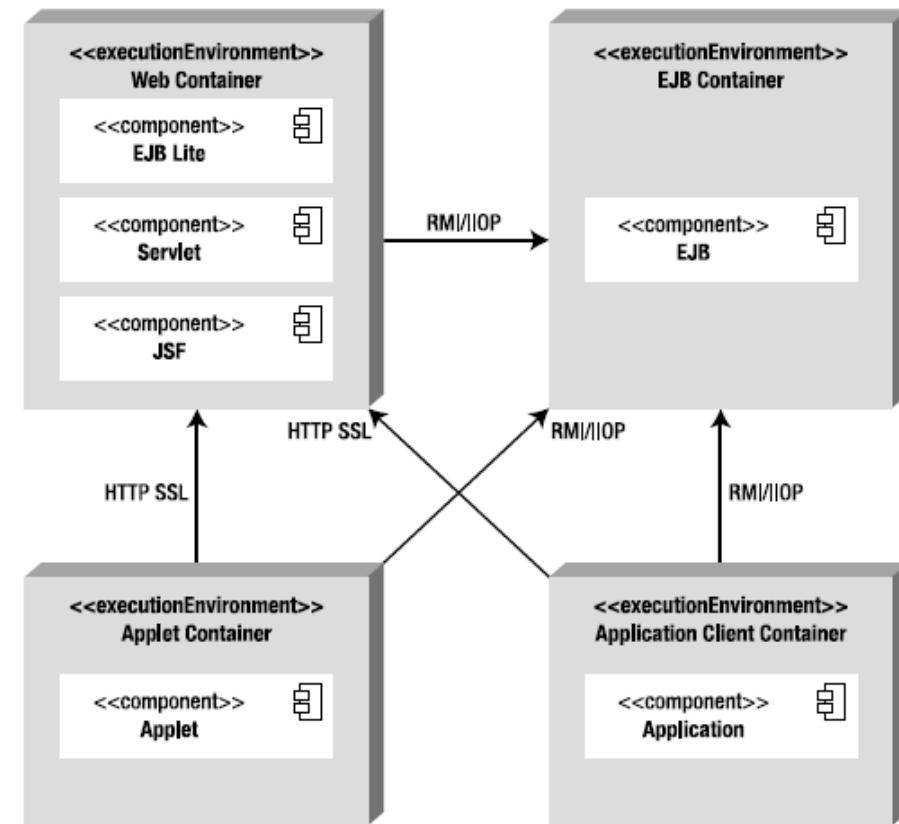
- Ein Java-EE-Server (**Application Server**) wird in diverse **logische Systeme** unterteilt. Diese werden **Container** genannt.
- Die aktuelle Spezifikation erfordert die folgenden Container:
 - einen **EJB-Container** als Laufzeitumgebung für **Enterprise Java Beans**
 - einen **Web-Container** als Laufzeitumgebung für **Servlets, Java Server Pages (JSP), Java Server Faces (JSF)**
 - einen **JMS-Provider** als Verwaltungssystem für Nachrichtenwarteschlangen (MOM).
 - einen **JCA-Container** als Laufzeitumgebung für **JCA Connectoren**. Dieser ist zwar nicht explizit definiert, faktisch jedoch muss jeder Application-Server-Hersteller diesen implementieren. Denn im Enterprise Java Beans (EJB) sowie im Web-Container sind Restriktionen definiert, welche für die JCA-Laufzeitumgebung nicht gelten. Dabei handelt es sich beispielsweise um das Starten von **Threads** oder das **Lesen und Schreiben in Dateien** etc.



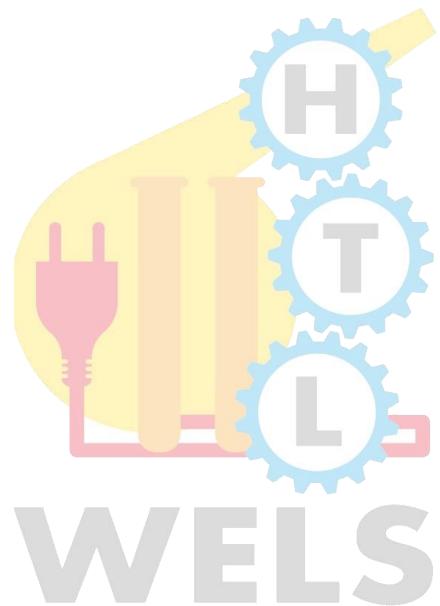
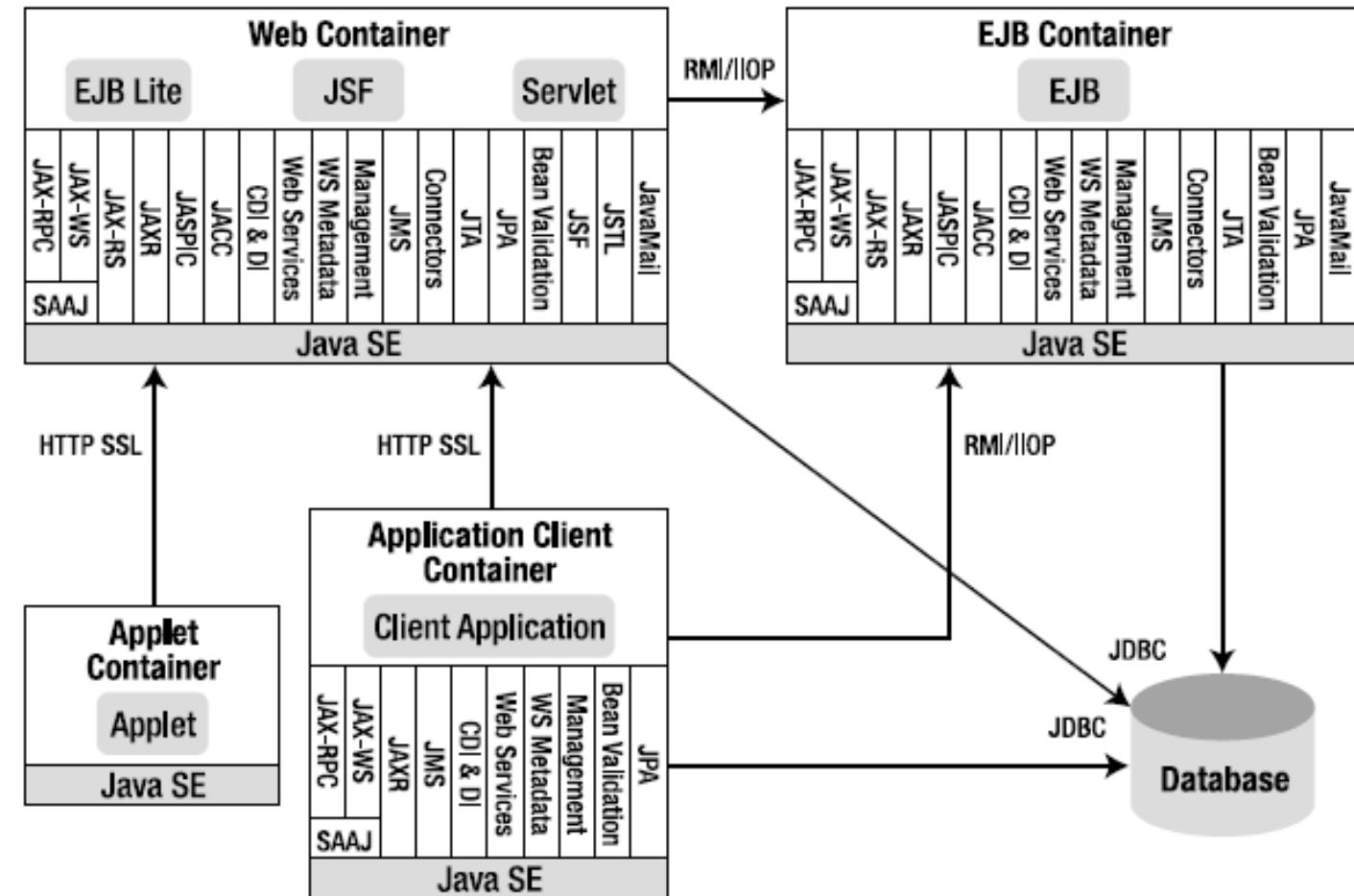
JEE-Container - Übersicht



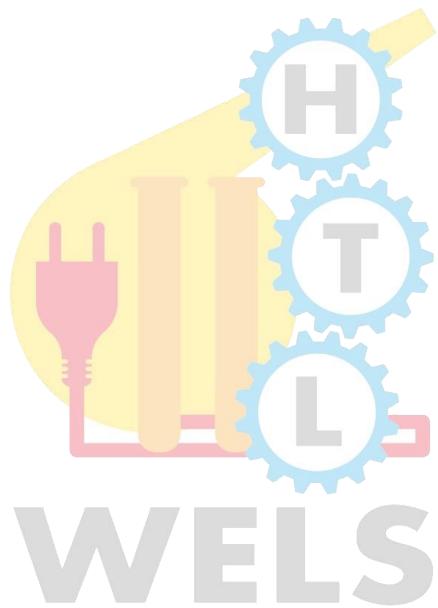
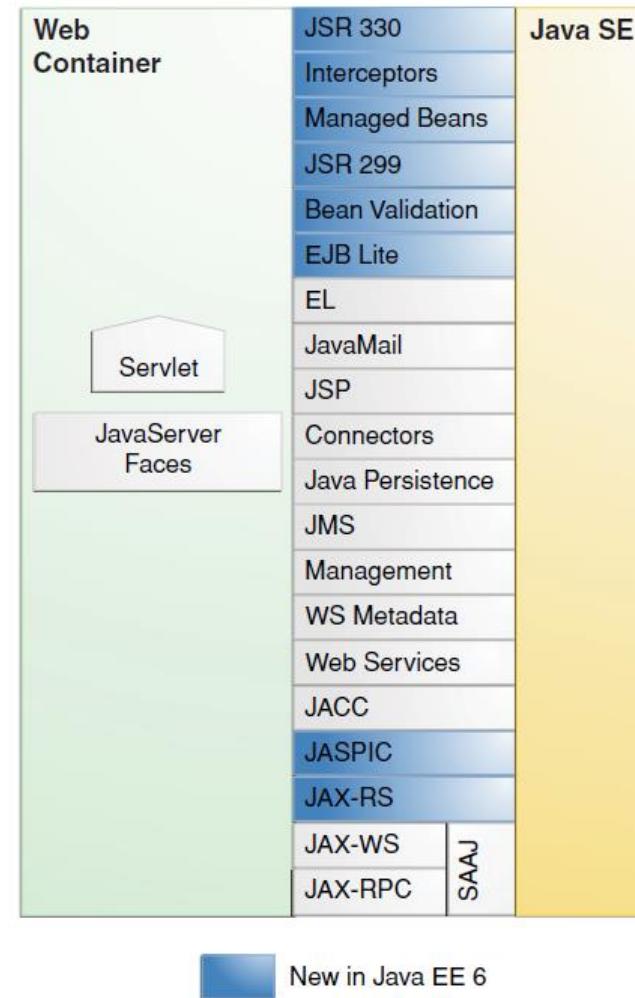
Container - Inhalte



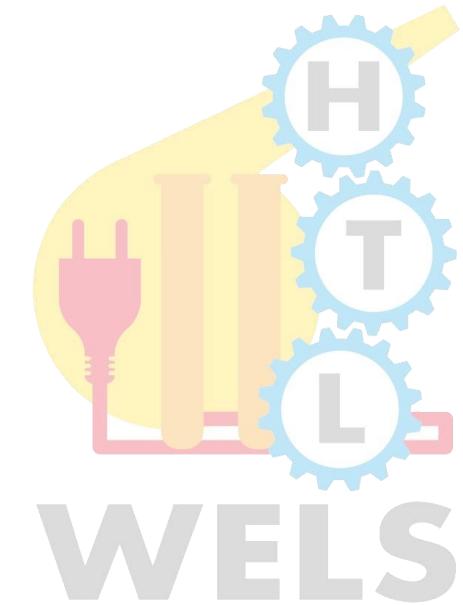
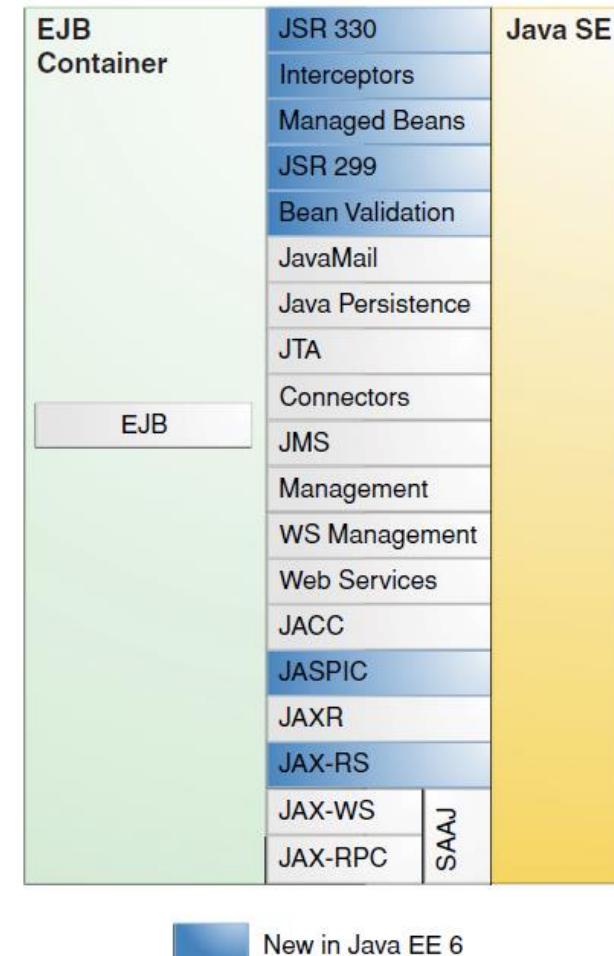
Architecture / Services JEE Containers



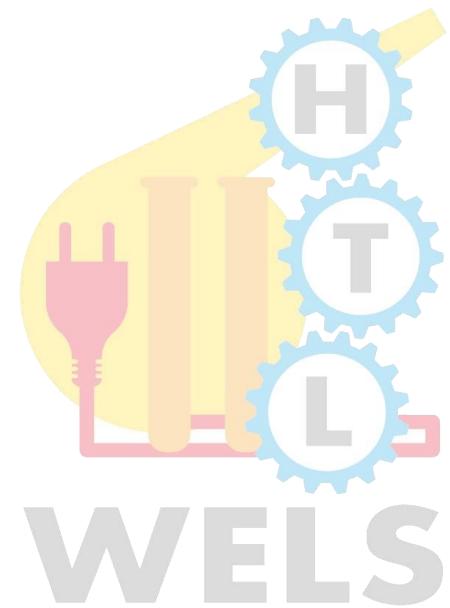
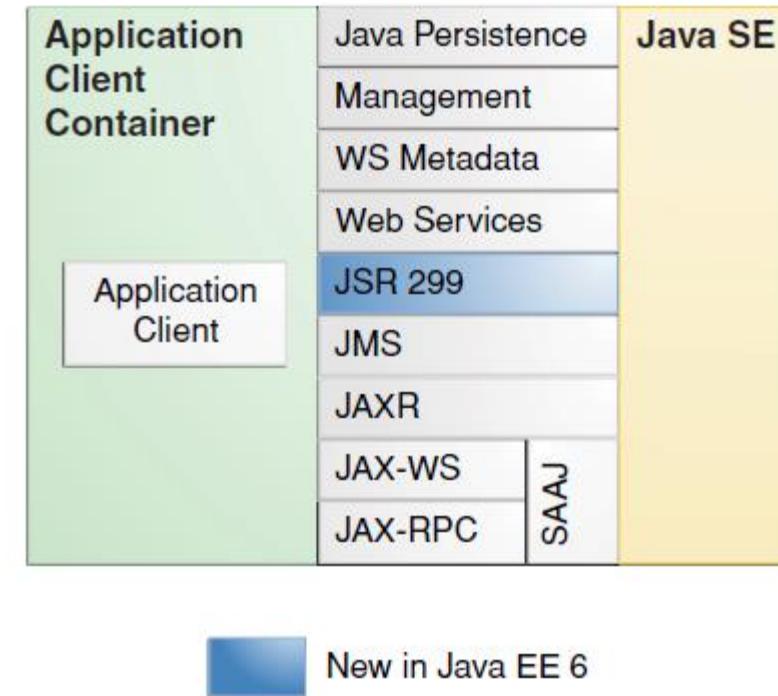
API's eines JEE Web-Containers



API's eines JEE EJB-Containers



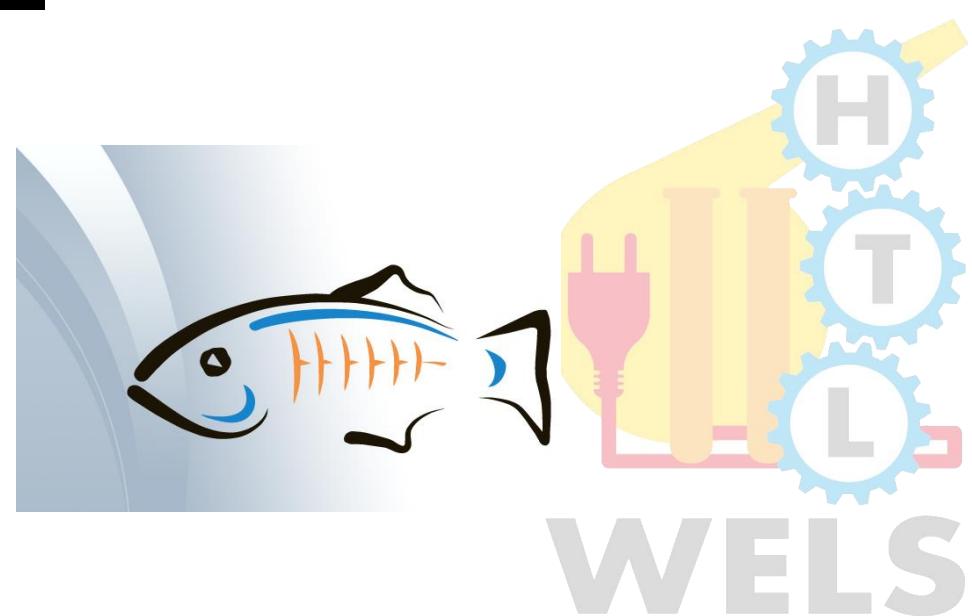
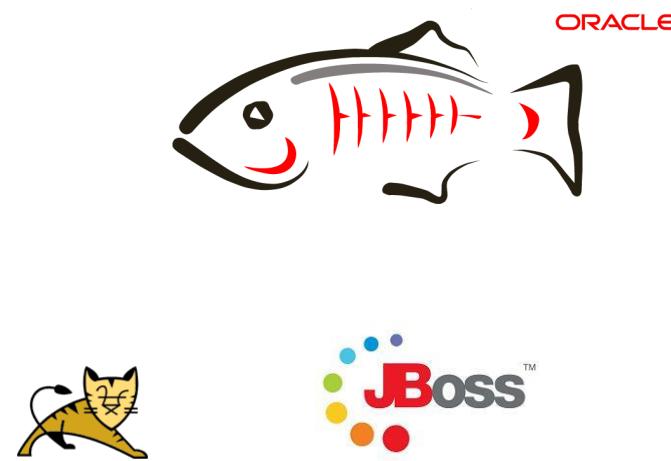
API's eines JEE ACC-Containers



Implementierungen (Application Server)

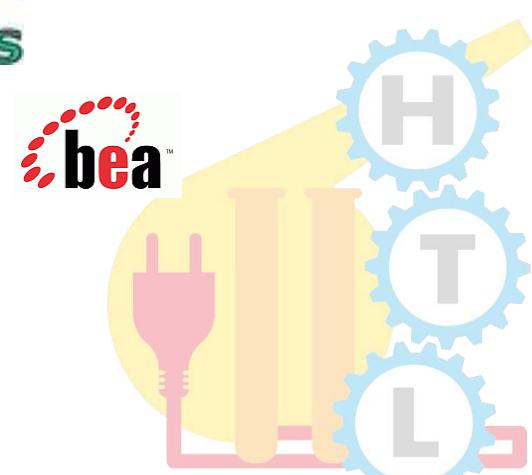


- Eine Implementierung des Java-EE-Standards kann zusätzlich von Sun für die jeweilige Version **zertifiziert** werden.
- Dadurch wird die **grundätzliche Kompatibilität** der Anwendungen zwischen den Servern bestätigt.
- Jedoch zeigt sich in der Praxis oft, dass eine Portierung einer Applikation von einem Java-EE-Server zum anderen mit Problemen verbunden ist. So werden teilweise unbewusst **Hersteller-abhängige Bibliotheken** genutzt.



Komplette JEE Application Server

- **Opensource**
 - **Apache Geronimo** (benutzt wahlweise Apache Tomcat oder Jetty) (Java EE 5)
 - **WildFly** (früher: **JBoss** Application Server)
 - **JOnAS** (benutzt Apache Tomcat) (Java EE 1.4)
 - **Sun GlassFish Enterprise Server** (Java EE 6)
- **Kommerzielle Implementierungen**
 - **Oracle WebLogic** (seit der Übernahme von BEA durch Oracle) (Java EE 5)
 - **SAP NetWeaver Application Server** (Java EE 5)
 - **IBM WebSphere Application Server (WAS)** (Java EE 5)
 - **Oracle Application Server** (Java EE 5)



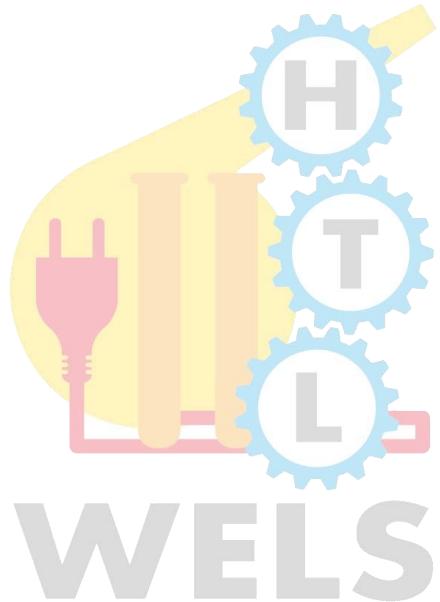
WELS



Teilimplementierungen

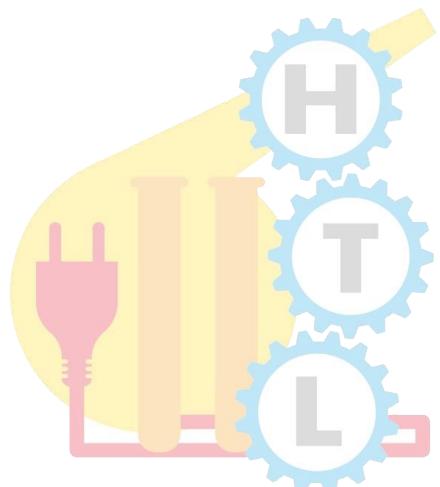
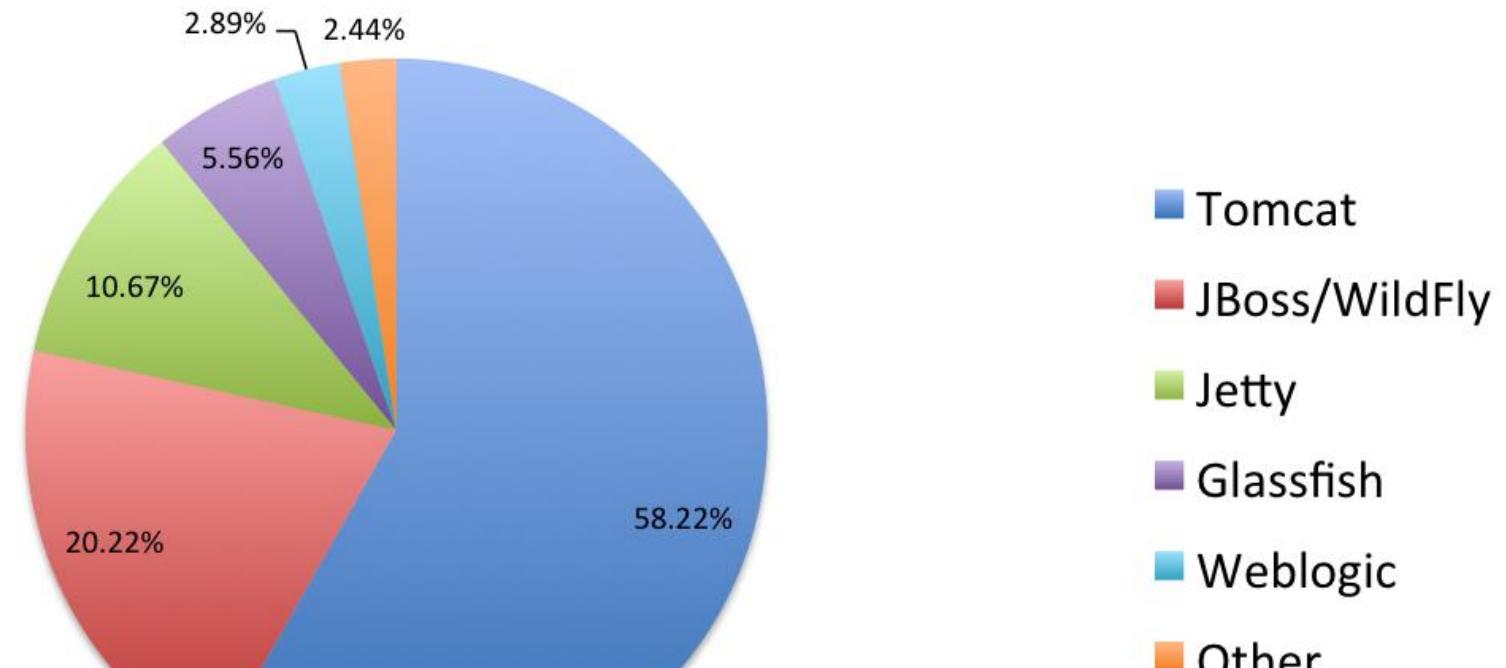
- **Separate Web-Container (Servlet-/JSP-Container)**
 - Apache Tomcat – Open Source
 - Caucho Technology Resin – Open Source
 - Enhydra Server – Open Source
 - Jetty – Open Source
- **Separate EJB-Container**
 - Apache OpenEJB – Open Source

jetty://

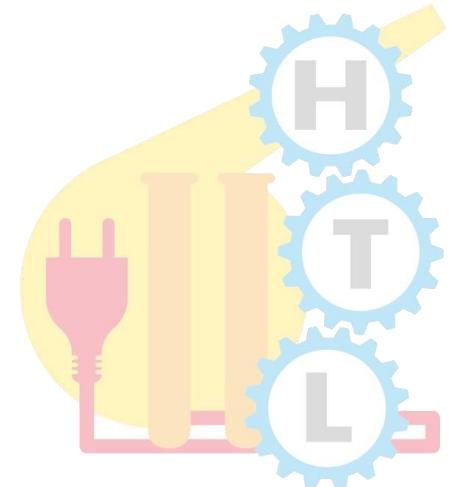


Verbreitung der verschiedenen Application Server

Application server market share 2016

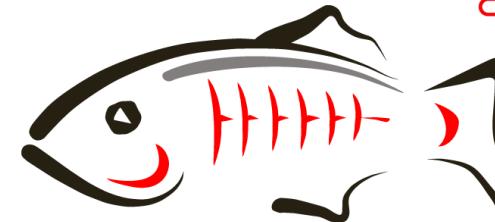


WELS

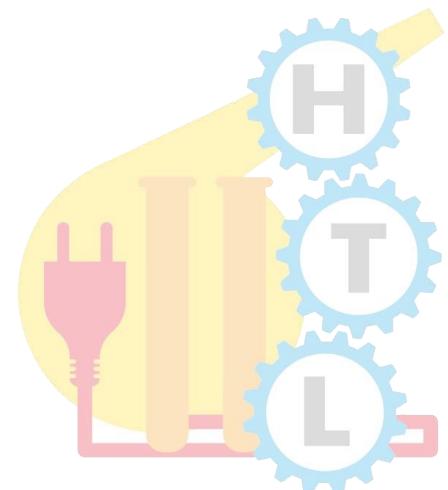


WELS

Verbreitung von Glassfish

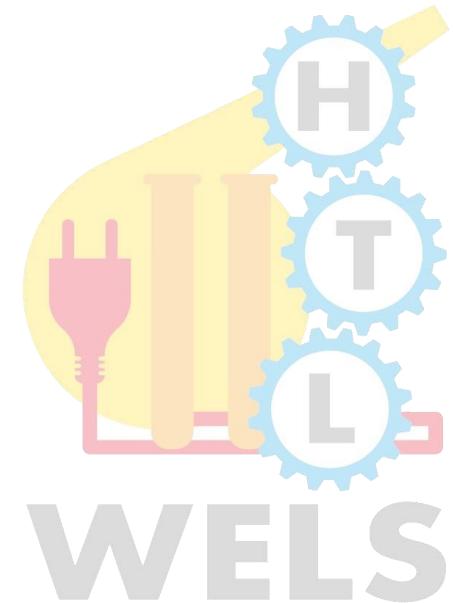
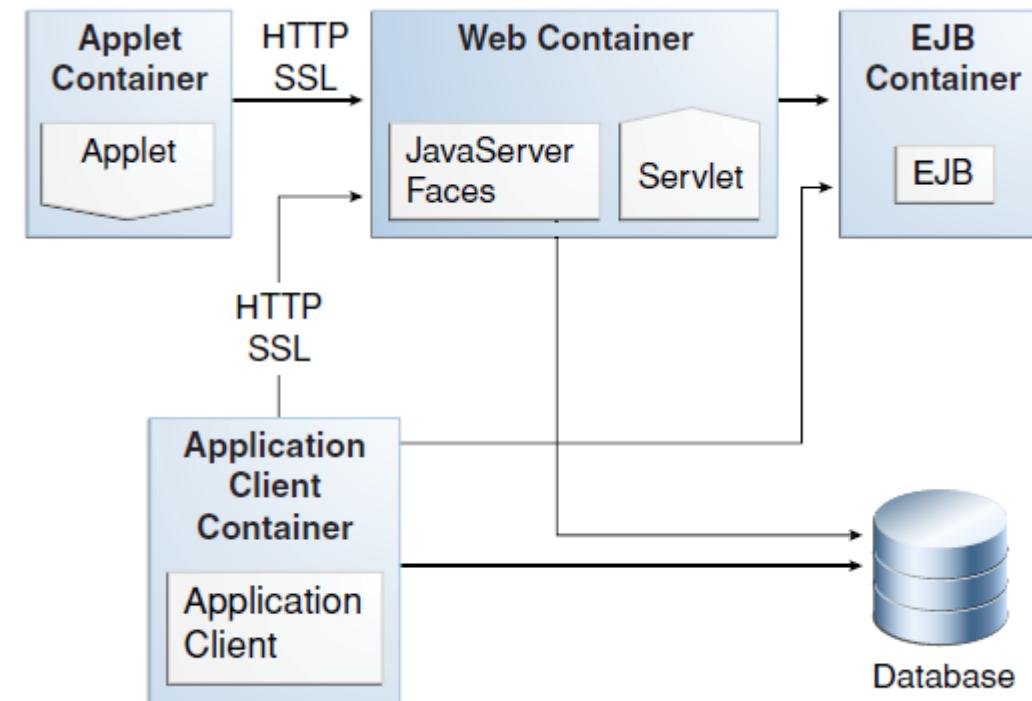


ORACLE®



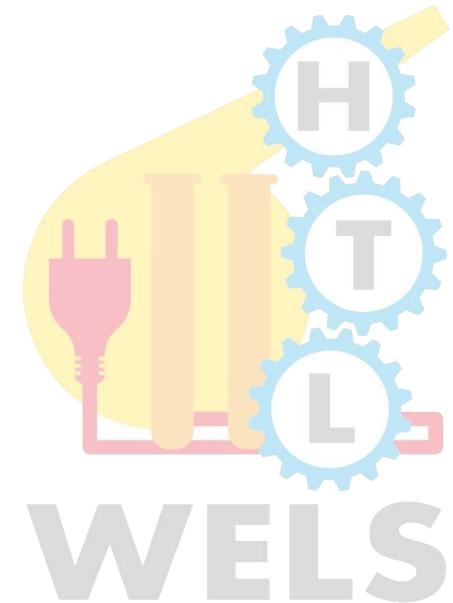
WELS

JEE Containers



Java SE Packaging

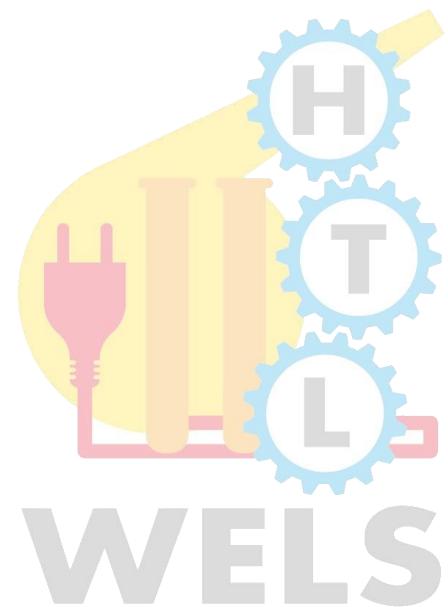
- In der Java SE wird das **JAR-File** als zentrales Strukturierungs- und Gruppierungskonstrukt für die unterschiedlichen Dateien einer Java-Anwendung verwendet.
- Ein JAR-Archive ist ein **Zip-File**, dass Java Klassen, Deployment Deskriptoren, Ressources , Libaries und Sonstiges enthält.
- Es kann, muss aber nicht komprimiert sein.
- Der Vorteil liegt vor allem darin, dass man nur eine Datei handhaben muss.
- Mittels der beiden Paketen „`java.util.jar`“ und „`java.util.zip`“ bietet die Java SE Klassen an, um JAR- oder ZIP-Archive auszulesen oder zu erstellen.



JEE-Packaging (1)

- JEE unterscheidet je nach Art der Anwendung drei Arten von unterschiedlichen Containeren:
- **Application Client Container: JAR-File**
 - Beliebige Anzahl von Java-Objekten
 - Optional: **META-INF** Verzeichnis (für Meta-Informationen zur Archives)
 - META-INF/MANIFEST.MF** (beinhaltet Package- und Erweiterungspezifische Daten)
 - META-INF/applicaitonclient.xml**
- **EJB Container: EAR-File**
 - Enthält ein oder mehrere Session- und/oder Message-Driven Beans
 - Optional: META-INF/ejb-jar.xml → Deployment Descriptor
 - **Kann nur in einem EJB-Container deployed werden**

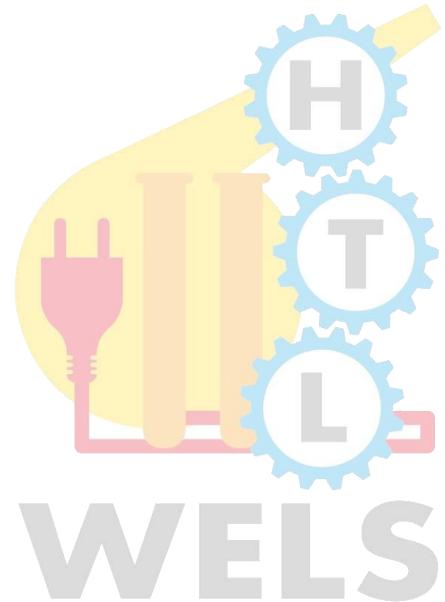
Beschreibung des



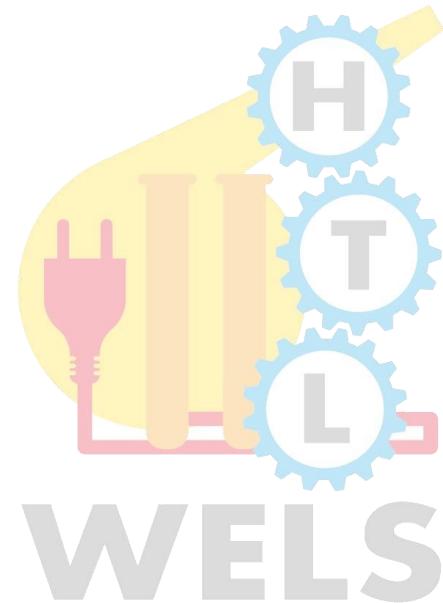
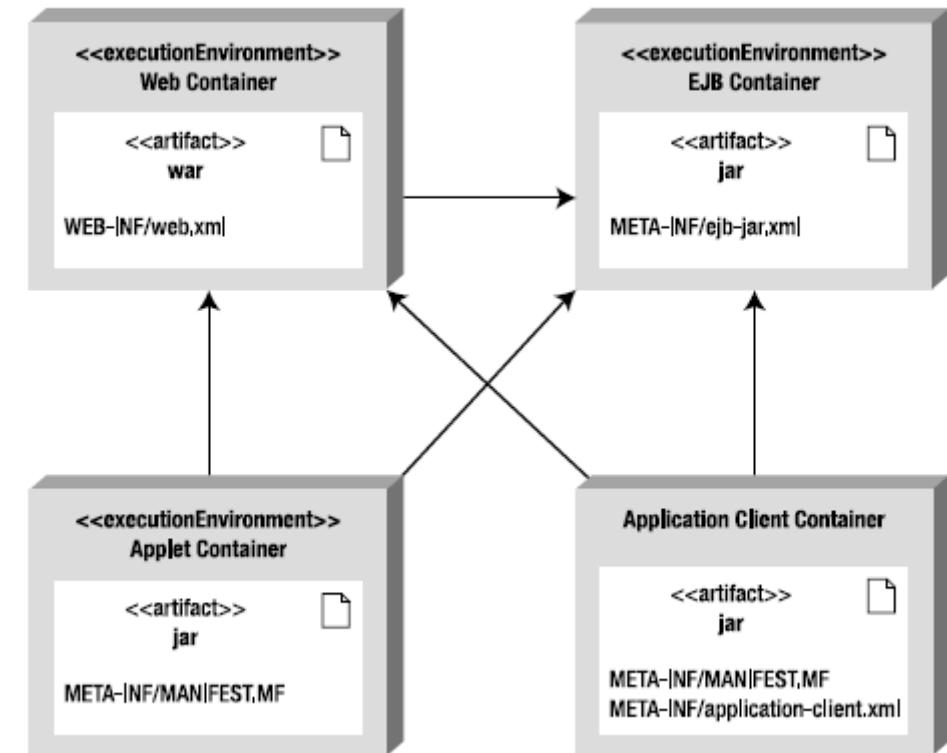
JEE-Packaging (2)

▪ Web Application Container: WAR-File

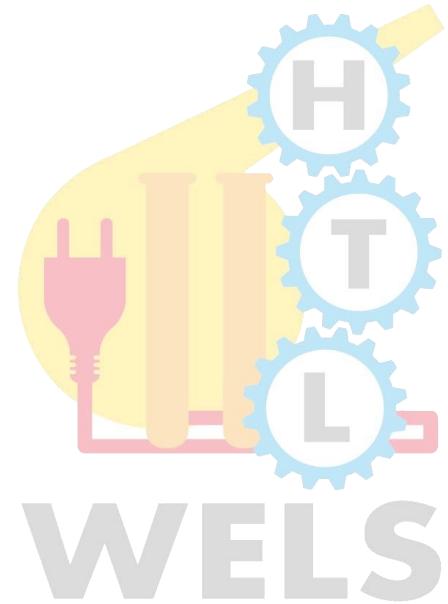
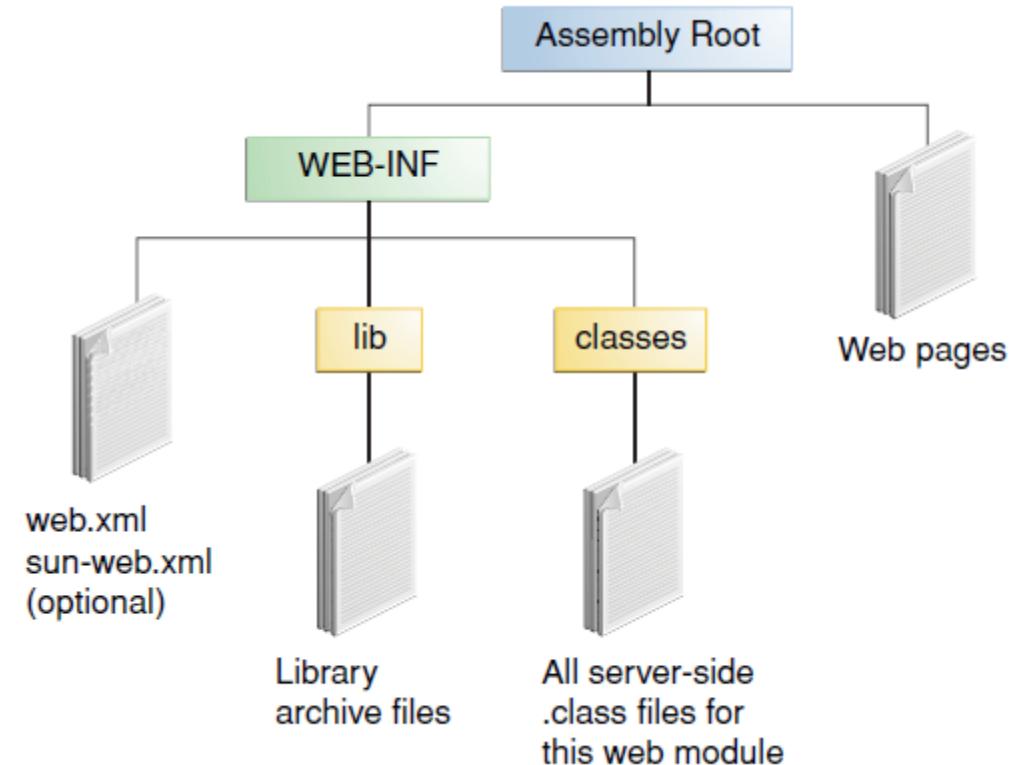
- Web bezogene Dateien: Servlets, JSP, JSF, Web-Services , CSS, ...
- Seit JEE 6.0: EJB Lite Beans
- Optimal: WEB-INF/web.xml → WEB Deployment Descriptor
- Optional: WEB-INF/ejb-jar.xml → Deployment Descriptor
- Optional: WEB-INF/classes → Java.class files



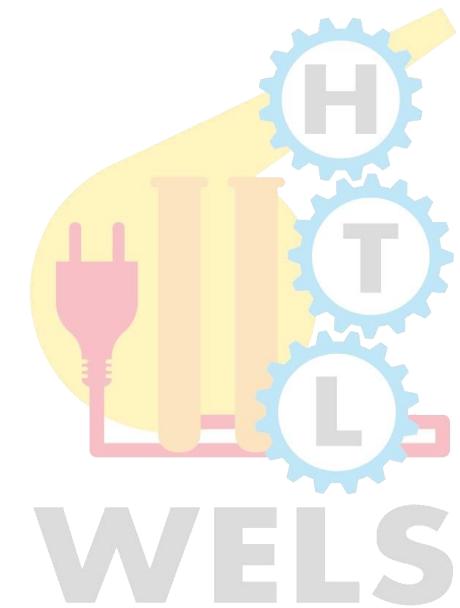
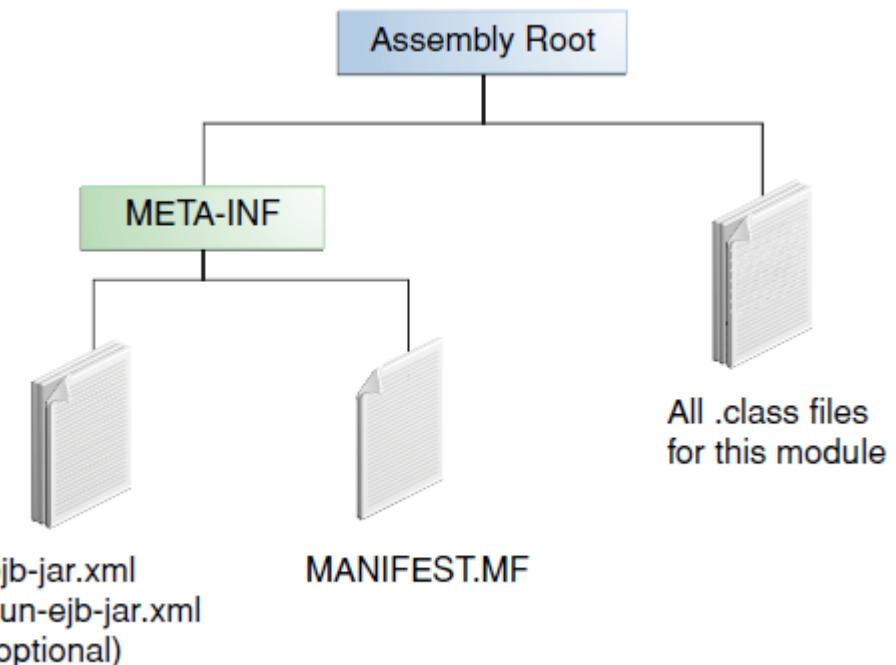
JEE – Packaging (3)

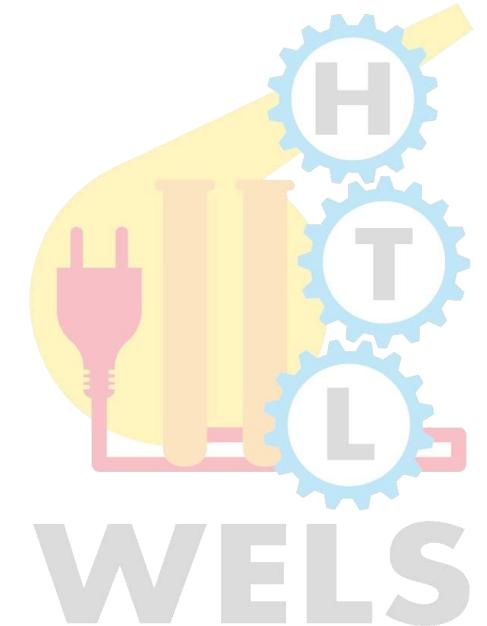


Web Module Structure



EJB Structure



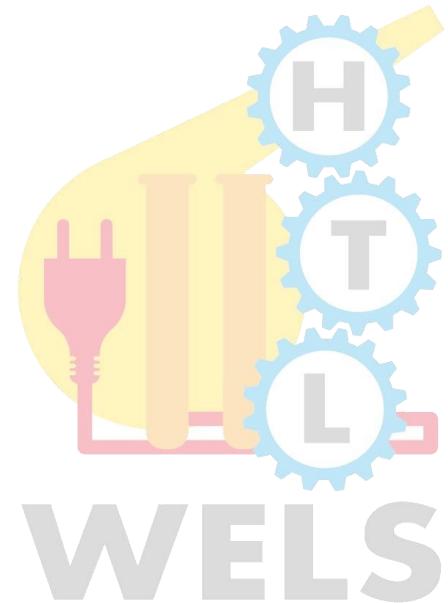


JEE API's

- **Servlets**
- **JSP / JSF**
- **EJB (mit JPA)**

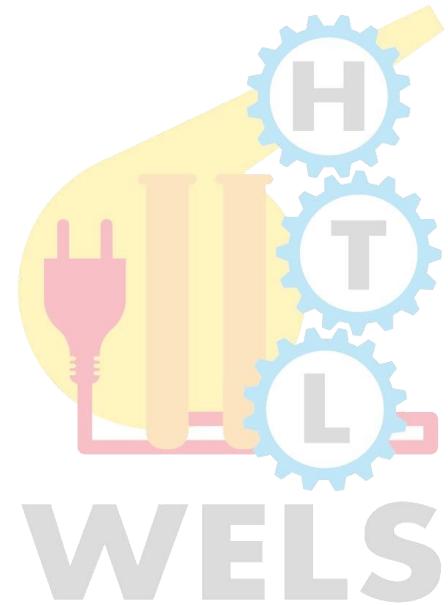
Die Ziele von JEE 6 (JEE 7, JEE 8)

- “Rightsizing” der Plattform
 - Vollständige Plattform vs. Modulare Plattform mit Profilen
- Weitere Vereinfachung der Entwicklung
 - Java EE 5 vereinfachte die Entwicklung mit EJBs/JPA und Web Services Java EE 6 vereinfacht die Entwicklung von Web Applikationen
- Erweiterbarkeit – Plugin Schnittstellen für Open Source Frameworks
- Verbesserung und Vereinfachung der IT-Security



Java EE – Was bringt's wirklich?

- Geringerer Aufwand für Programmentwicklung
- Programmieren wird produktiver
- Anwendungen werden wartbarer
- Standardisierung → Herstellerunabhängigkeit
- „Write Once – Run Anywhere“
- Unterstützung vieler und verschiedener Clients

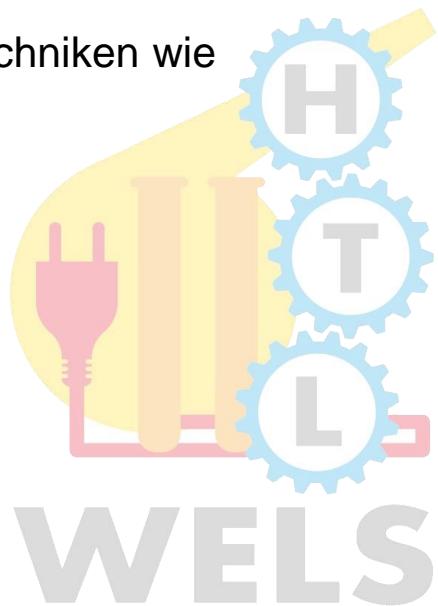


JEE 9 → Jakarta EE 9



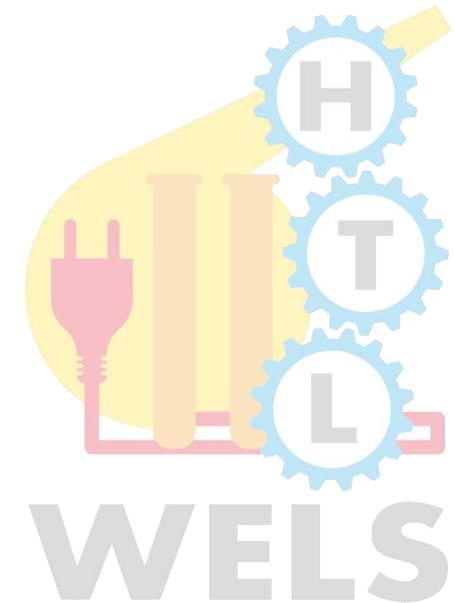
- Am 12. September 2017 hat Oracle bekanntgegeben, dass die gesamte JEE (inklusive der Glassfish-Implementierung) an die Eclipse Foundation übergibt.
- Das Top-Level-Projekt heißt bei Eclipse: Eclipse Enterprise 4 Java (EE4J).
- Am 26. Februar 2018 wurde verlautbart dass aus urheberrechtlichen Gründen Java EE nun Jakarta EE heißen wird.
- Java EE 8 ist die Basis für die Weiterentwicklung.

- Zukünftige Entwicklungs-Ziele:
 - **Cloud Native Java** (Microservices und eine bessere Integration von Container-Deployment-Techniken wie Kubernetes, Docker und Co.)
 - **Neues Governance Modell** (Entschärfung des JCP – kein Einfluss von Oracle mehr)



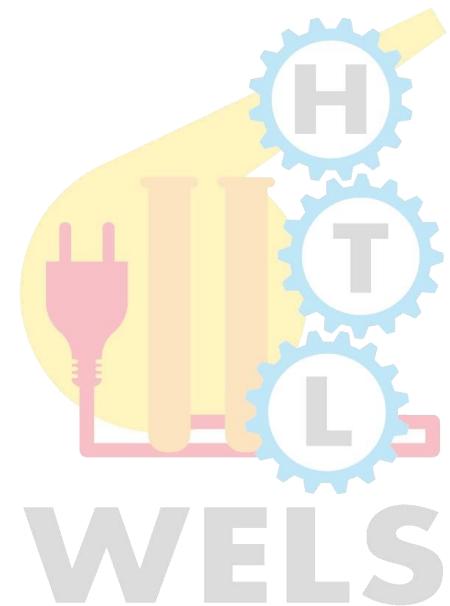
Neustrukturierung

- Jakarta Enterprise Beans
- Jakarta Persistence API
- Jakarta Contexts and Dependency Injection
- Jakarta EE Platform
- Jakarta API for JSON Binding
- Jakarta Servlet
- Jakarta API for RESTful Web Services
- Jakarta Server Faces
- Jakarta API for JSON Processing
- Jakarta EE Security API
- Jakarta Bean Validation

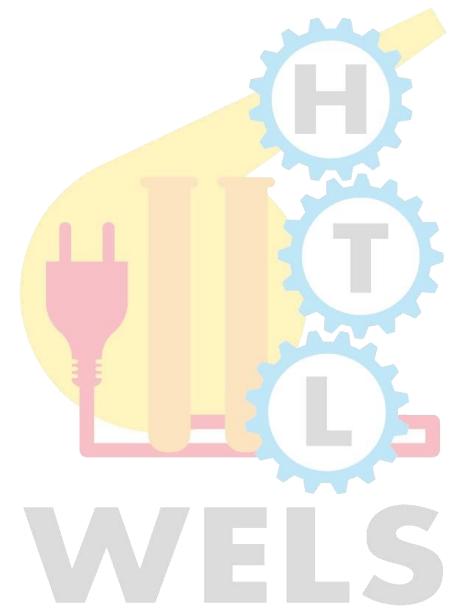
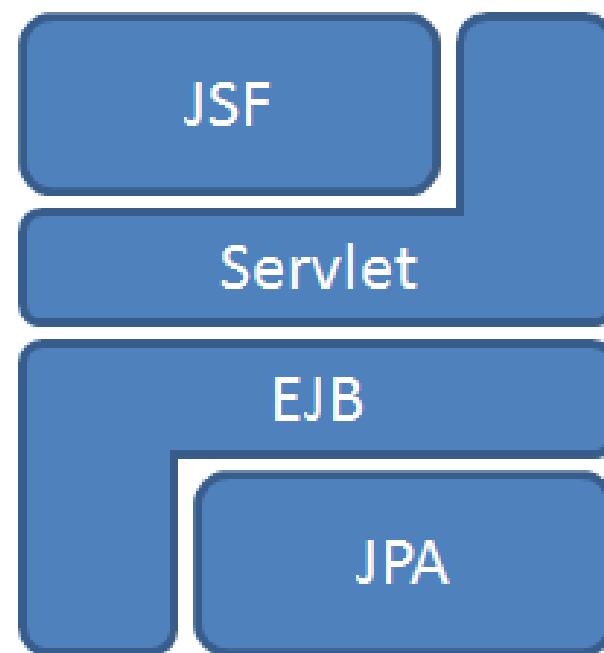


Generelle Änderungen von JEE 6 (JEE 7)

- **Einführung von Java EE Profilen (siehe Exkurs: Java Profile)**
 - Profile als Sub- oder Supermenge von Java EE
 - Web Profile(Profile A/B) als erster Vorschlag
 - Enthält Submenge von Java EE Technologien für Web Applikationen wie Servlet 3.0, JSP 2.2, JSF2.0(*), Web Beans(*), JAX-RS, JAX-WS etc.
- **Deployment von EJBs in Web Modulen**
 - Betonung des POJO Charakters von EJBs
 - Vereinfachtes Packaging für Entwickler
- **Pruning für veraltete Technologien**
 - Verschlanken der Plattform durch Entfernen von APIs

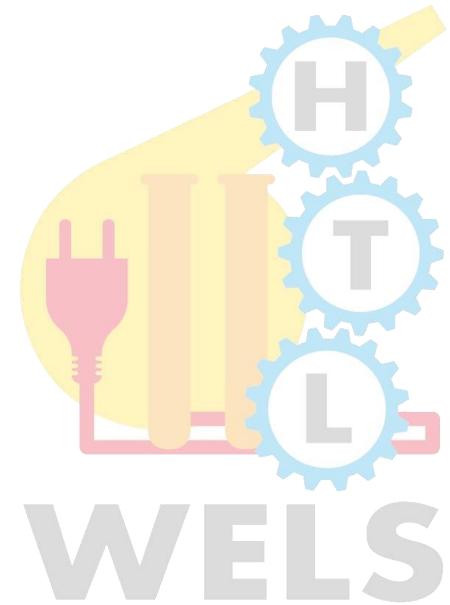


Einordnung EJB / JPA / Servlets / JSF



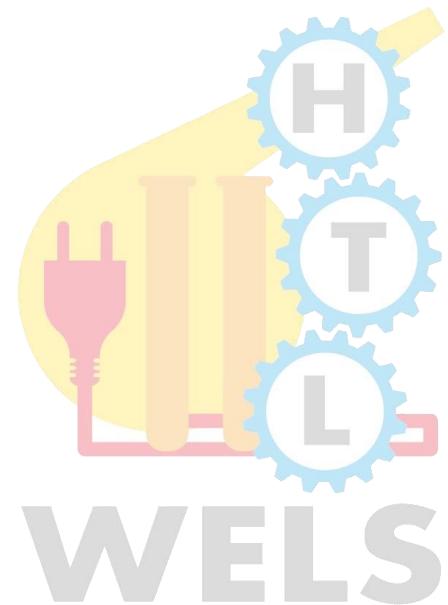
Exkurs: JEE Profile #1

- Ein Profil ist ein durch den JCP definierte und somit standardisierte **Konfiguration** der Java-EE-Plattform, die "optimal" auf eine bestimmte **Art** von **Anwendungen** zugeschnitten ist.
- Ein Profile kann dabei ein **Subset** der zugehörigen Java-EE-Version beinhalten aber auch **zusätzliche Anforderungen** bzw. **Technologien** definieren.
- Denkbar wäre zum **Beispiel** ein **Web Service Profile**, dass bewusst auf alle Visualisierungstechnologien wie JSF, JSTL und JSP verzichtet.
- Ebenfalls denkbar wäre aber auch ein spezielles Java EE Portlet Profile, dass zusätzlich zu den bereits in Java EE enthaltenen APIs das Java Portlet API (JSR 286) voraussetzt.



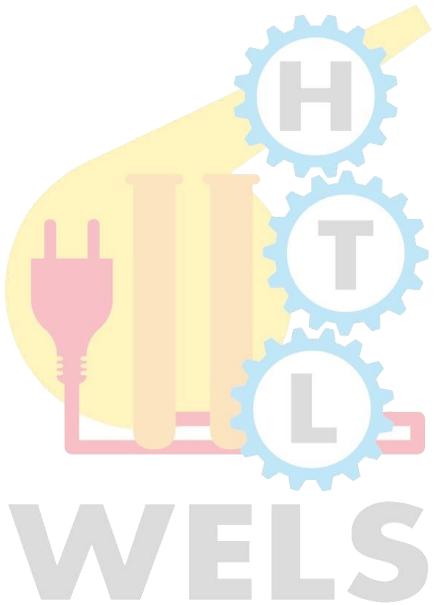
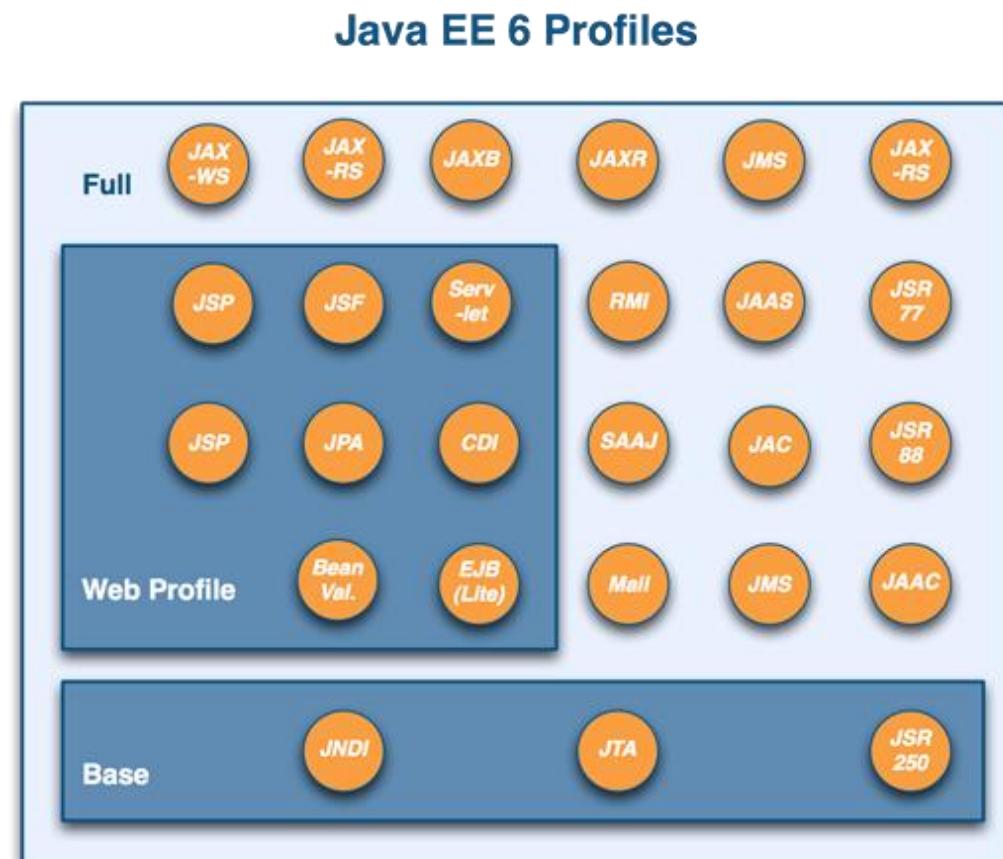
Exkurs: JEE Profile #2

- Alle Profile haben ein sehr schmales Basisset an API's und Features gemeinsam:
 - Resource and Component Lifecycle Annotations aus dem JSR 250 (@Resource, @Resources, @PostConstruct, @PreDestroy)
 - JNDI „java: „ Naming Context
 - JTA (Java Transaction API)
- Optional:
 - RMI/IOP Interoperability Requirements
 - Unterstützung von java:comp/ORB

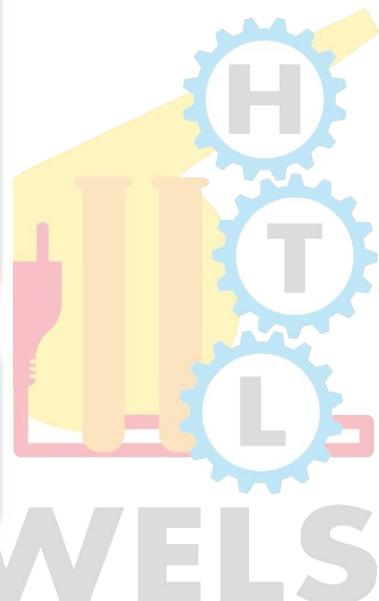
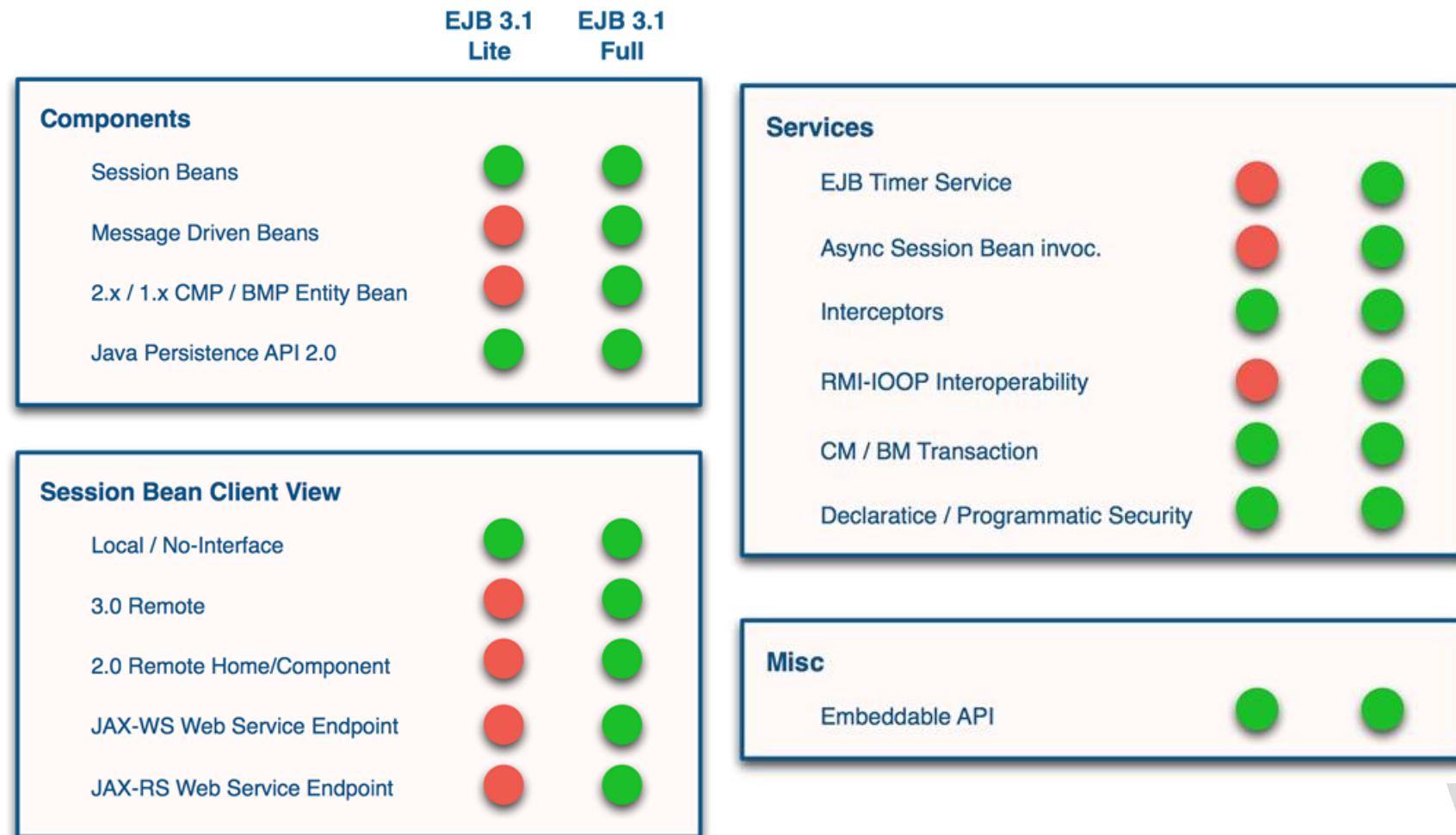


Exkurs: JEE Profile #3

- Aktuell existiert eine Profildefinition: Web Profile mit EJB Lite



EJB Lite



Deployment – EJB Lite

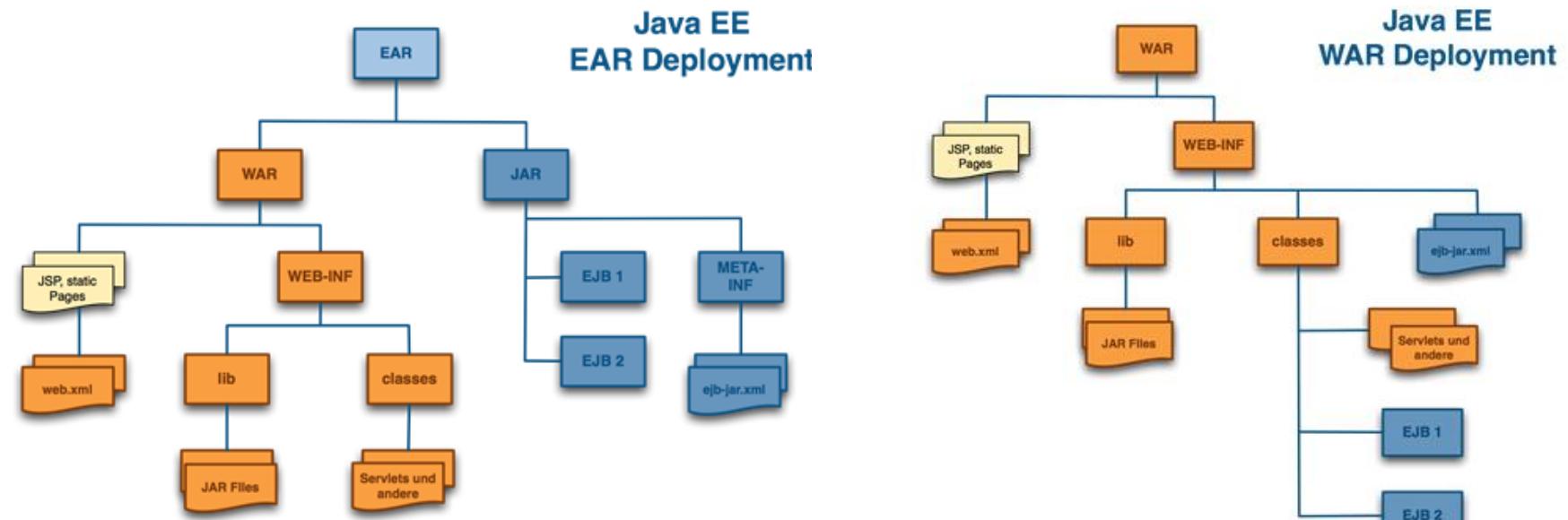
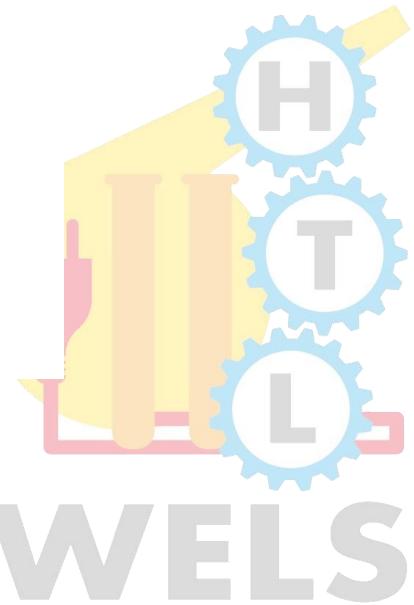
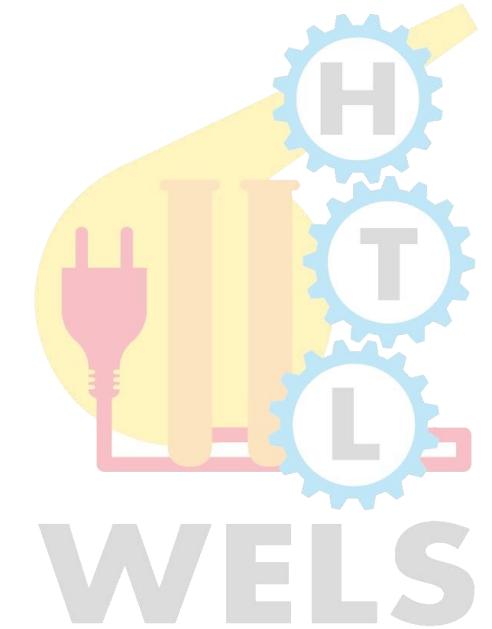
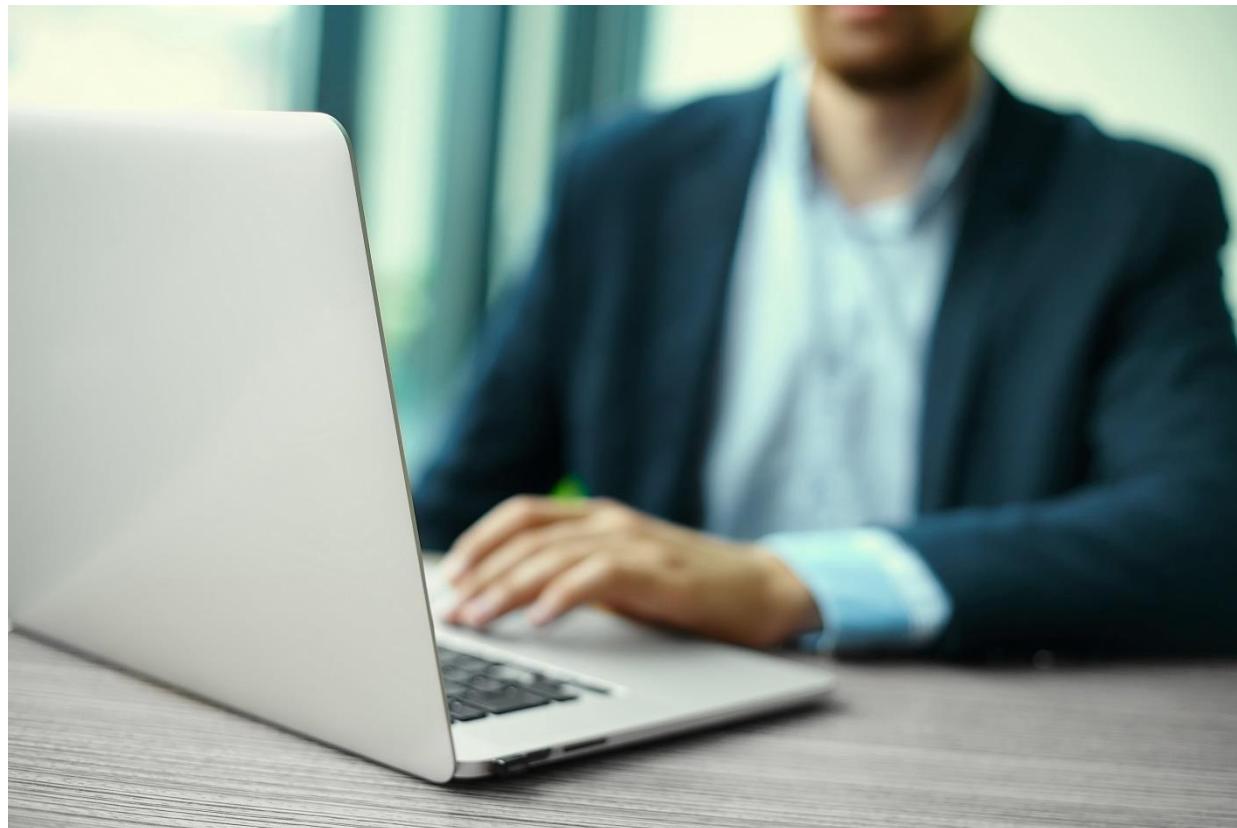


Abb. 3: Java-EE-6-Deployment-Szenarien

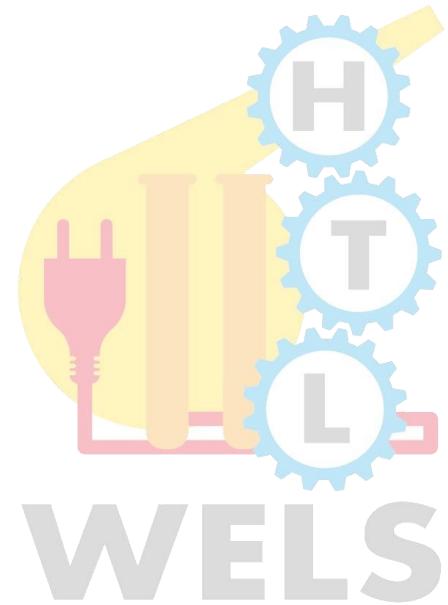




Enterprise Java Beans (3.2)

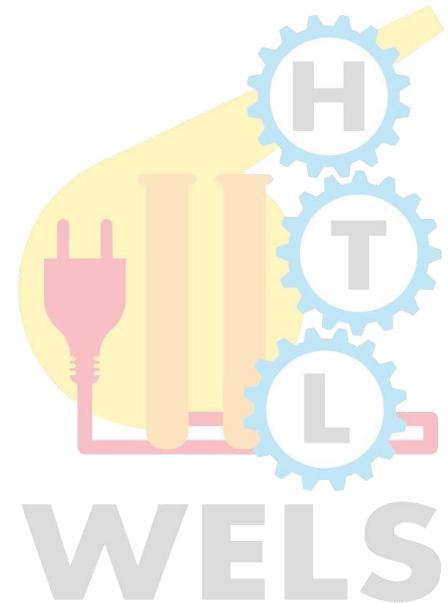
„Business Systeme“ → Beispiele

- Webshop
- Bestellsystem für Zulieferer etwa der Automobilindustrie
- Verwaltungssystem für Krankenhaus (Patientenakten, Medikamentenbestellung, Arbeitsorganisation, etc.)
- Flughafenverwaltung
- Vernetzung der Filialen einer bundesweiten Ladenkette mit der Zentrale
- Human Resources Management einer global operierenden Firma



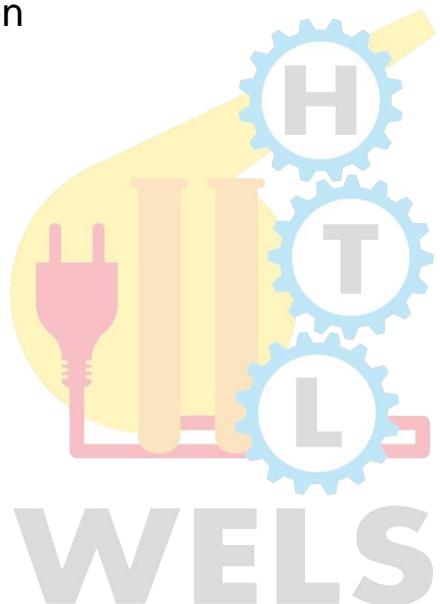
„Business Systeme“ → Problemfelder #1

- **Kompatibilität:** Komponenten unterschiedlicher Plattformen sollen zusammenarbeiten
- **Remote Method Invocation (RMI/RPC):** Rechner A will Methode auf Rechner B aufrufen
- **Sicherheit:** Verschiedene Clients sollen verschiedene Rechte bekommen
- **Back-End Integration:** Existierende Systeme ("Back Ends") wie Datenbanken sollen eingebunden werden
- **Parallelität** (durch Prozesse oder Threading), wenn mehrere Clients einen Server ansprechen
- **Transaktionen/Synchronisierung** auf persistenten Daten



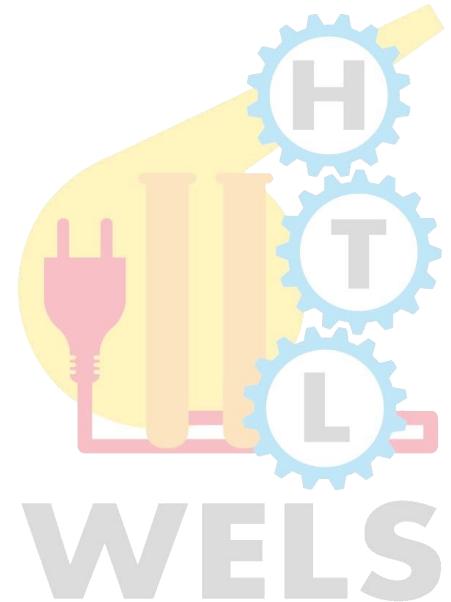
„Business Systeme“ → Problemfelder #2

- **Caching:** Von mehreren Clients benutzte Daten nur einmal laden
- **Messaging:** Oft wollen die Clients asynchron zum Server arbeiten, d. h. beim Aufruf eines Servers nicht blockiert werden
- **Protokollierung und Überwachung:** Wenn etwas schief geht, sollte es Protokolldaten geben, um den Fehler zu finden
- Bilden von "**Resource Pools**": Die Anzahl vorhandener Instanzen eines Diensts sind begrenzt: Wiederverwendung ungenutzter Instanzen
- **Sauberes Abschalten**, ohne allen Clients einfach den Boden unter den Füßen wegzuziehen
- **Clustering** zur Ausfallsicherheit: Mehrere Server mit demselben Zustand (Abgleich bei Änderungen?)



„Business Systeme“ → Problemfelder #3

- **Load Balancing:** mehrere Server für den gleichen Dienst ⇒ den Server mit der geringsten Last ansprechen
- **Transparent Fail-Over:** Bei Ausfall eines Servers oder eines Teils des Netzwerks sollte ein anderer dessen Funktion übernehmen können
- **Dynamisches Ersetzen** eines Programms durch eine neuere Version ("Redeployment")
- ...



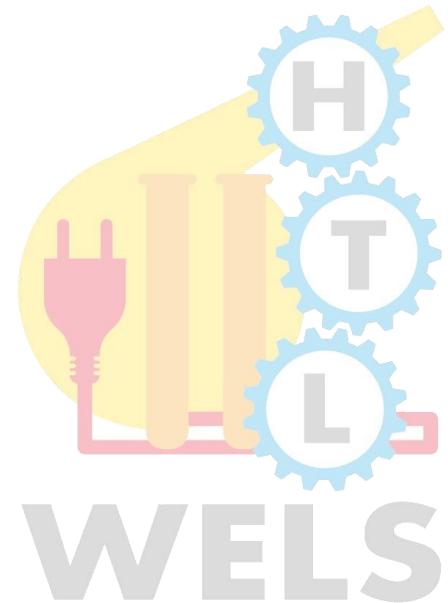
Middleware

- Um die vorher beschrieben Probleme zu lösen brauche ich Systeme die mir die entsprechende Flexibilität bieten.
- Solche Systeme werden als

Middleware

bezeichnet

- Bekannte Middleware-Ansätze
 - Corba,
 - J2EE/JEE und
 - .NET



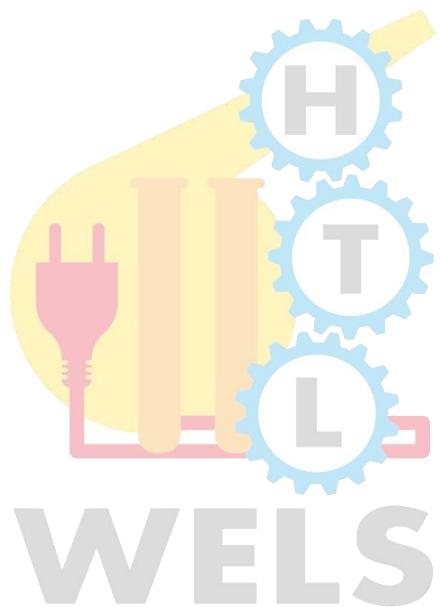
Middleware – Vorteile / Nachteile

▪ Vorteile

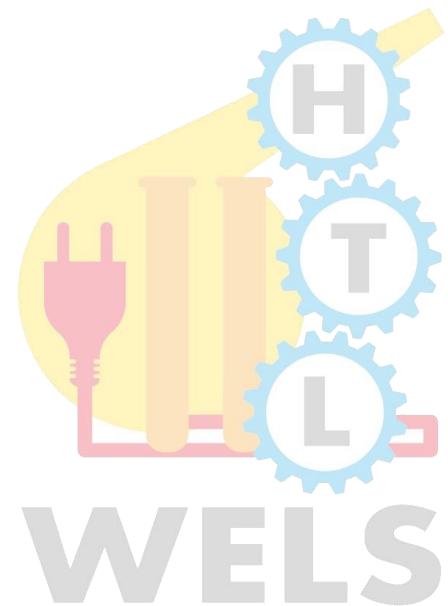
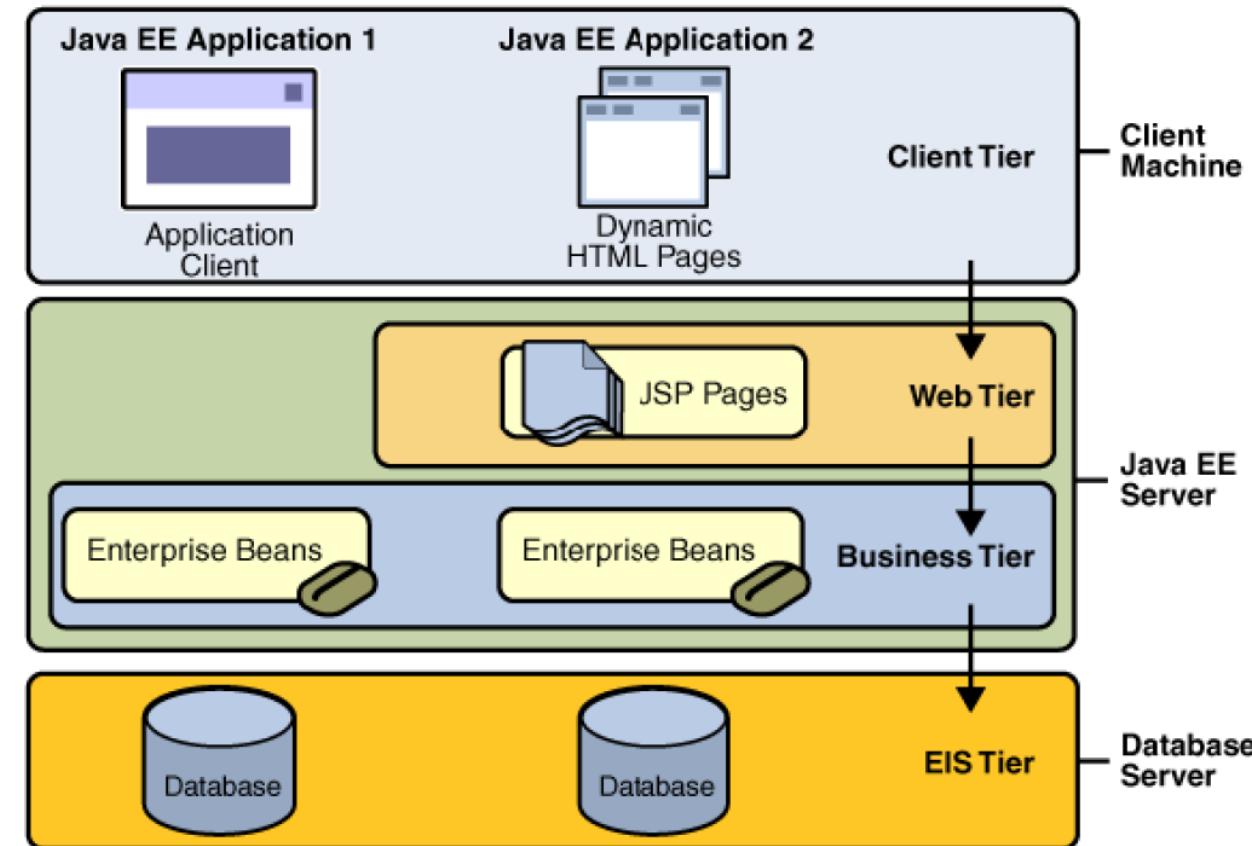
- keine Expertise für die Probleme notwendig, die Middleware löst
- keine Dokumentation oder Wartung für Middleware
- schnellere Entwicklung
- bessere Qualität
- Austauschbarkeit

▪ Nachteile

- potentiell geringere Effizienz, da Middleware sehr generisch und an vielerlei Anforderungen anpassbar sein muss
- evtl. hoher Aufwand, sich in die Middleware einzuarbeiten



Schichtenmodell von J2EE



Middleware: Corba vs. JEE (EJB)

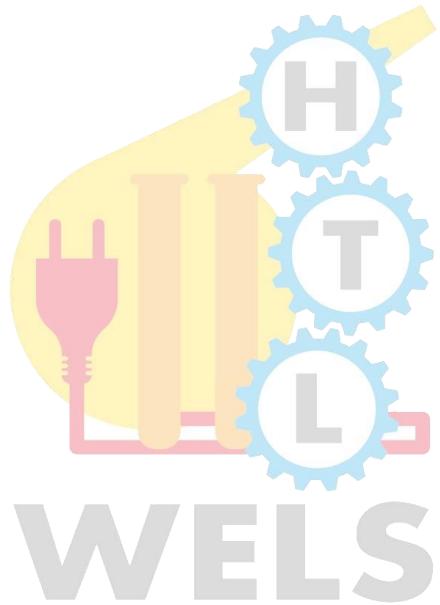
Corba: **Explizite** Middleware

```
transfer(Account acc1,  
         Account acc2,  
         Amount x) {  
    security_check();  
    start_transaction();  
    acc1val = db_load();  
    acc2val = db_load();  
    db_store(acc1val - x);  
    db_store(acc2val + x);  
    end_transaction();  
}
```

EJB (und auch .NET):

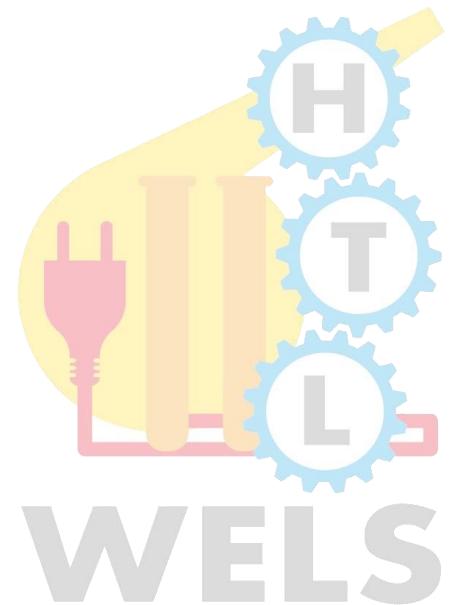
Implizite (auch **deklarative**)
Middleware

```
transfer(Account acc1,  
         Account acc2,  
         Amount x) {  
    acc1.addValue(-x);  
    acc2.addValue(x);  
}  
+ deployment descriptor  
oder Annotationen
```



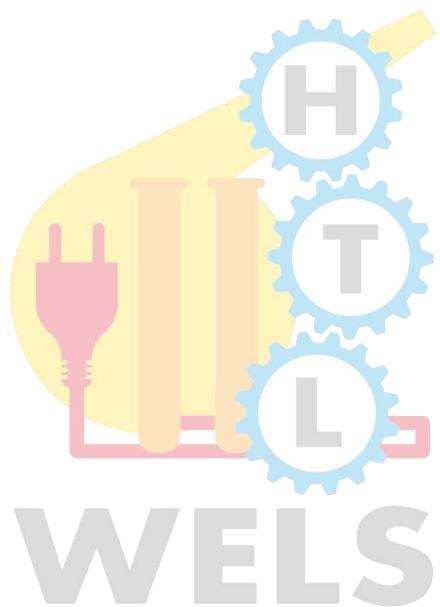
Vorteile Implizite Middleware

- Einfacher zu schreiben: **Kein** "aufgeblähter Code"
- **Middleware-Code** für Security, Transaktionen etc. steht zur Verfügung
- **Einfacher zu warten:** Keine Änderung am Code bei Änderung der Middleware
- Einfacher zu unterstützen: Sicherheit oder DB lässt sich ändern, ohne dass der Sourcecode der Applikation offengelegt werden muss
- aber: schwieriger zu debuggen!



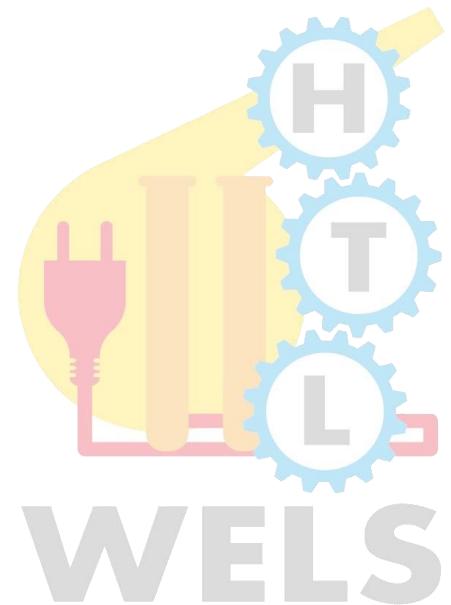
Was ist eine Java Bean?

- **Java Beans** sind **Software-Komponenten** für die Programmiersprache **Java**. JavaBeans entwickelten sich aus der Notwendigkeit heraus, GUI-Klassen (AWT, Swing) einfach instanziieren (**Reflexion**) und übertragen (**RMI**) zu können.
- Java Beans werden auch als Container zur Datenübertragung verwendet. Daher zeichnen sich alle JavaBeans durch folgende Eigenschaften aus:
 - Öffentlicher **Standardkonstruktor** (Default constructor)
 - **Serialisierbarkeit** (Serializable)
 - Öffentliche **Zugriffsmethoden** (Public Getters/Setters)
- Auf Grund dieser Eigenschaften eignen sich JavaBeans auch als Datenobjekte für **Persistenz-Frameworks**



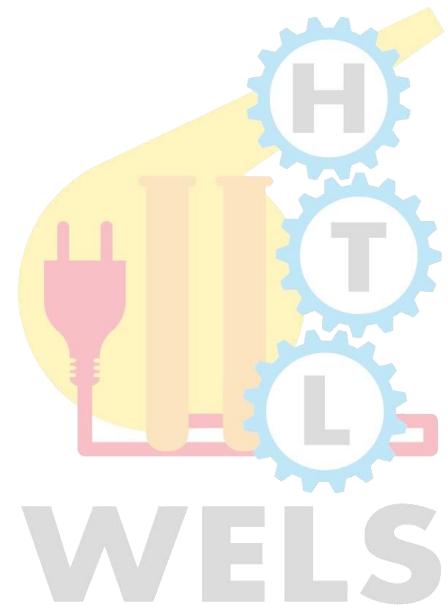
Was ist eine Enterprise Java Bean?

- **Enterprise JavaBeans (EJB)** sind **standardisierte Komponenten** innerhalb eines **Java-EE-Servers** (Java Enterprise Edition). Sie vereinfachen die Entwicklung komplexer **mehrschichtiger verteilter Softwaresysteme** mittels Java. Mit Enterprise JavaBeans können wichtige Konzepte für Unternehmensanwendungen, z. B. Transaktions-, Namens- oder Sicherheitsdienste, umgesetzt werden, die für die **Geschäftslogik** einer Anwendung nötig sind.



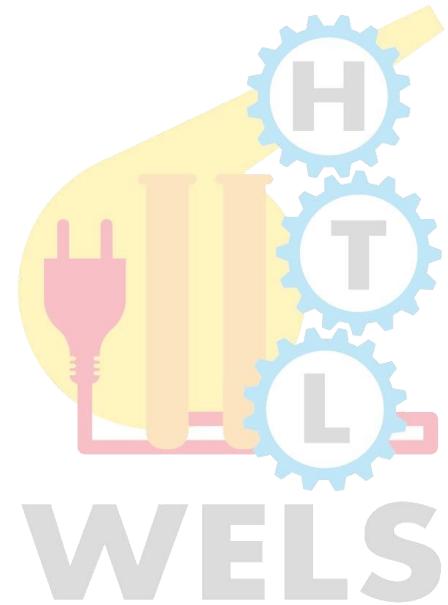
Definition

- **Enterprise JavaBeans (EJB)** ist eine Technologie für **Server-Komponenten** im Rahmen großer Unternehmens-Anwendungen
 - d.h. Overkill für kleine Anwendungen
 - macht sich erst ab einer gewissen Größenordnung bezahlt => Skalierbarkeit (immer gleich langsam)
- EJB gehört zur **Middleware**
 - Business Tier
 - zwischen Datenbank- und Präsentationsschicht
 - zwischen Applikationen
 - zwischen Prozessen, zwischen Rechnern



Rückblick EJB 2.1

- Zu jeder EJB mussten
 - **zahlreiche Interfaces** implementiert werden
 - manche Methoden, die lt. Spezifikation zwingend erforderlich waren, tauchten aber in keinem Interface auf
 - **Deployment-Deskriptoren** geschrieben werden
 - sehr viel XML
 - EJBs sind schwergewichtig
 - für **eine Bean** waren **fünf Artefakte** zu implementieren

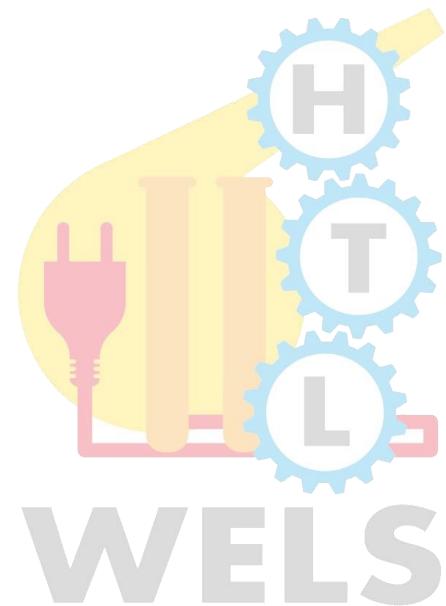


Vergleich EJB 2.1 ↔ EJB 3.0

- 97 % weniger XML
- 36 % weniger Klassen

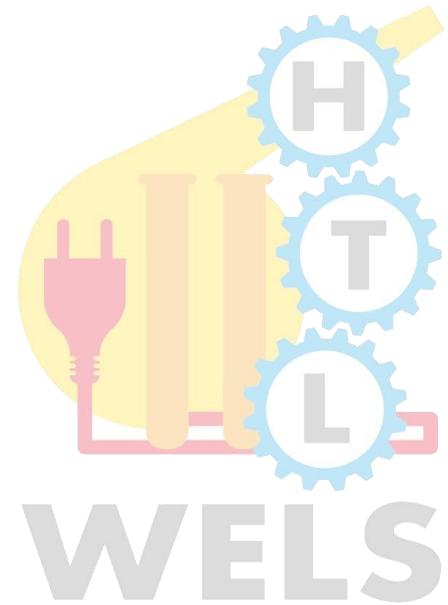
Table 1: Summary of Findings

Application Name	Item Measured	J2EE 1.4 Platform	Java EE 5 Platform	Improvement
AdventureBuilder	Number of classes	67	43	36% fewer classes
	Lines of code	3,284	2,777	15% fewer lines of code
RosterApp	Number of classes	17	7	59% fewer classes
	Lines of code	987	716	27% fewer lines of code
	Number of XML files	9	2	78% fewer XML files
	Lines of XML code	792	26	97% fewer lines of XML code



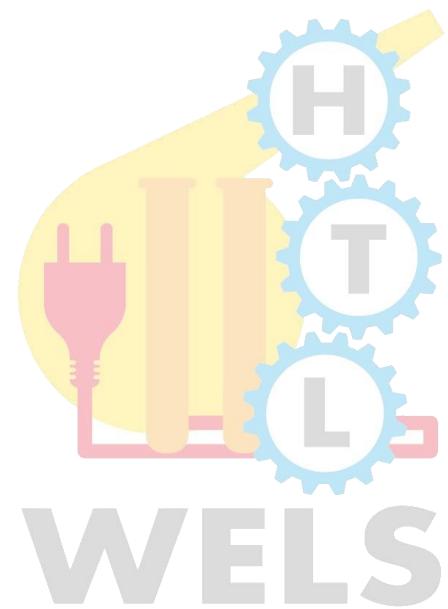
Ziele der EJB-Architektur

- Anwendungsprogrammierer erstellt nur Anwendungslogik und zwar komponentenbasiert ...
- Alle wichtigen, anwendungsunabhängigen Dienste werden vom EJB-Container realisiert
 - Nutzung der Dienste ist **implizit**
 - wird nicht ausprogrammiert
 - Spezifikation und Konfiguration der Dienste bei Erstellung und/oder beim *Deployment* der EJB
- EJB-Container übernimmt Rolle eines „**Component Transaction Monitors**“
 - realisiert Komponentenmodell, Transaktions- und Ressourcen-Management

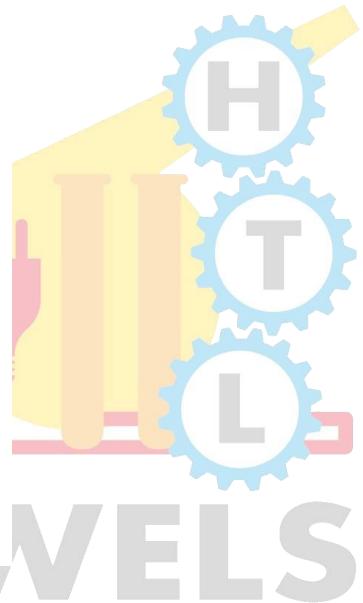
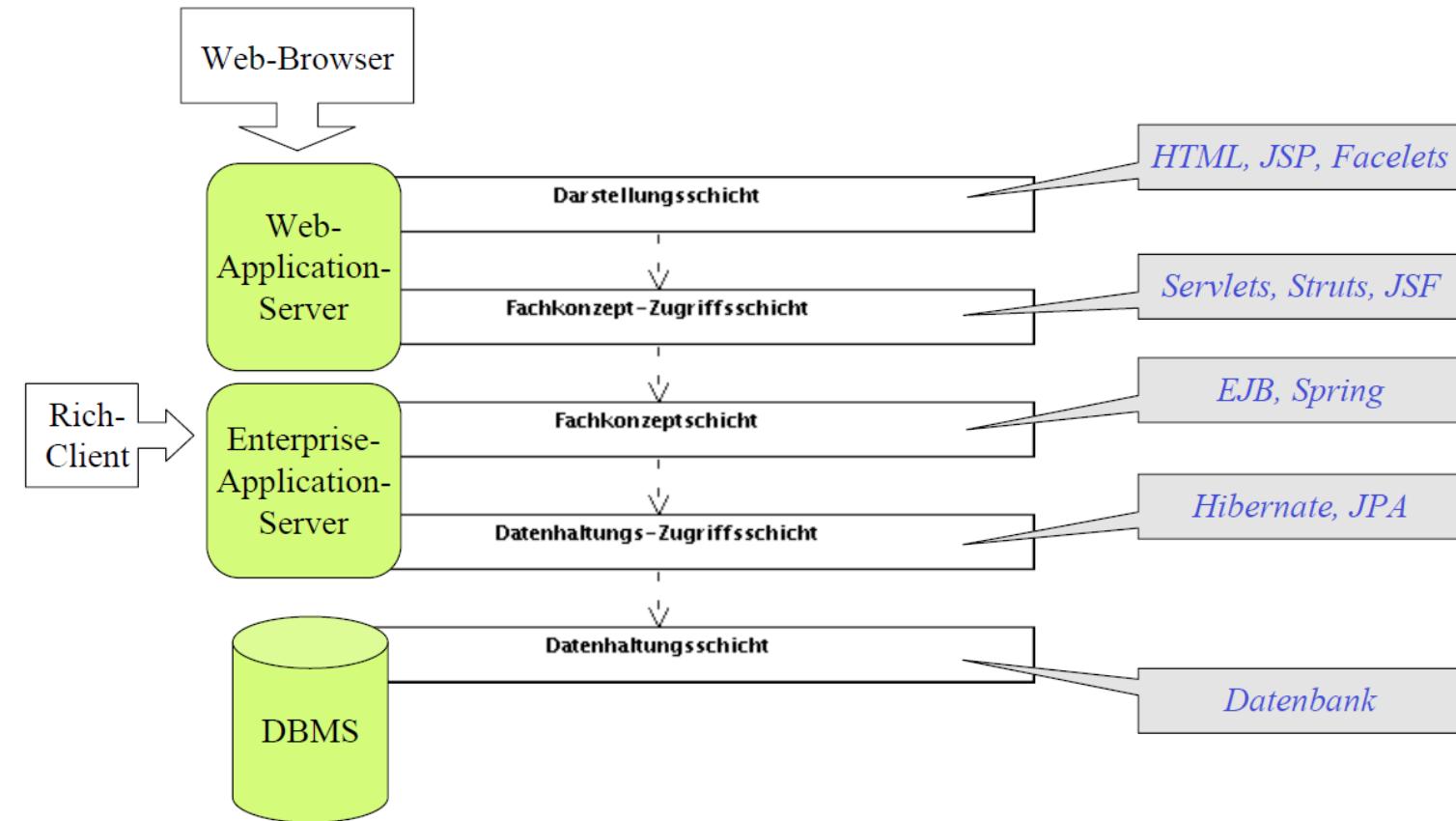


Component Transaction Monitor

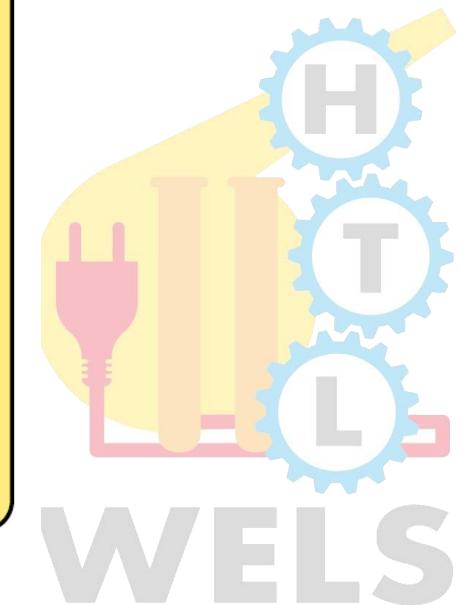
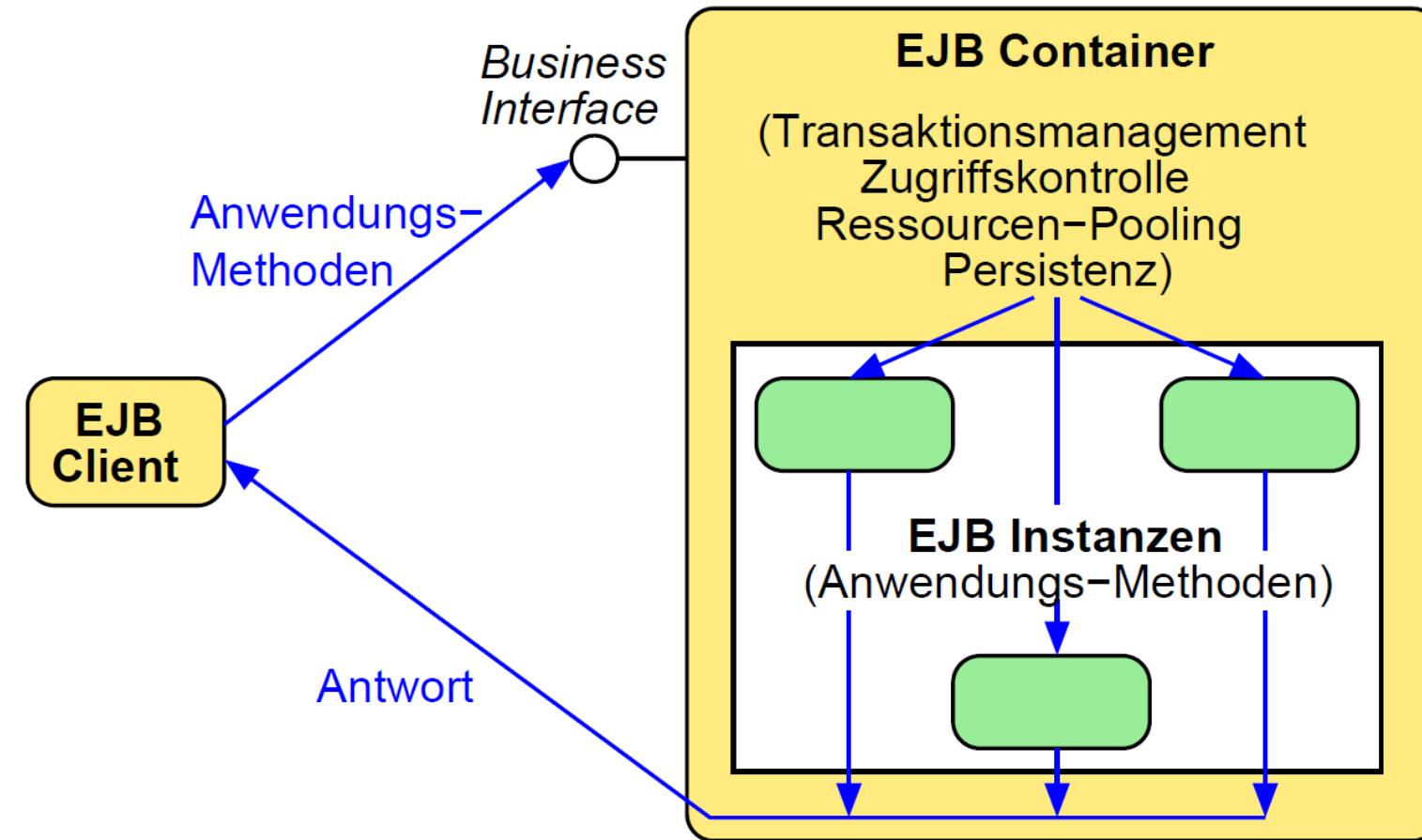
- Vereinigung von **Transaktionsmonitor** und **ORB**
- Aufgabe: automatisches Management von
 - Transaktionen
 - Persistenz von Objekten
 - Sicherheit (insbes. Zugriffskontrolle)
 - Ressourcenverwaltung (z.B. *Pooling*)
 - Objektverteilung
 - Nebenläufigkeit
- **Fokus: zuverlässige, transaktionsorientierte Anwendungen mit vielen Nutzern**



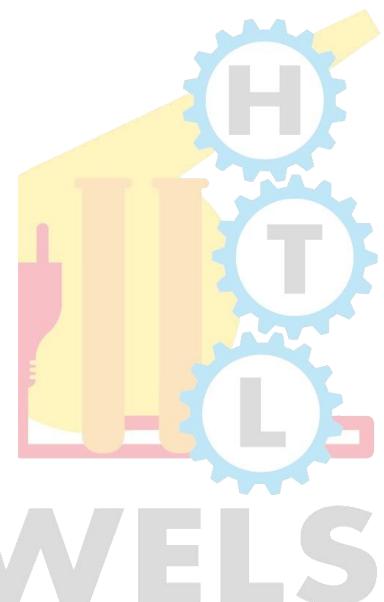
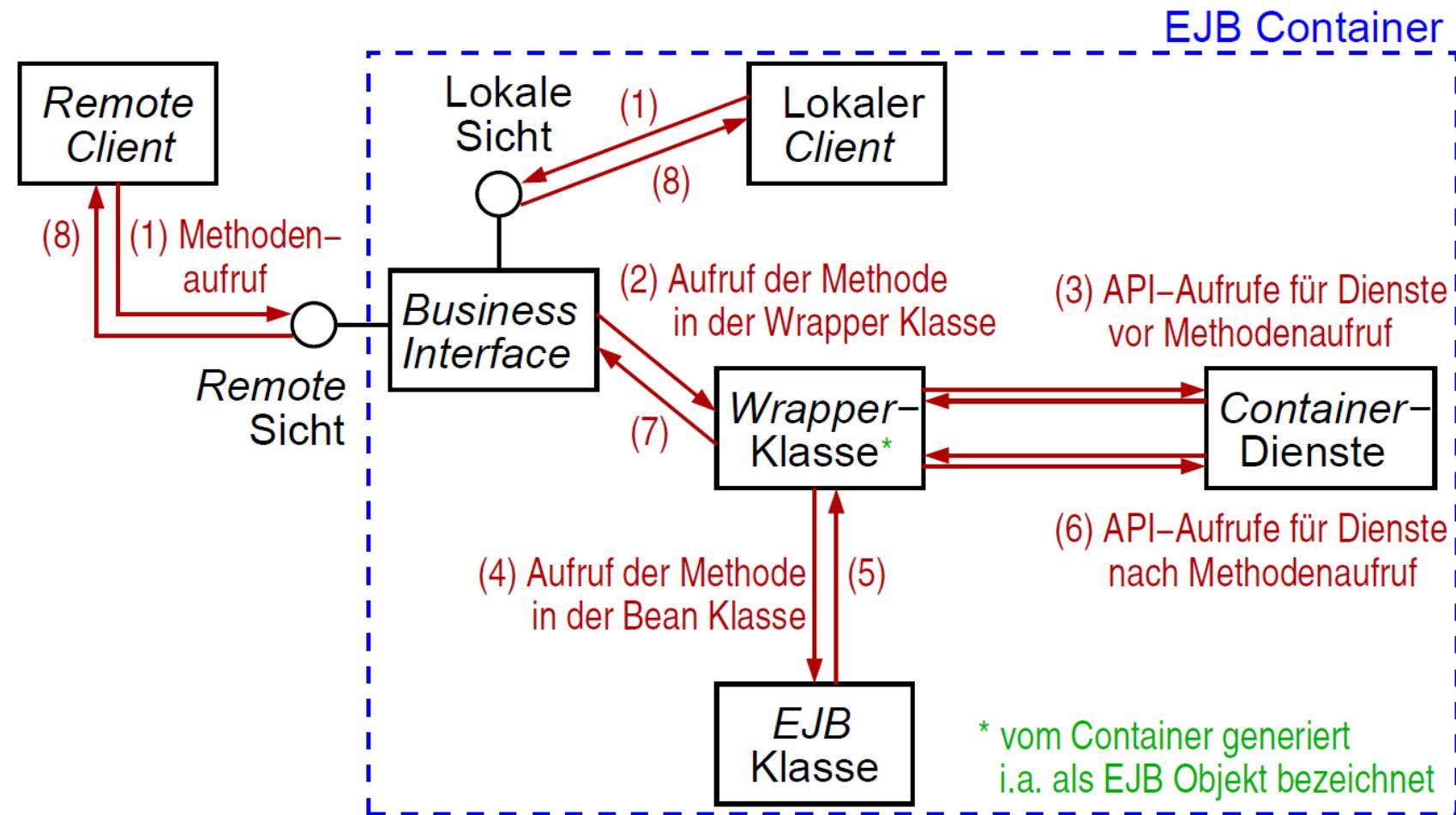
Architektur einer Enterprise Applikation



Grundlegende Elemente einer EJB-Umgebung

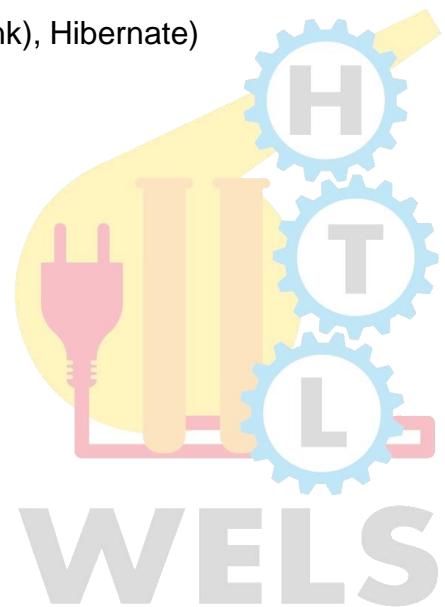


Das EJB 3.0 Programmiermodell



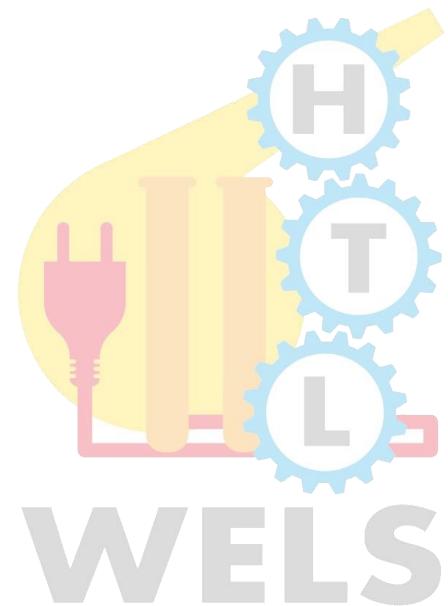
Mittel zur Vereinfachung von EJB 3.0

- **Annotationen**
 - erst ab Java 5 verfügbar
 - anstelle von XML
- **POJOs und POJIs**
 - leichtgewichtige Objekte ohne Zwang, irgendwelche Interfaces zu implementieren
- **Dependency Injection**
 - leichtgewichtige Verwaltung durch Container
- **Komplett neue Persistenz-Abbildung**
 - Es wird nur noch der Persistenz-Dienst spezifiziert (JPA), die Implementierung erfolgt durch Provider (Toplink (heute EclipseLink), Hibernate)



Prinzipien bei EJB 3.0/3.1

- **Deklaration statt Programmierung**
 - mittels Annotationen @Entity, @Stateless, ...
 - Inversion of Control
- **Hollywood-Prinzip**
 - "*Don't call us, we call you!*"
 - keine Aufrufe von Diensten in Fach-Objekt mehr wie bei Spring
- **Configuration by Exception**
 - nur in Ausnahmefällen ist Konfiguration notwendig
 - ansonsten reichen die Voreinstellungen



Enterprise Java Bean

```
public class Adresse {  
    ...  
}
```

POJO

+

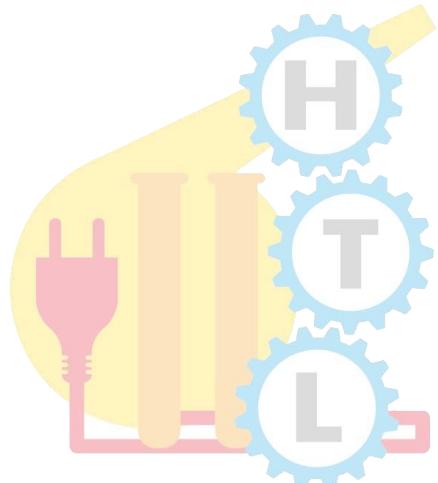


=

Annotation



EJB



WELS

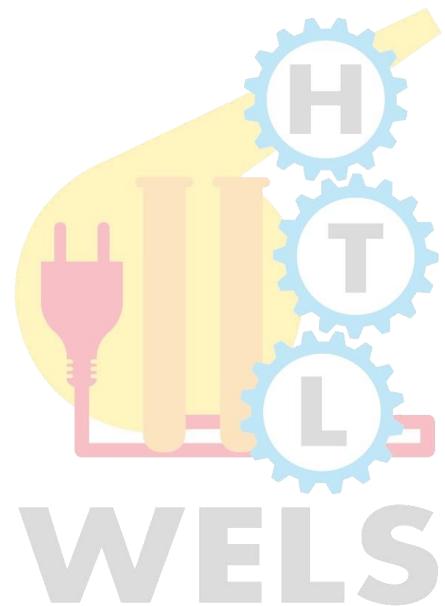
Arten von EJB's

▪ *Entity Beans*

- Daten-Objekte, die persistent (in Datenbank) gespeichert und von mehreren Clients genutzt werden d.h. Objekte bzw. Beziehungen der Anwendungsmodellierung z.B. Kunde, Konto, Aktie, ...
- in EJB 3.0 nicht weiterentwickelt, aber weiterhin unterstützt

▪ *Entities* (ab EJB 3.0, Java Persistence API)

- Ziele und Aufgaben wie bei *Entity Beans*, aber nur **lokal zugreifbar, leichtgewichtiger und besser standardisiert**
- Persistenz wird automatisch durch Container realisiert
- auch unabhängig von Java EE verwendbar



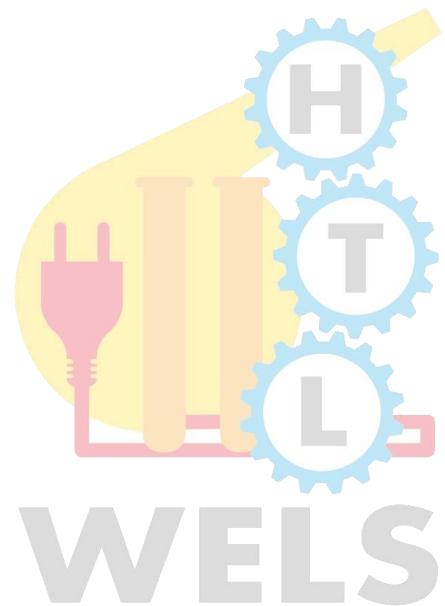
Arten von EJB's

▪ **Session Beans**

- realisieren Aktionen, die mehrere Anwendungsobjekte (*Entities* bzw. *Entity Beans*) betreffen, d.h. die Geschäftslogik
- zustandslos (**stateless**) oder zustandsbehaftet (**stateful**)
 - d.h. merkt sich die Bean Daten zwischen zwei Aufrufen eines Clients?
 - *stateful Session Beans* repräsentieren Client-Sitzungen
- kein **persistenter** Zustand
- In EJB 3.1 wurde der Status: Singleton eingeführt

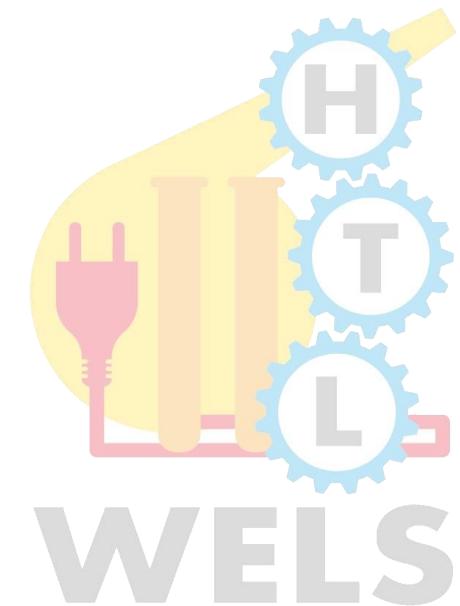
▪ **Message Driven Beans** (ab EJB 2.0)

- ähnlich wie *Session Beans*, aber asynchrone Schnittstelle
- Operationen werden über JMS-Nachrichten aufgerufen statt über (entfernte) Methodenaufrufe



Zusammenfassung: 3 Arten von Beans

- **Session Beans** (Enthalten Anwendungslogik)
 - Repräsentieren einen Client (in einer Session) auf dem Server können **stateful** oder **stateless** sein
 - EJB 2.1: implementieren **javax.ejb.SessionBean**
 - EJB 3.0/3.1: Annotation: @Stateful, @Stateless, @Singleton
- **Message Driven Beans** (Zur asynchronen Kommunikation)
 - Verarbeiten (einzelne) Nachrichten
 - EJB 2.1: implementieren **javax.ejb.MessageDrivenBean**
 - EJB 3.0/3.1: Annotation: @MessageDriven
- **Entity Beans** (Enthalten persistente Daten)
 - EJB 2.1: implementieren **javax.ejb.EntityBean**
 - EJB 3.0: Obsolet → durch Entities ersetzt

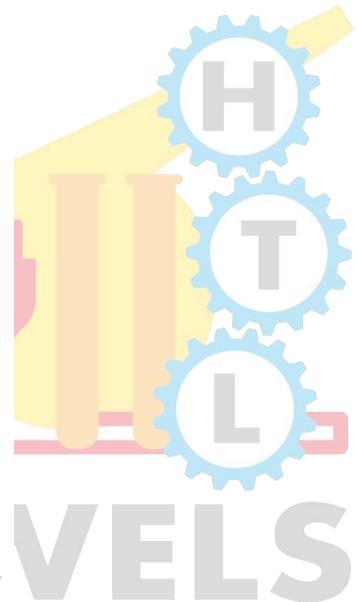


EJB 2.1 → Einfaches Session-Bean

```
import javax.ejb.SessionContext;

public class ComputeBean implements javax.ejb.SessionBean
{
    // EJB-required methods
    public void ejbCreate() { }
    public void ejbRemove() { }
    public void ejbActivate() { }
    public void ejbPassivate() { }
    public void setSessionContext(SessionContext ctx) { }

    // Business methods
    public int square(int x)
    {
        return x * x;
    }
}
```

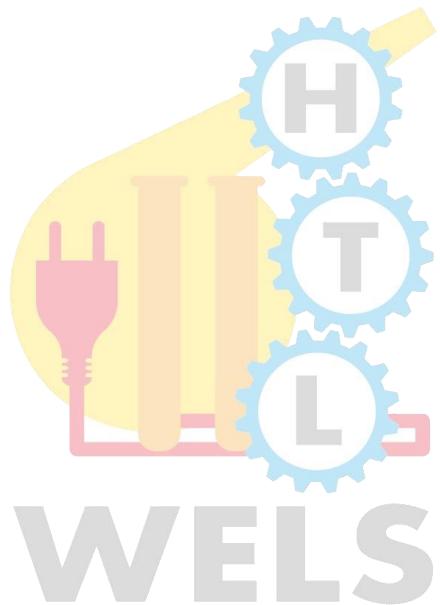


EJB 3.0 → Einfaches Session-Bean

```
@Stateless public class ComputeBean implements Compute {  
    // Business methods  
    public int square(int x)  
    { return x * x; }  
}
```

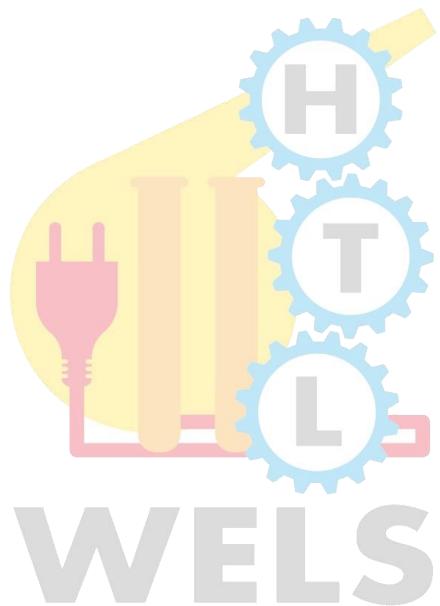
Nur noch ein (normales) Business Interface statt Remote Interface:

```
@Remote public interface Compute {  
    public int square(int x);  
}
```



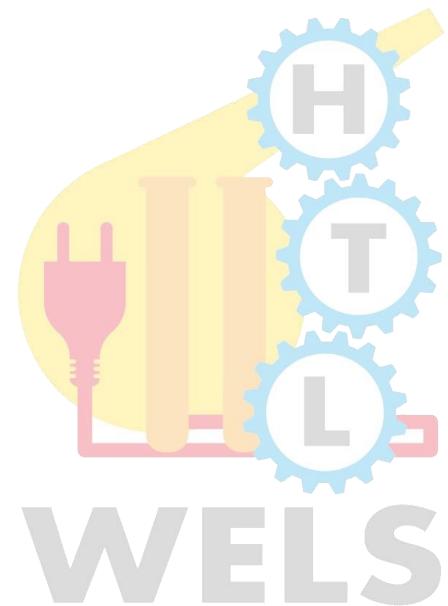
Anatomie einer EJB-Anwendung #1

- Eine EJB-Anwendung besteht aus:
 - **Session** (bzw. *Message Driven*) **Beans** für die **Anwendungslogik**
 - Session Beans sind normale Java-Klassen mit Annotationen
- **Local und Remote Business Interfaces** der **Session Beans**
 - normale Java-Interfaces, mit Annotationen
 - i.d.R. durch zugehörige **Session Bean** implementiert
- **Entities** (bzw. *Entity Beans*) für das **Datenmodell**
 - Entities sind normale Java-Klassen mit Annotationen
- Deployment-Deskriptoren
 - XML-Dokumente
 - beschreiben Beans und vom Container benötigte Dienste



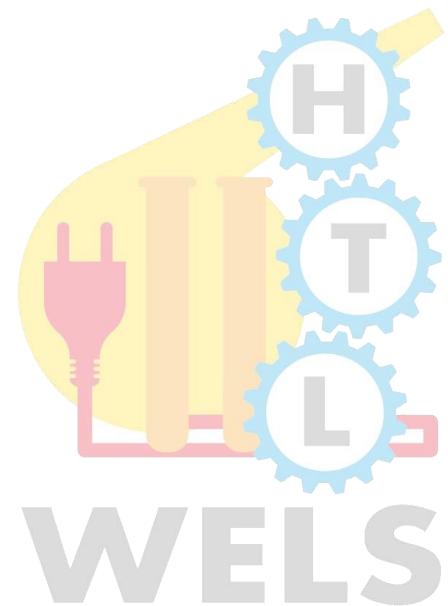
Anatomie einer EJB-Anwendung #1

- Mehrere Beans werden mit gemeinsamen Deployment-Deskriptoren in ein JAR-Archiv gepackt
 - Deployment-Deskriptoren im Verzeichnis /META-INF
 - META-INF/ejb-jar.xml kennzeichnet Archiv als EJB Archiv
- EJB-Anwendung besteht aus ein oder mehreren JAR-Archiven
- Alle weiteren benötigten Klassen werden zur Laufzeit generiert
 - z.B.: Wrapper -Klassen für EBJ-Klassen bzw Implementierungsklassen der Business Interfaces
 - Stub-Klassen für die Clients
(Basis: generische Proxy-Klasse java.lang.reflect.Proxy)



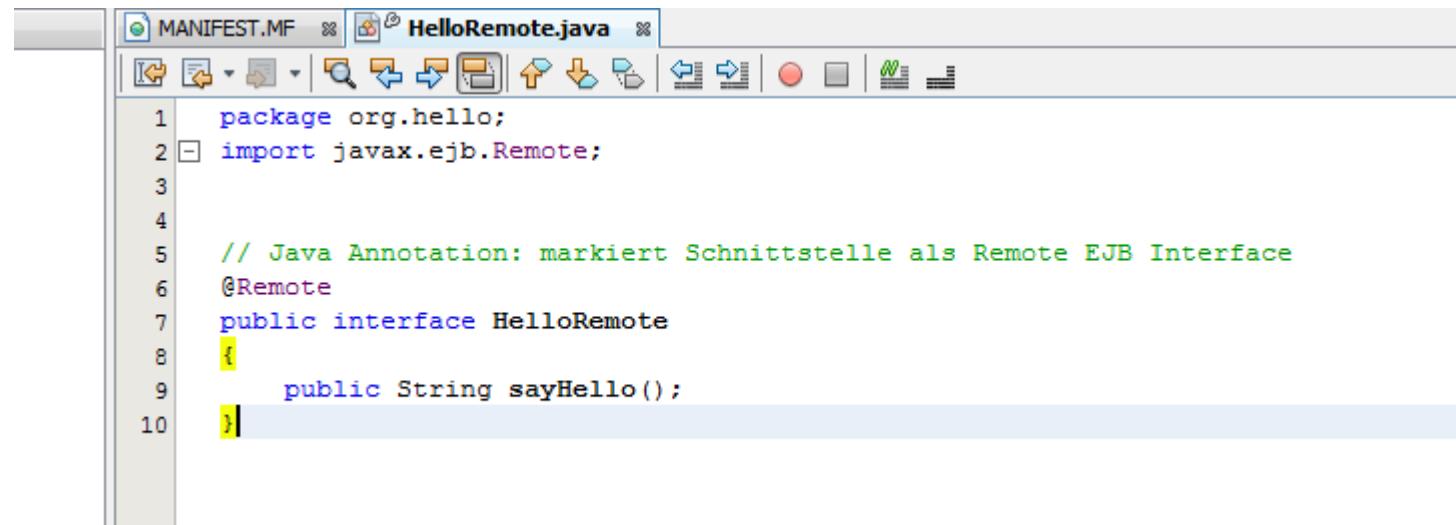
Deployment-Deskriptoren

- **ejb-jar.xml**: für *Session Beans* (und andere)
 - kann Annotationen in den EJB-Klassen überschreiben bzw. ergänzen
 - z.B. Kennzeichnung als EJB, Art der EJB, Verwaltung der EJB zur Laufzeit (Transaktionen, Zugriffskontrolle), ...
- Arbeitsaufteilung zwischen EJB-Entwickler (Annotationen) und Anwendungsentwickler (*Deployment-Deskriptor*)
- **persistence.xml**: für *Entities*
 - analog zu ejb-jar.xml
 - spezifiziert zusätzlich die Verbindung zur Datenbank
 - Zusätzlich Container-spezifische Deskriptoren möglich



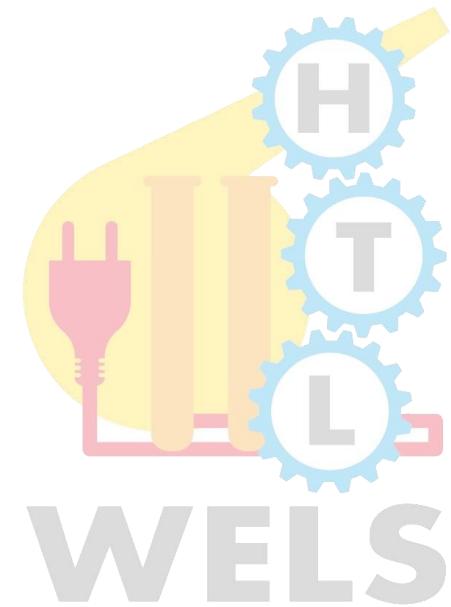
Beispiel Remote

- Remote Business Interface



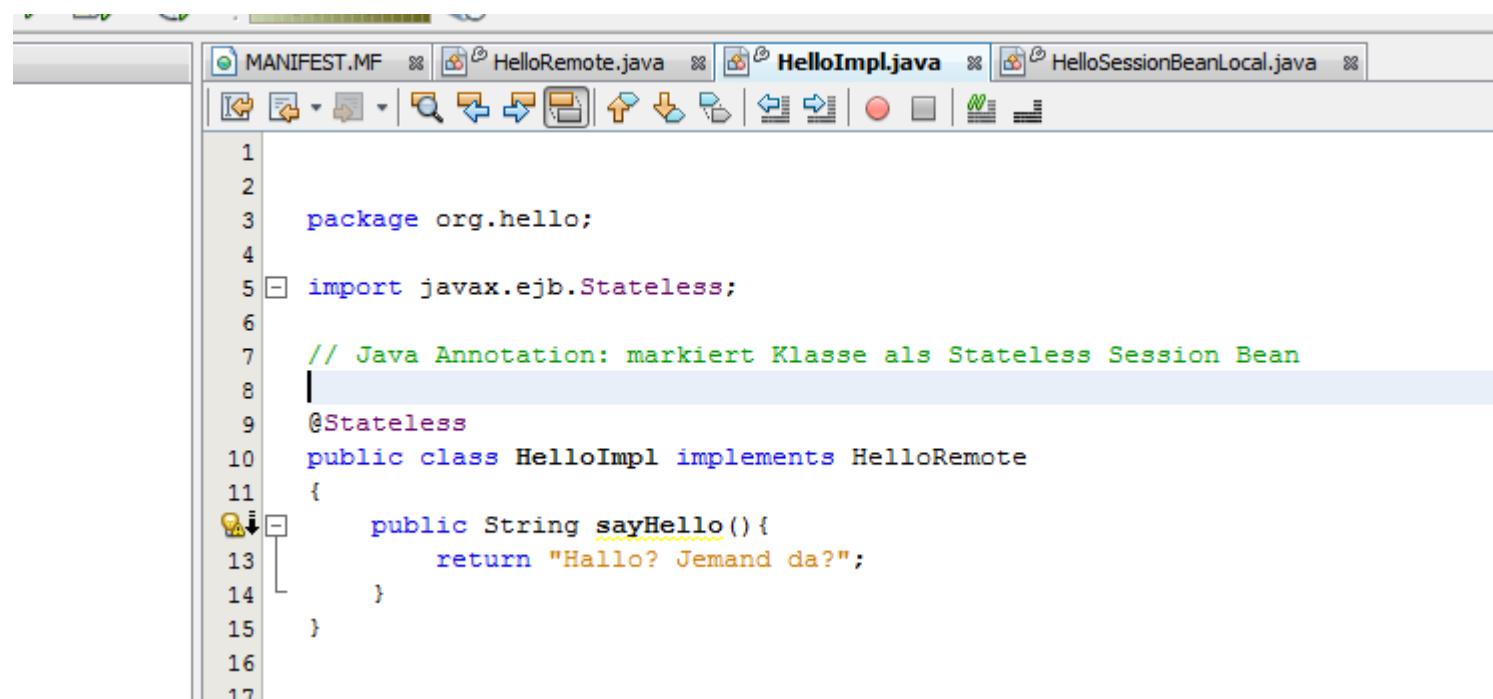
The screenshot shows a Java code editor with the file `HelloRemote.java` open. The code defines a remote business interface:

```
1 package org.hello;
2 import javax.ejb.Remote;
3
4 // Java Annotation: markiert Schnittstelle als Remote EJB Interface
5 @Remote
6 public interface HelloRemote
7 {
8     public String sayHello();
9 }
10 }
```



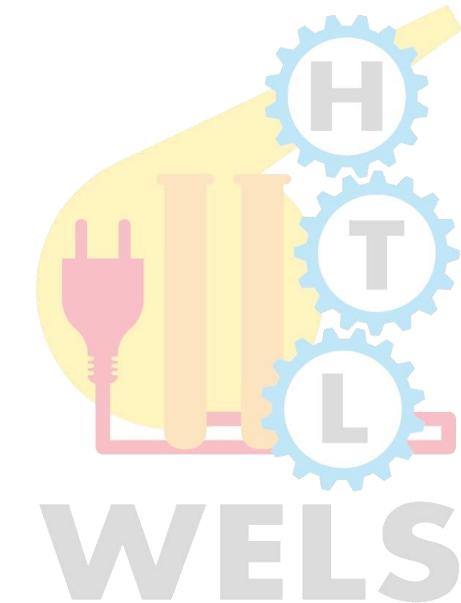
Beispiel Remote

- EJB Klasse



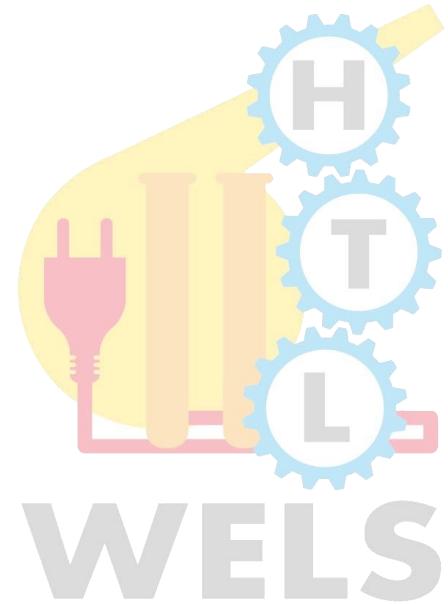
The screenshot shows a Java code editor with the file `HelloImpl.java` open. The code defines a stateless session bean named `HelloImpl` that implements the interface `HelloRemote`. The class contains a single method `sayHello()` which returns the string "Hallo? Jemand da?". The code editor has syntax highlighting and line numbers. The top menu bar shows tabs for `MANIFEST.MF`, `HelloRemote.java`, `HelloImpl.java` (which is the active tab), and `HelloSessionBeanLocal.java`.

```
1
2
3 package org.hello;
4
5 import javax.ejb.Stateless;
6
7 // Java Annotation: markiert Klasse als Stateless Session Bean
8
9 @Stateless
10 public class HelloImpl implements HelloRemote
11 {
12     public String sayHello(){
13         return "Hallo? Jemand da?";
14     }
15 }
16
17
```



Deployment Deskriptor (ejb-jar.xml)

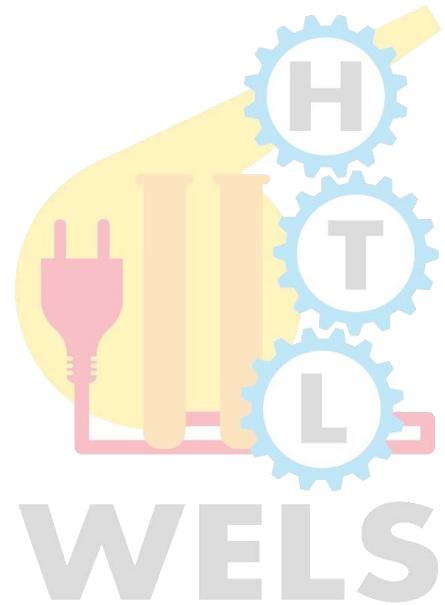
- <?xml version="1.0" encoding="UTF-8"?>
- <ejb-jar/>
- "Leerer" Deskriptor
- Zeigt lediglich an, dass es sich um eine EJB handelt



Dienste des EJB-Containers #1

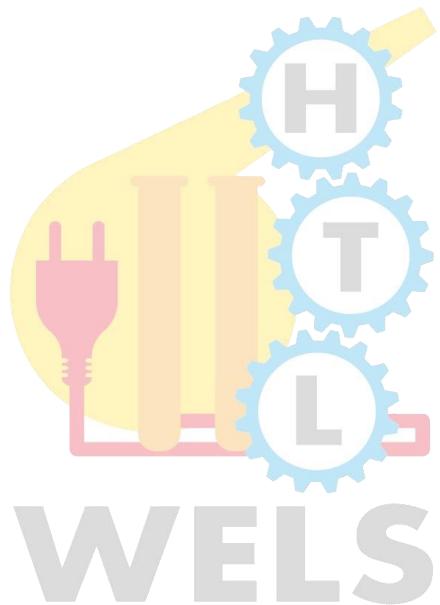
▪ Ressourcen-Management

- **Pooling:** Container hält einen Pool von Bean-Instanzen vor
 - z.B. bei stateless Session Beans: Aufrufe gehen an beliebige Instanz im Pool
 - Ziel u.a.: vermeide teuren Auf- und Abbau von Datenbank-Verbindungen
- **Passivierung und Aktivierung** von stateful Session Beans
 - Container kann Session Beans tempor "ar passivieren
 - z.B. wenn zuviele Bean-Instanzen vorhanden sind
 - bei Passivierung wird Zustand der Bean auf Festplatte gesichert und Bean-Instanz gelöscht
 - bei nächster Client-Anfrage: neue Bean-Instanz wird erzeugt und initialisiert



Dienste des EJB-Containers #2

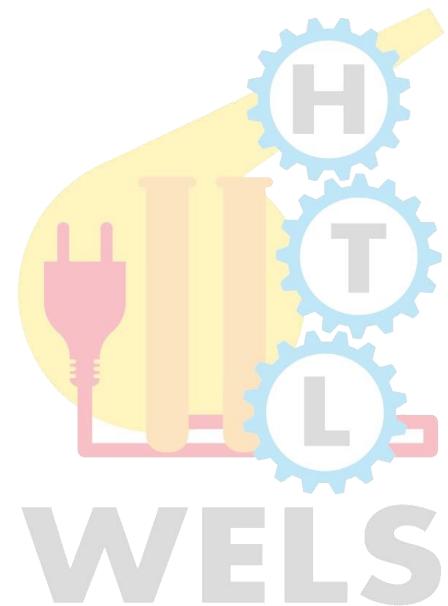
- **Namensdienst:** Zugriff über JNDI-Schnittstelle
- **Nebenläufigkeit**
 - Session Beans: immer nur von einem Client genutzt
 - Entities: optimistisches Locking
 - setzt Versions-Attribut in der Entity voraus
 - Alternative: explizite Read und Write-Locks
- **Transaktionen**
 - Methoden von Session Beans können automatisch als Transaktionen ausgeführt werden
- **Persistenz**
 - Datenfelder einer Entity werden automatisch mit dem Inhalt einer Datenbank synchronisiert



Dienste des EJB-Containers #3

▪ Sicherheit

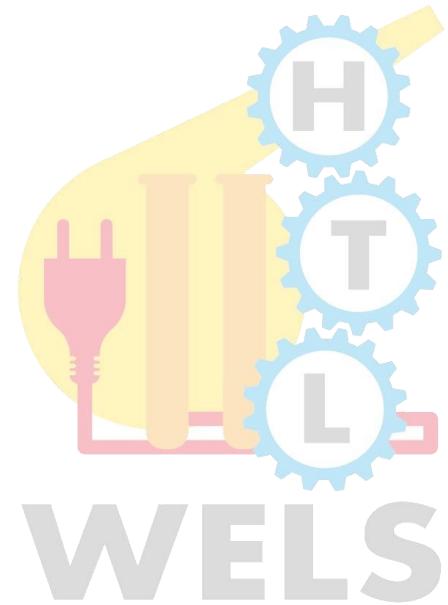
- Authentifizierung
 - Vorgehen abhängig von EJB-Container-Implementierung
 - oft: Benutzername / Passwort als Properties über JNDI übergeben
- Autorisierung
 - Festlegung, wer welche Methoden aufrufen darf über Deployment-Deskriptor oder Annotationen
- rollenbasierte Zugriffskontrolle
- sichere Kommunikation
 - abhängig von EJB-Container-Implementierung
 - meist: Nutzung von TLS/SSL



Session Beans – Details #1

▪ Stateless Session Beans

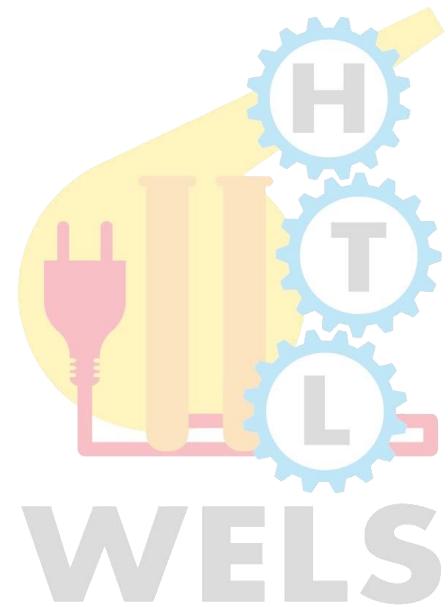
- Container verwaltet Pool **identischer Bean-Instanzen**
 - Erzeugung / Löschung nur durch Container
- Aufrufe von Anwendungsmethoden werden an beliebige *Bean*-Instanz im Pool geleitet
- keine Passivierung / Aktivierung



Session Beans – Details #2

▪ Stateful Session Beans

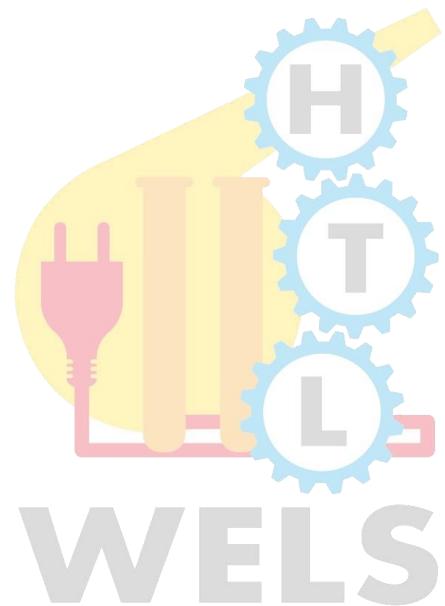
- Erzeugung / Löschung (indirekt) auf Veranlassung des Clients
- Bean-Instanzen werden Client- Sitzungen (und damit EJB-Objekten) fest zugeordnet
 - Aufrufe eines Clients gehen an dieselbe Bean-Instanz
- bei Bedarf: Passivierung / Aktivierung durch Container



Session Beans: Details #3

Annotationen für Schnittstellen und Bean-Implementierungen

- **@Local** und **@Remote** (aus javax.ejb)
 - markieren ein Java-Interface als lokale bzw. remote Schnittstelle einer EJB
 - lokale Schnittstelle kann nur innerhalb des Containers (von anderen EJBs) genutzt werden
 - Zuordnung zur Bean-Klasse über implements-Beziehung
- **@Stateless** und **@Stateful** (aus javax.ejb)
 - markieren eine Java-Klasse als entsprechende Session Bean
- **@Remove** (aus javax.ejb)
 - markiert eine Methode in einer (stateful) Session Bean-Klasse, nach deren Ausführung die Bean-Instanz gelöscht werden soll



Session Beans: Details #4

- Beispiele

→ RemIf.java:

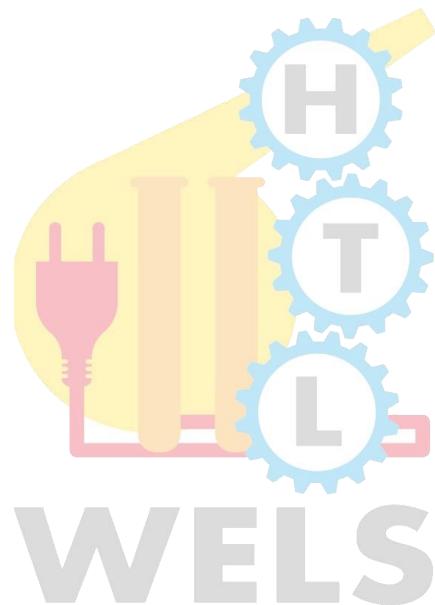
```
@Remote  
public interface RemIf {  
    public String sayHello();  
    public void bye();  
}
```

→ LocIf.java:

```
@Local  
public interface LocIf {  
    public String sayHello();  
    public void doIt();  
}
```

→ MyBean.java:

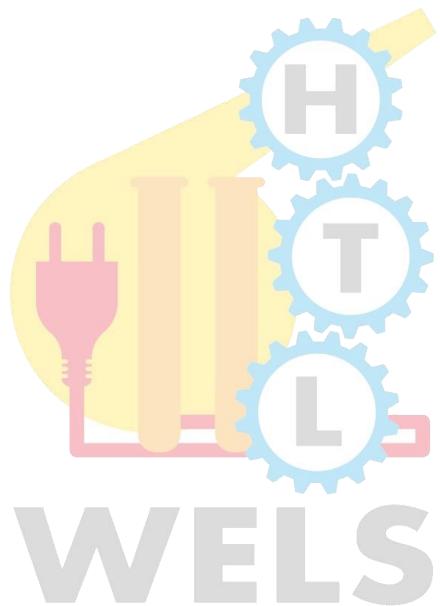
```
@Stateful  
public class MyBean implements LocIf, RemIf {  
    public String sayHello() { ... }  
    @Remove public void bye() { ... }  
    public void doIt() { ... }  
}
```



Session Beans: Details #5

Lebenszyklus-Callbacks

- **@PostConstruct** bzw. **@PreDestroy** (aus javax.annotation)
 - markiert Methode in einer *Bean*-Klasse, die nach Erzeugung bzw. vor Löschung einer *Bean*-Instanz aufgerufen werden soll
 - aber: keine Garantie, dass *Bean*-Instanz jemals gelöscht wird
- **@PrePassivate** bzw. **@PostActivate** (aus javax.ejb)
 - markiert Methode, die vor Passivierung bzw. nach Aktivierung einer *Bean*-Instanz aufgerufen werden soll
- Alle Callback-Methoden sollten wie folgt deklariert werden:
 - `public void name () { ... }`
- Auch möglich: Definition einer eigenen Klasse für die Callbacks

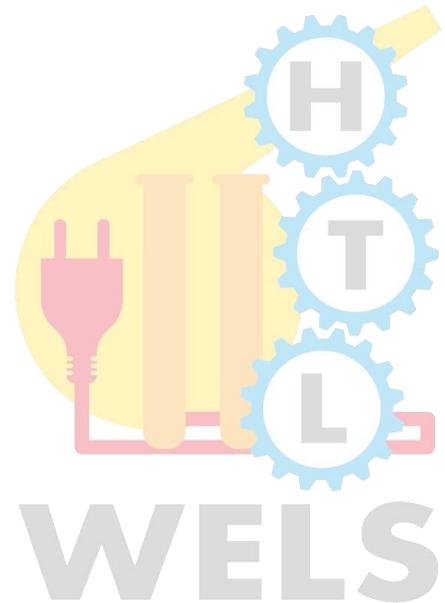


Session Beans: Details #6

Interceptor-Methoden

@AroundInvoke (aus javax.interceptor)

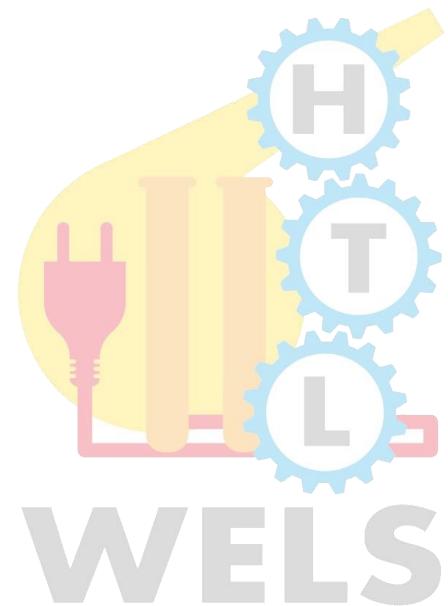
- markiert Methode, die alle Methodenaufrufe einer Bean abfängt
- Deklaration der Methode:
 - `public Object name (InvocationContext c) throws Exception { ... }`
- InvocationContext erlaubt u.a.:
 - Abfrage von Zielobjekt, Methodename und Parameter
 - Ausführung des abgefangenen Methodenaufrufs
- Einsatz z.B. Protokollierung, Zugriffskontrolle, ...
- Auch möglich: Definition einer eigenen Klasse für den *Interceptor*



Session Beans: Details #7

Dependency Injection

- Aufgabe: Beschaffung von Referenzen auf Ressourcen, die der Container bereitstellt
- Lösung: passendes Attribut wird mit einer Annotation versehen
 - Attribut wird dann automatisch vom Container initialisiert
- **@Resource** (aus javax.annotation)
 - für beliebige Ressourcen des Containers
 - z.B. SessionContext: erlaubt Zugriff auf Benutzer-Name, aktuelle Transaktion, ...
- **@EJB** (aus javax.ejb)
 - speziell, um Referenzen auf EJBs zu holen
 - (Alternative: explizite Nutzung von JNDI)



Session Beans: Details #8

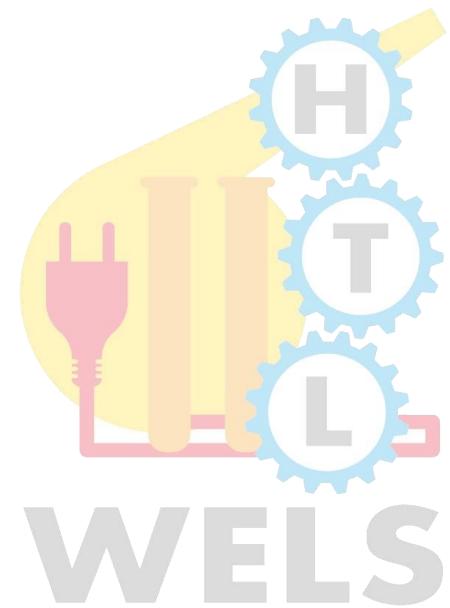
- Beispiele:

MyBeanLocal.java:

```
@Local public interface MyBeanLocal {  
    public String getName();  
}
```

MyBean.java:

```
@Stateful  
public class MyBean implements MyBeanLocal {  
    // Attribut wird vom Container initialisiert!  
    @Resource private SessionContext context;  
    public String getName() {  
        return context.getCallerPrincipal().getName();  
    }  
}
```



Session Beans: Details #9

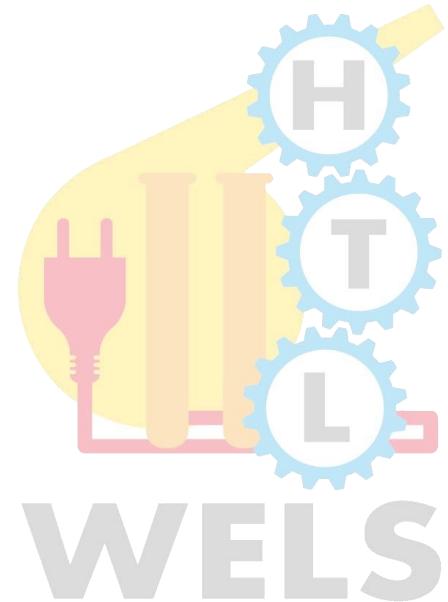
- Beispiele:

HelloRemote.java:

```
@Remote public interface HelloRemote {  
    public String sayHello();  
    public String sayGoodBye();  
}
```

HelloImpl.java:

```
@Stateful  
public class HelloImpl implements HelloRemote {  
    // Referenz auf MyBean wird vom Container eingetragen!  
    @EJB private MyBeanLocal myBean;  
    public String sayHello() {  
        return "Hallo " + myBean.getName();  
    }  
    @Remove public String sayGoodBye() { ... }
```



Session Beans: Details #10

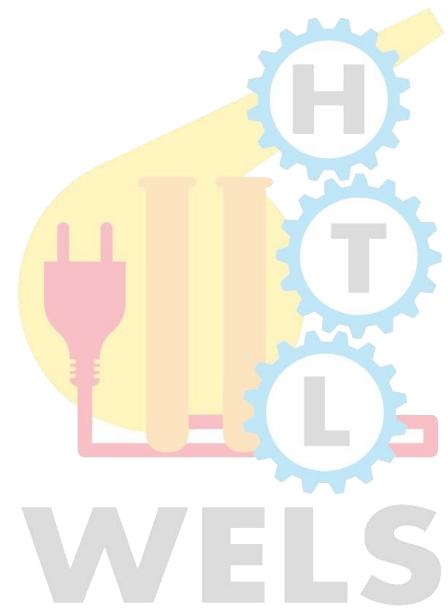
- Beispiele:

HelloImpl.java ...:

```
@PostConstruct public void start() {
    System.out.println("@PostConstruct HelloImpl");
}

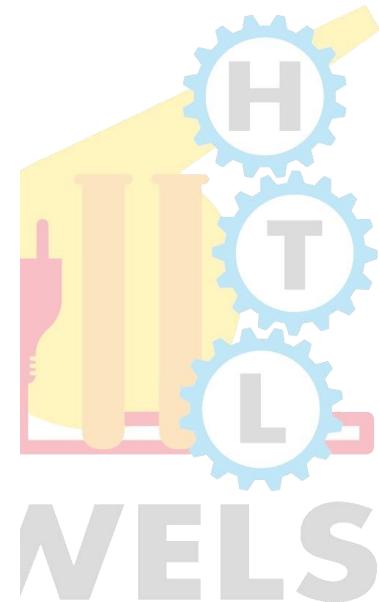
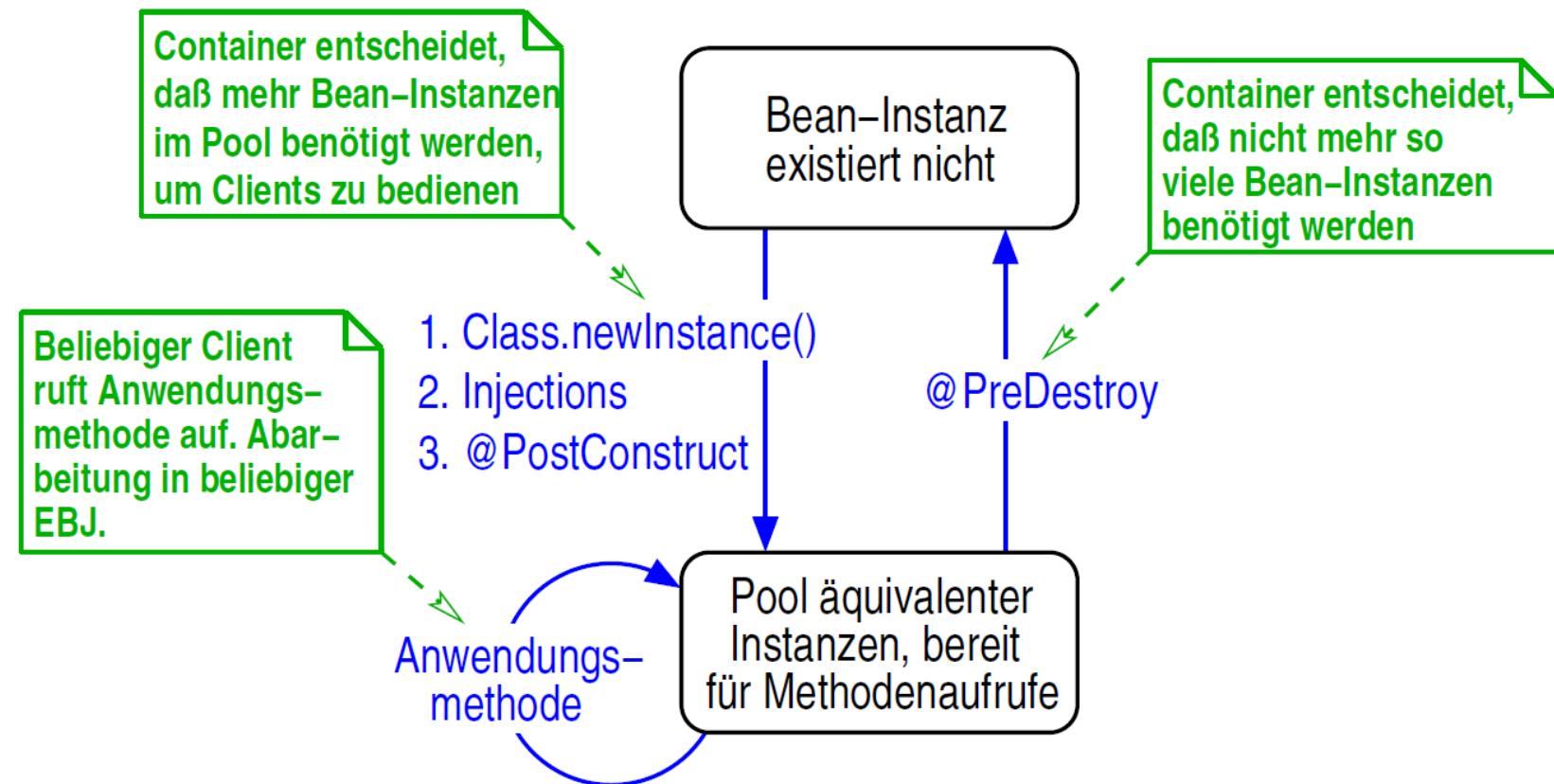
@PreDestroy public void stop() {
    System.out.println("@PreDestroy HelloImpl");
}

@AroundInvoke
public Object inv(InvocationContext c)
        throws Exception {
    System.out.println("HelloImpl: Calling "
        + c.getMethod().getName());
    return c.proceed();
}
```



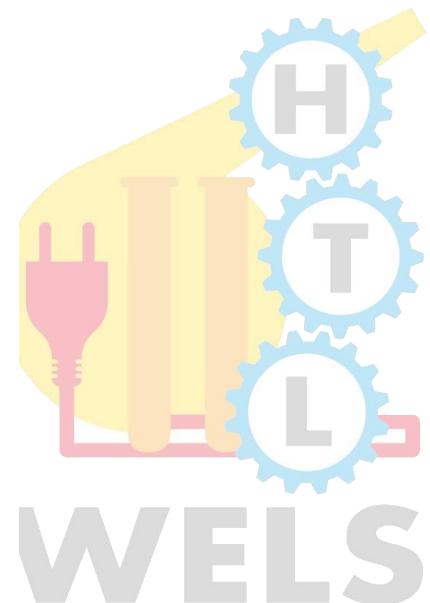
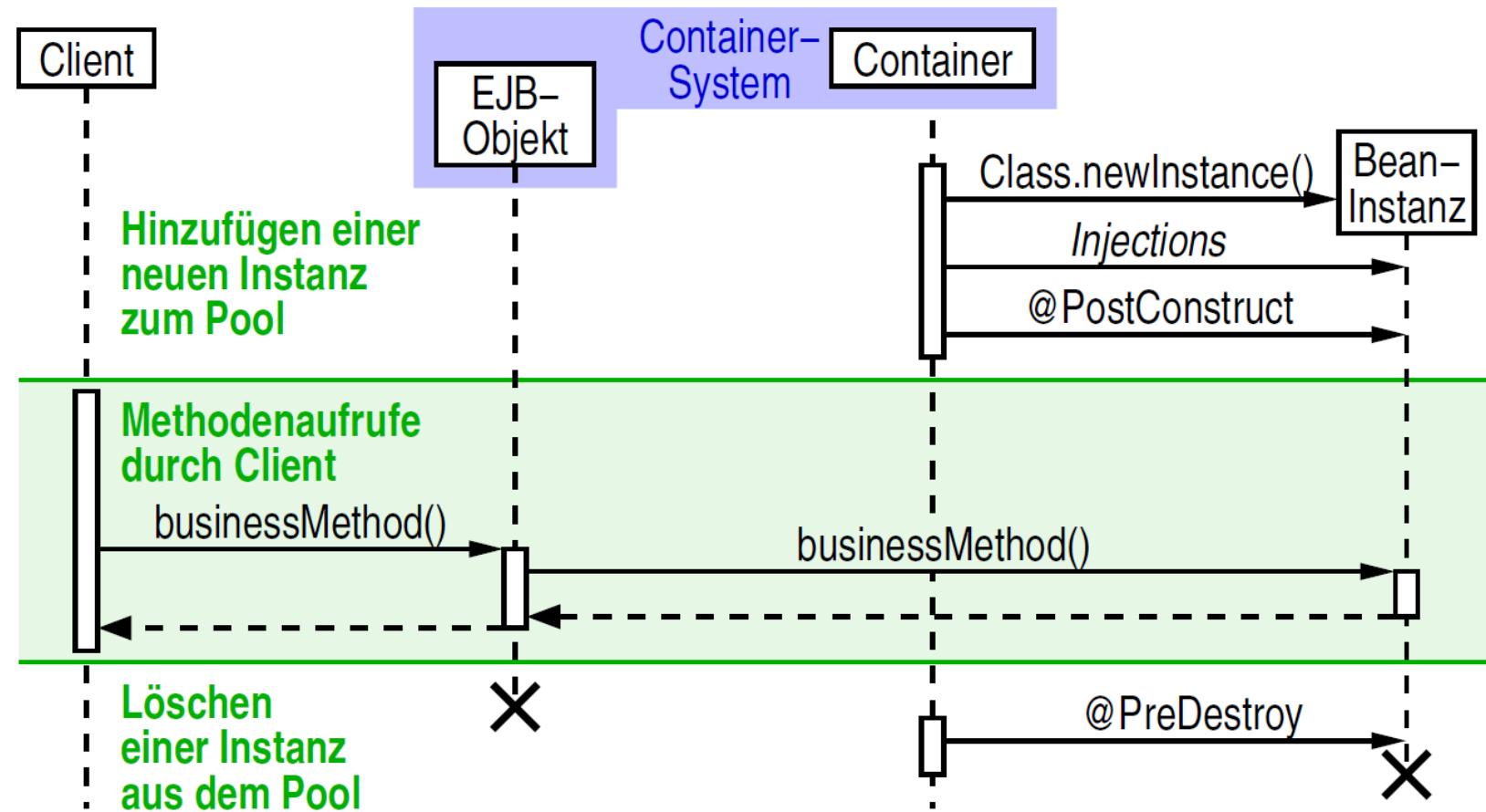
Session Beans: Details #11

Lebenszyklus einer *Stateless Session Bean*



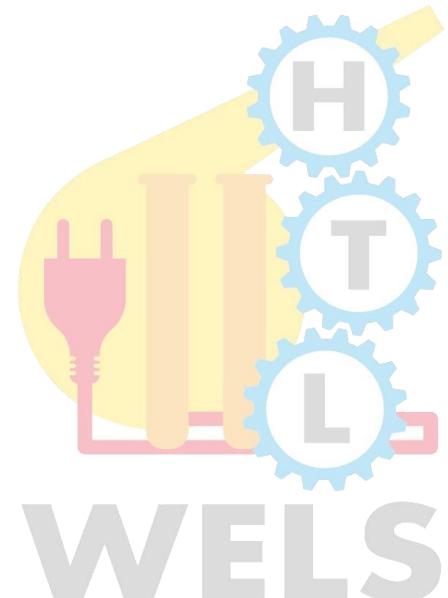
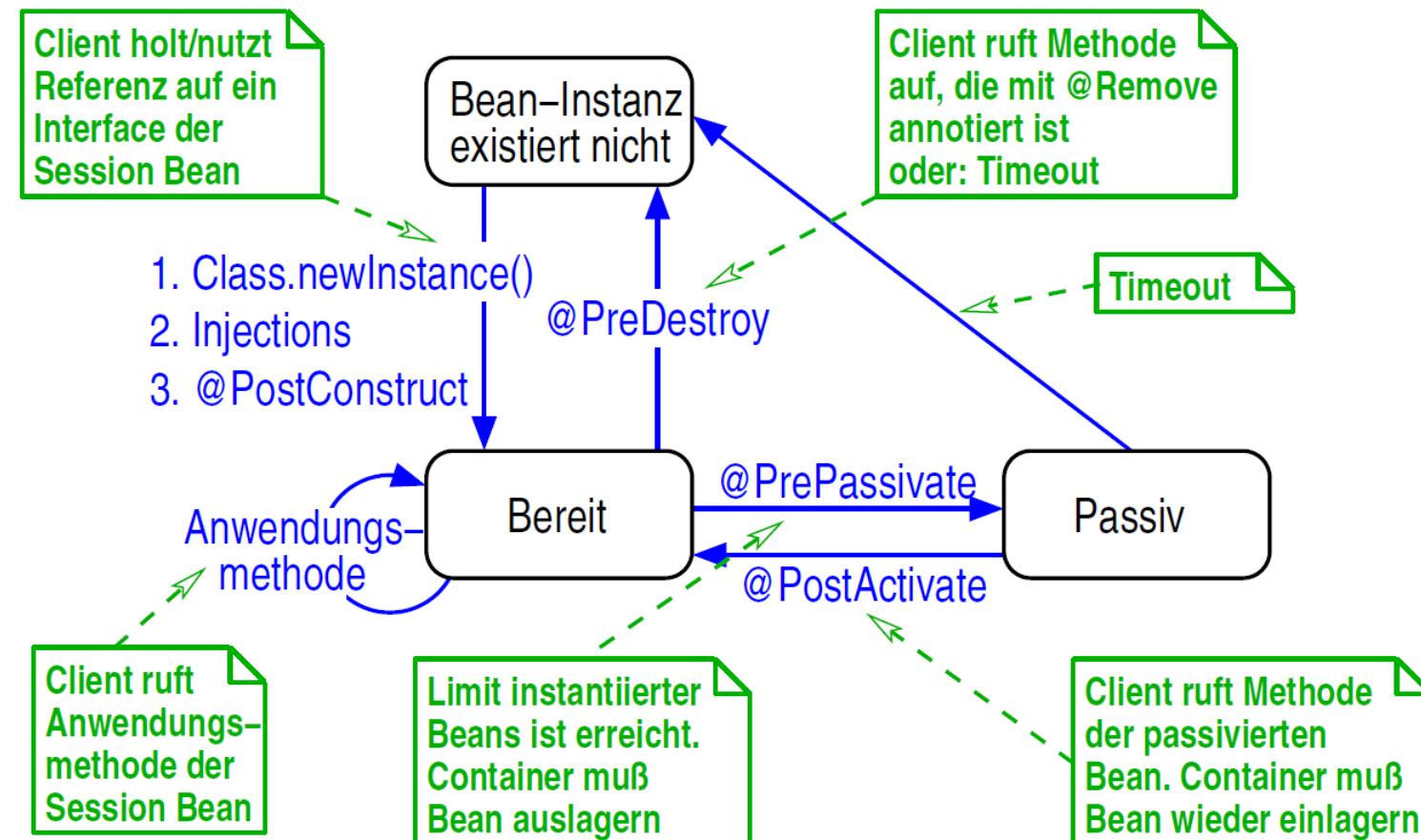
Session Beans: Details #12

Typische Abläufe bei *Stateless Session Beans*



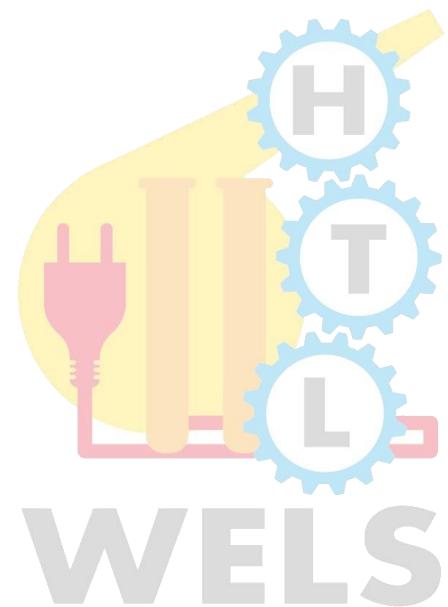
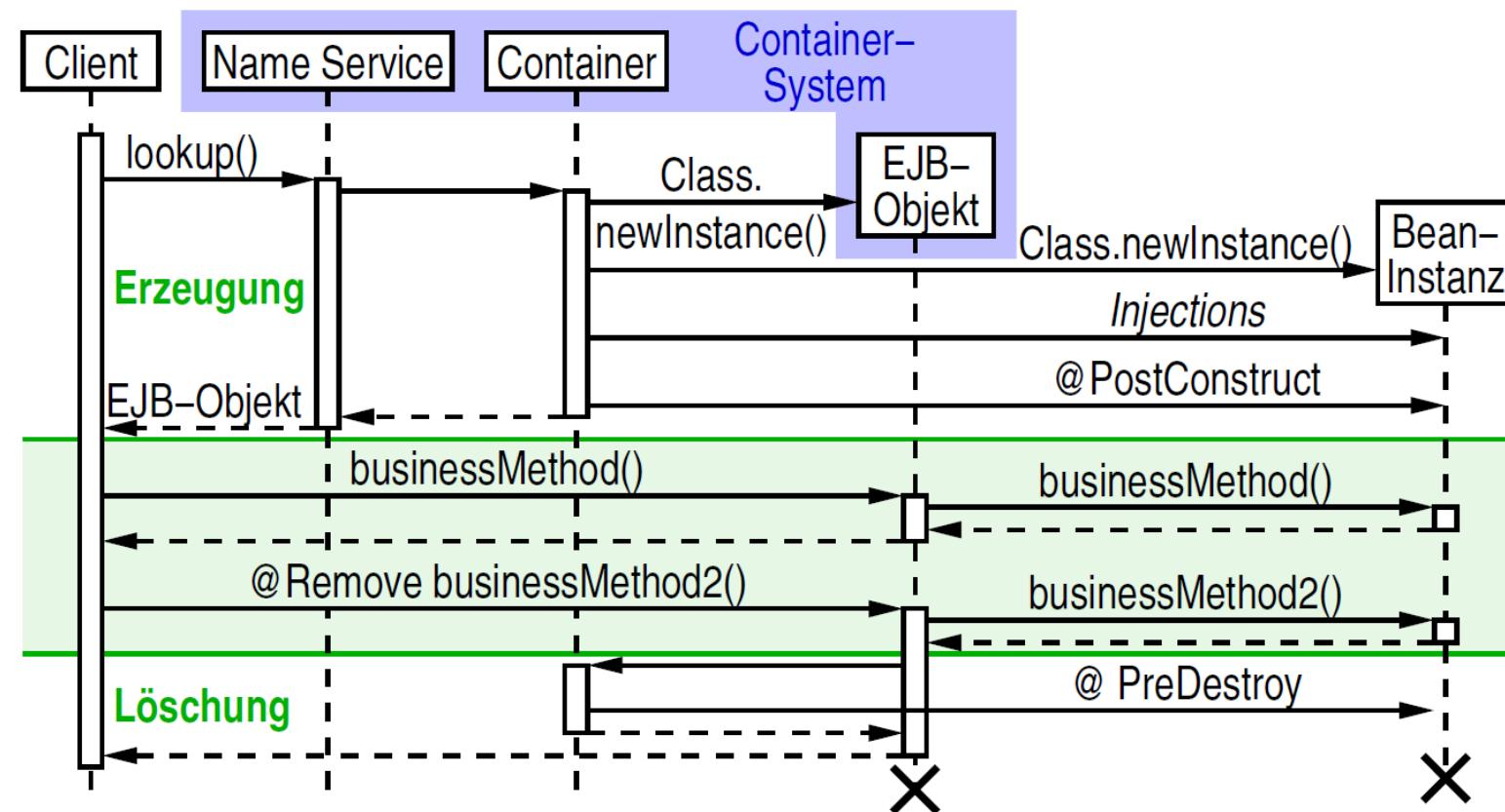
Session Beans: Details #13

Lebenszyklus einer Stateful Session Bean



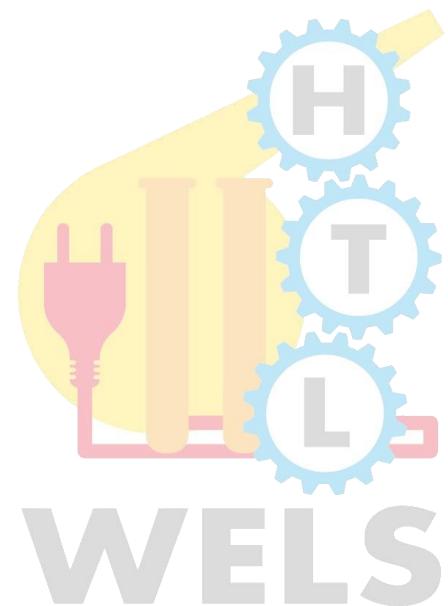
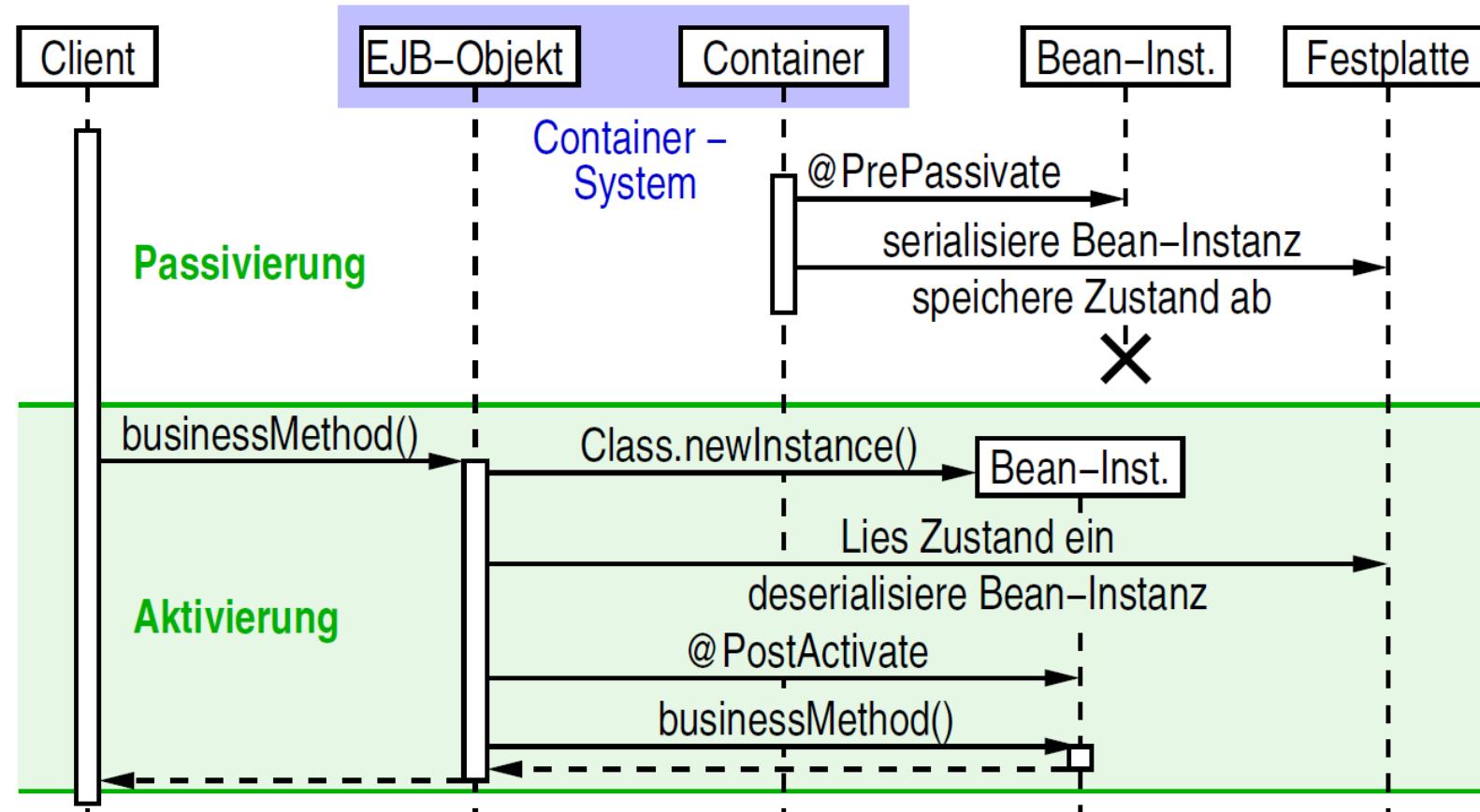
Session Beans: Details #14

Erzeugung/Löschung einer *Stateful Session Bean*



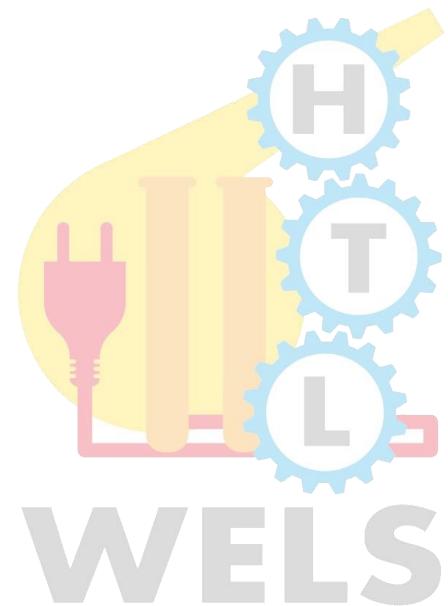
Session Beans: Details #15

Passivierung/Aktivierung einer *Stateful Session Bean*



Entities: Details #1

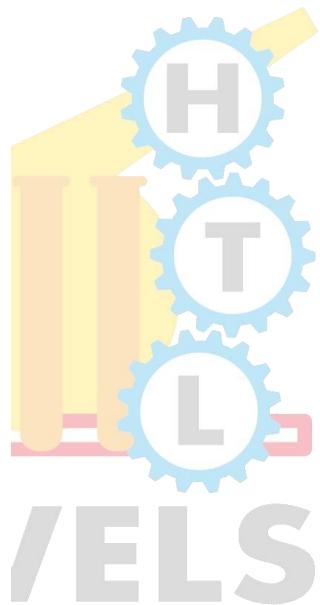
- Entities realisieren persistente Datenobjekte einer Anwendung
- Basis: **Java Persistence API** (JPA)
 - unabhängig von EJB und Java EE nutzbar
- **Eigenschaften** (Abgrenzung zu Session Beans):
 - für den Client sichtbare, persistente Identität (Primärschlüssel)
 - unabhängig von Objektreferenz
 - persistenter, für Client sichtbarer Zustand
 - nicht entfernt zugreifbar
 - Lebensdauer völlig unabhängig von der Anwendung
- Persistenz der Entities wird automatisch durch **Persistence Provider** gesichert



Entities: Details #2

Entity Account.java:

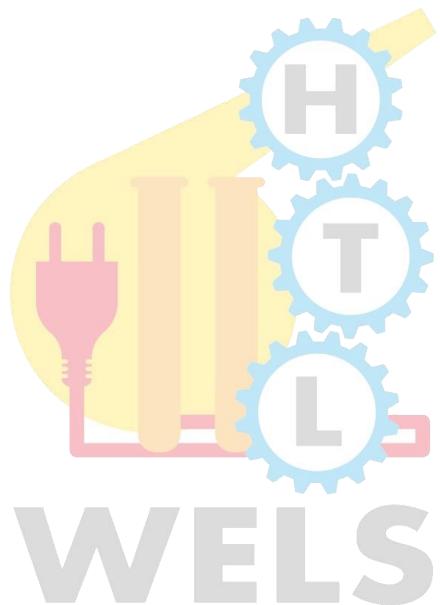
```
import javax.persistence.*;  
  
@Entity // Markiert Klasse als Entity  
public class Account implements java.io.Serializable {  
  
    @Id // Markiert Attribut als Primärschlüssel  
    private int accountNo;  
    private String name;  
  
    public int getAccountNo() { return accountNo; }  
    public String getName() { return name; }  
    public void setName(String nm) { name = nm; }  
    public Account(int no, String nm) {  
        accountNo = no; name = nm;  
    }  
}
```



Entities: Details #3

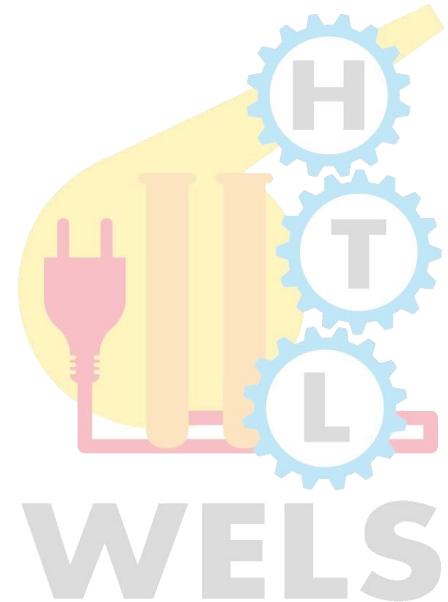
Deployment-Deskriptor META-INF/persistence.xml:

```
<persistence
    xmlns="http://java.sun.com/xml/ns/persistence"
    version="1.0">
    <persistence-unit name="intro">
        <jta-data-source>My DataSource</jta-data-source>
        <non-jta-data-source>My Unmanaged DataSource
        </non-jta-data-source>
        <class>org.Hello.Account</class>
        <properties>
            <property name="openjpa.jdbc.SynchronizeMappings"
                value="buildSchema(ForeignKeys=true)"/>
        </properties>
    </persistence-unit>
</persistence>
```



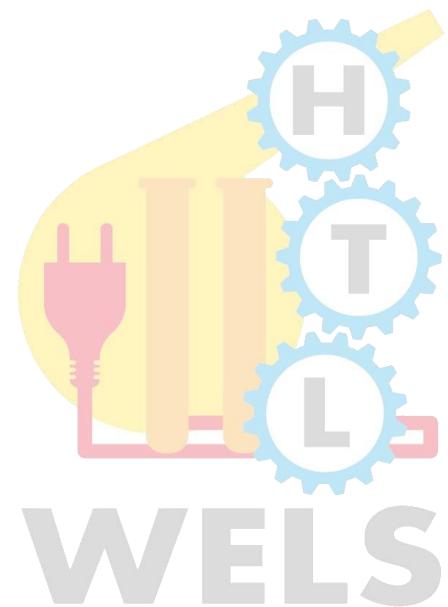
Entities: Details #4

- Eine Entity-Klasse muss **Serializable** nicht implementieren
 - falls Sie es tut, können Objekte auch als Parameter / Ergebnis von Remote-Methoden einer Session Bean auftreten
 - übergeben wird dabei eine Kopie, die **nicht** mit der Datenbank synchronisiert wird
- Abbildung von Klasse auf Datenbank-Tabelle und von Attributen auf Spalten wird vom Persistence Provider vorgenommen
 - kann durch **Annotationen** genau gesteuert werden
- Entity-Klasse muss ein Primärschlüssel-Attribut deklarieren (**@Id**)
- **Primärschlüssel** kann auch eine eigene Klasse sein
- Entity-Klasse darf auch Geschäftsmethoden besitzen



Entities: Details #5

- **Field Access**
 - Persistence Provider greift direkt auf die Attribute zu
 - Mapping-Annotationen (hier: @Id) bei den Attributen
- Alternative: **Property Access**
- Persistence Provider greift auf den Zustand nur über **Get- und Set-Methoden** zu
 - Mapping-Annotationen bei den Get-Methoden
 - Achtung: es müssen immer Get- **und** Set-Methoden implementiert werden
- Pro Entity ist nur **eine der Alternativen** erlaubt



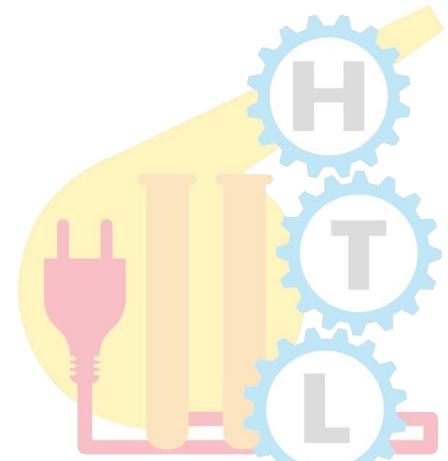
Entities: Details #6

- Beispiel zur Nutzung der Entity in einer Session Bean

Remote-Schnittstelle BankRemote.java:

```
import javax.ejb.Remote;

@Remote
public interface BankRemote
{
    public Account create(int n, String name);
    public String getName(int n);
    public void close(int n);
}
```



Entities: Details #7

- Beispiel zur Nutzung der Entity in einer Session Bean ...

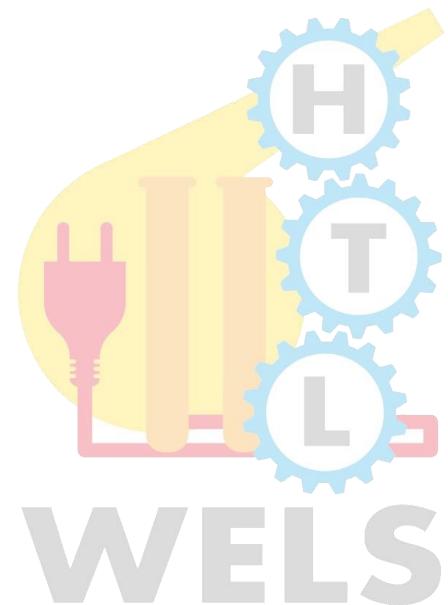
Bean-Implementierung BankImpl.java:

```
import javax.ejb.*;
import javax.persistence.*;

@Stateless
public class BankImpl implements BankRemote {

    // Dependency Injection: Entity Manager für Persistenz-Einheit 'intro'
    @PersistenceContext(unitName="intro")
    private EntityManager manager;

    public Account create(int n, String name) {
        Account acc = new Account(n, name);
        // Objekt ab jetzt durch Persistence Provider verwalten
        manager.persist(acc);
```

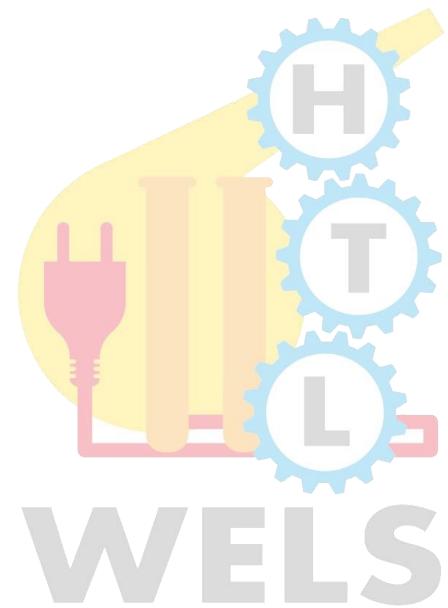


Entities: Details #8

- Beispiel zur Nutzung der Entity in einer Session Bean ...

Bean-Implementierung BankImpl.java:

```
// Rückgabewert ist eine losgelöste Kopie des Objekts!
    return acc;
}
public String getName(int n) {
    // Findet Objekt mit gegebenem Primärschlüssel
    Account acc = manager.find(Account.class, n);
    return acc.getName();
}
public void close(int n) {
    Account acc = manager.find(Account.class, n);
    // Datenbank-Eintrag löschen
    manager.remove(acc);
}
```



Entities: Details #9

- Beispiel zur Nutzung der Entity in einer Session Bean ...

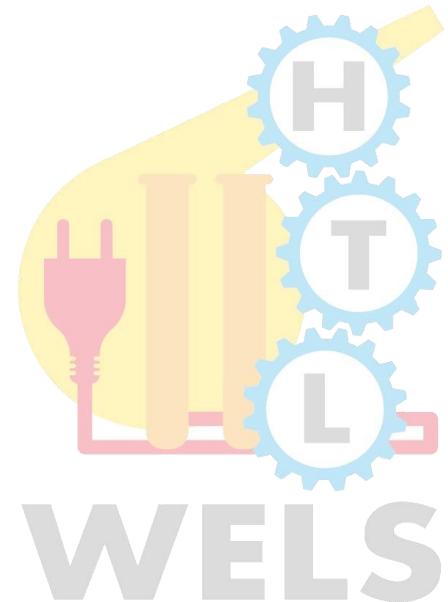
Client BankClient.java:

```
...
Object obj = ctx.lookup("BankImplRemote");
BankRemote bank = (BankRemote) obj;

// Erzeugt neue Entity (und Datenbankeintrag)
Account acc = bank.create(n, args[1]);

// acc ist eine Kopie des Eintrags, lokale Methodenaufrufe
System.out.println(acc.getName());
acc.setName("Niemand");

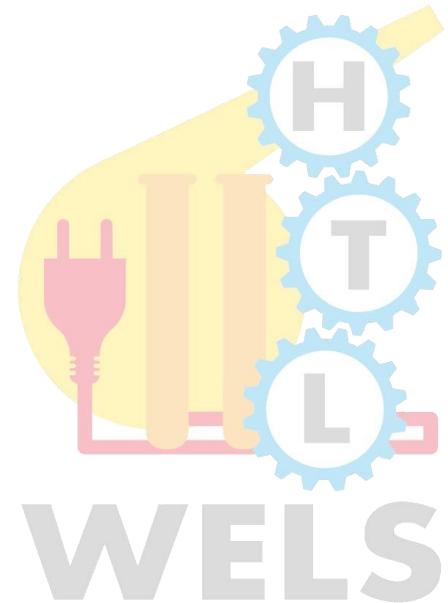
// Remote-Aufrufe der Entity Bean
System.out.println(bank.getName(n));
bank.close(n);
```



Entities: Details #10

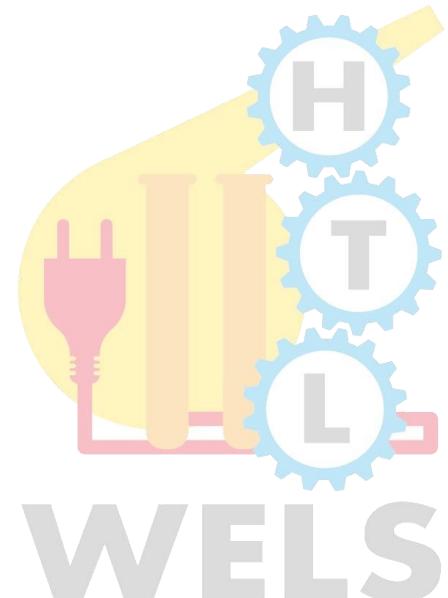
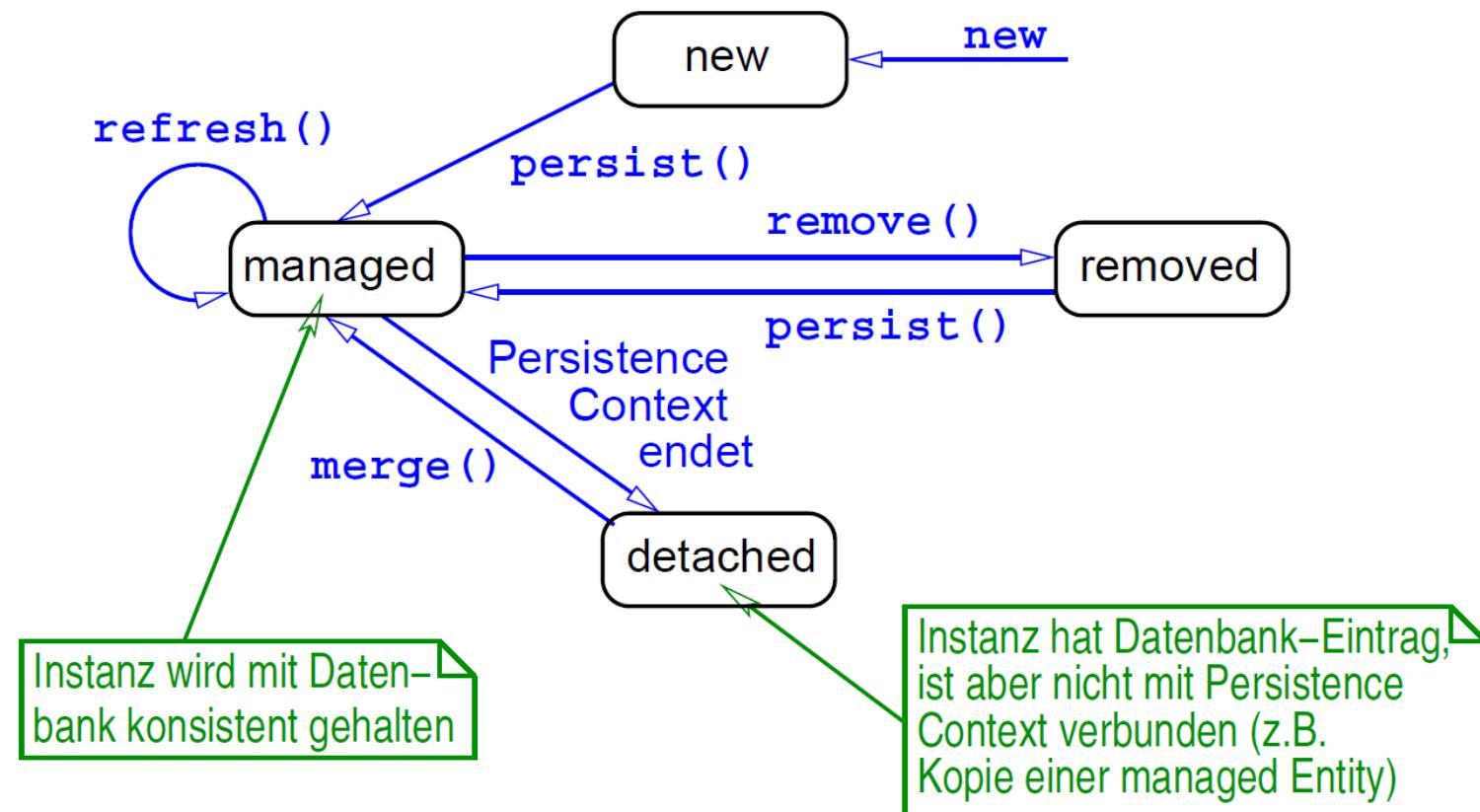
▪ Persistence Context

- Verbindung zwischen Instanzen im Speicher und der Datenbank
- Methoden der Schnittstelle **EntityManager** u.a.:
 - **void persist(Object entity)**
 - Instanz verwalten und persistent machen
 - **<T> T find(Class<T> entityClass, Object primaryKey)**
 - Instanz zu gegebenem Primärschlüssel suchen
 - **void remove(Object entity)**
 - Instanz aus der Datenbank löschen
 - **void refresh(Object entity)**
 - Instanz aus Datenbank neu laden
 - **<T> T merge(T entity)**
 - Zustand der Instanz in *Persistence Context* hineinmischen



Entities: Details #11

Lebenszyklus einer Entity



Entities: Details #12 - Objektzustand

- **new**

Objekt ist neu erzeugt, hat noch keinen Zusammenhang mit der Datenbank und noch keine gültige ID.

- **managed**

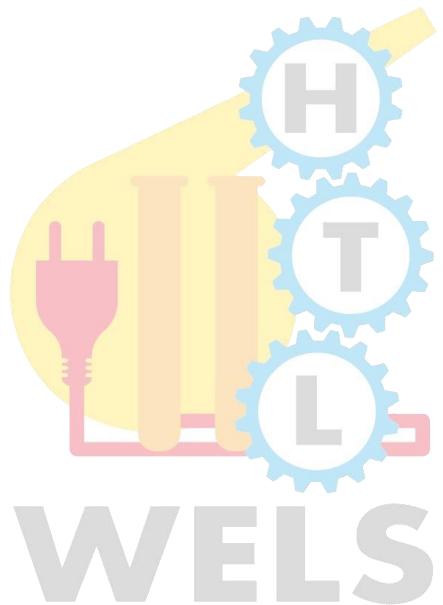
Das Objekt hat eine Entsprechung in der Datenbank. Änderungen werden vom Entity Manager automatisch getracked und mit der DB abgeglichen.

- **detached**

Das Objekt hat eine Entsprechung in der Datenbank, wurde aber abgekoppelt. Der Zustand wird nicht mehr automatisch abgeglichen mit der Datenbank.

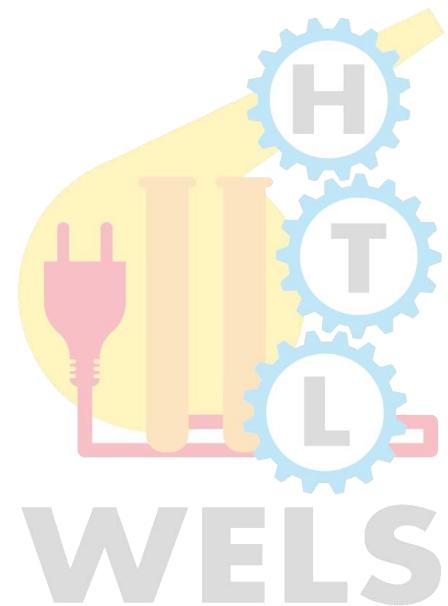
- **removed**

Das Objekt existiert noch, ist aber zum Löschen markiert.



Entities: Details #13

- Lebenszyklus einer Entity ...
 - Persistence Context endet per Voreinstellung mit dem Ende der aktuellen Transaktion
 - Einstellung über Attribut `type` von `@PersistenceContext`
- Synchronisation mit Datenbank i.a. am Ende jeder Transaktion
 - einstellbar über `setFlushMode()` Methode von `EntityManager`
 - ggf. auch explizite Synchronisation durch Methode `flush()`
- JPA verwendet standardmäßig ein **optimistisches Sperrprotokoll**
 - Datensätze werden nicht gesperrt
 - bei gleichzeitigen Änderungen durch zwei Transaktionen wird eine davon zurückgesetzt
 - dazu notwendig: Versionsattribut (Annotation `@Version`)
- Lebenszyklus-Callbacks analog zu Session Beans möglich



Schritte zur Erzeugung von EJB

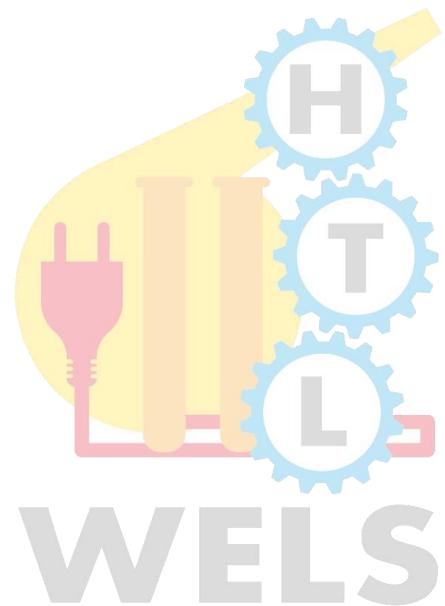
- 1. Schreibe eine **Enterprise Bean** (oder mehrere)
- 2. Beschreibe Schnittstellen für Clients
 - EJB 2.1: Remote (Component) Interface, und (Remote) Home Interface
 - **EJB 3.0/3.1:** (Remote) Business Interface
- 3. EJB 2.1: Schreibe eine XML-Beschreibungsdatei: deployment descriptor
EJB 3.0/3.1: Füge Annotationen hinzu
- 4. Packe alles in ein Jar-Archiv (EJB module).
- 5. Bringe das EJB Modul auf einen EJB Server, der den Java Container zur Verfügung stellt.

Schritte 3-5 können durch ein Deployment Tool teil-automatisiert werden. Das Deployment Tool kann in die IDE integriert sein.



Enterprise Java Beans 3.1 Neuerungen

- **Behutsame Erweiterung** von EJB 3.0 um folgende Aspekte
 - Vereinfachung der Entwicklung
 - Vereinfachte Local View auf EJB Session Beans
 - EJB Deployment in Web Archiven (.war Files)
 - “EJB lite” - Subset von EJB für Web Profile
 - Global JNDI Names
 - Container Bootstrapping API für Unit Testing
- **Neue Funktionalität**
 - Singleton Session Beans
 - EJB Timer Erweiterungen: Automatische Erzeugung von Timern und Calender Expressions
 - Asynchrone Aufrufe



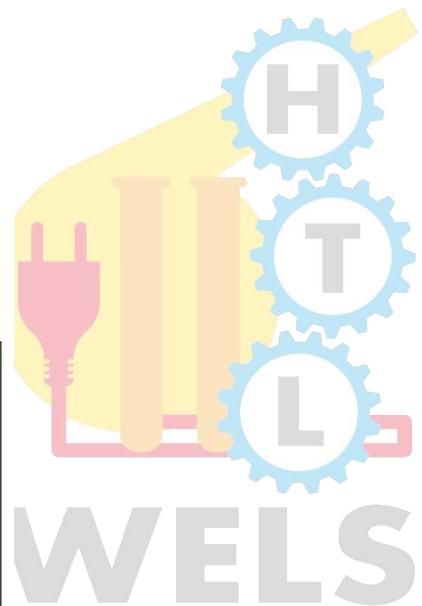
EJB 3.1 – “No Interface Local View”

- EJB 3.0 sieht für EJB Komponenten ein Business Interface vor
 - EJB 3.1 vereinfacht die Verwendung von Local EJBs per No Interface Local Views
 - Nur public Methoden der Bean exponiert

```
@Local
public interface BarBeanLocal {
    void foo();
}
@Stateless
public class BarBean
    implements BarBeanLocal {
    public void foo(){...}
}
```

EJB 3.1 no-Interface

```
@Stateless
public class BarBean
    public void foo(){...}
}
```



EJB 3.1 – Vereinfachtes Deployment

- Co-Packaging von Beans in Web Archiven vereinfacht Deployment
 - Single Naming & Classloader Environment

Pre-Java EE 6

MyEar.ear
/META-INF/application.xml

MyEJB.jar
/META-INF/ejb-jar.xml
/com/xyz/MyEJB.class

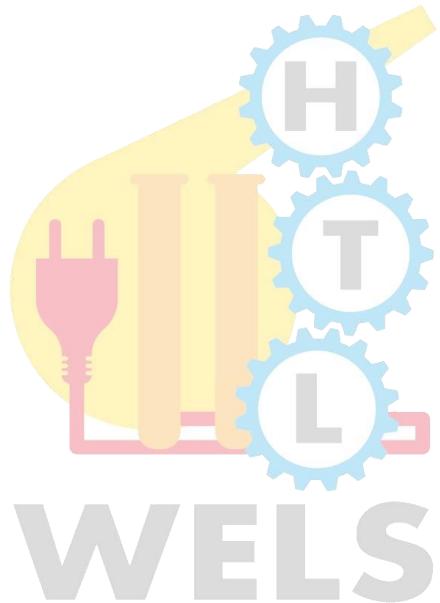
MyWeb.war
/WEB-INF/web.xml
/WEB-INF/classes/com/abc/MyServlet...

Mit Java EE 6

MyWeb.war

/WEB-INF/web.xml
/WEB-INF/ejb-jar.xml

/WEB-INF/classes/com/xyz/MyEJB.class
/WEB-INF/classes/com/abc/MyServlet.class



EJB 3.1 Singleton Beans

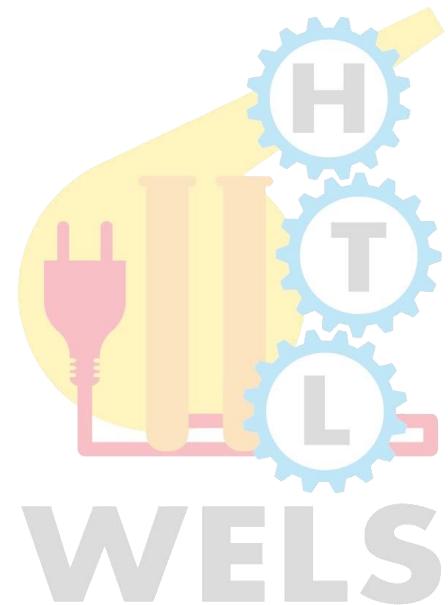
- Singleton Beans werden pro JVM instanziert
 - Erhalten Zustand zwischen Calls
 - Zustand muss nicht persistiert werden
 - Container vs. Bean managed Concurrency
 - @ReadOnly, @ReadWrite Lock Annotations

```
@Startup //Startup Annotation für Initialisierung bei Start
@Singleton //Marker Annotation für Singletons
@DependsOn("MySecondSingleton") //Abhängigkeit zu 2.ter Bean
public class MySingletonEJB {

    private String sharedData="...";

    @PostConstruct
    void init () {...}

}
```



EJB 3.1 – Asynchrone Aufrufe

- EJB 3.1 führt optionale Asynchronität für Session Beans ein
 - Return Typen: `void` oder `java.lang.concurrent.Future<v>`
- Verwendung: `@Asynchronous` Annotation für Klassen/Methoden und `javax.ejb.AsyncResult` als Helper Klasse
- Client kann Abbruch der Operation per `Future<v>.cancel()` erreichen und Bean kann per `SessionContext.isCancelled()` Abbruch prüfen
- Transaktionalität und Persistenz der Operation unterstützt

