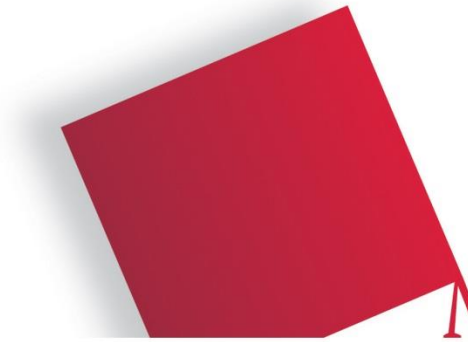


HOCHSCHULE
FÜR ANGEWANDTE
WISSENSCHAFTEN
MÜNCHEN

JavaEE - Grundlagen

FWP Aktuelle Technologien zur Entwicklung verteilter Java-Anwendungen

Michael Theis, Lehrbeauftragter

EINFÜHRUNG IN DIE JAVAEE-PLATTFORM

Die Java EE Plattform

- Java EE Spezifikation definiert
  - ein Programmiermodell für Applikationen
  - die Eigenschaften einer Laufzeitumgebung für Applikationen (Application Server)
- Hersteller liefern konkrete Implementierungen
- Fokus auf server-seitige Applikationen mit web-basierter Benutzeroberfläche
- Aktuelle Version: JavaEE 7

Ziele der JavaEE Plattform

- Standardisierter Applikationscontainer mit essentiellen Infrastrukturdiensten
  - Remoting, Transaktionen, Security, Persistenz
- Leichte und schnelle Entwicklung von portablen, serverseitigen Applikationen
  - POJO-basiertes Modell, Annotations, Convention over Configuration, Dependency Injection, AOP
- Flexibler Technologiestack (Profiles)
- Ausgelegt auf Erweiterbarkeit

Die Entwicklung von Java EE

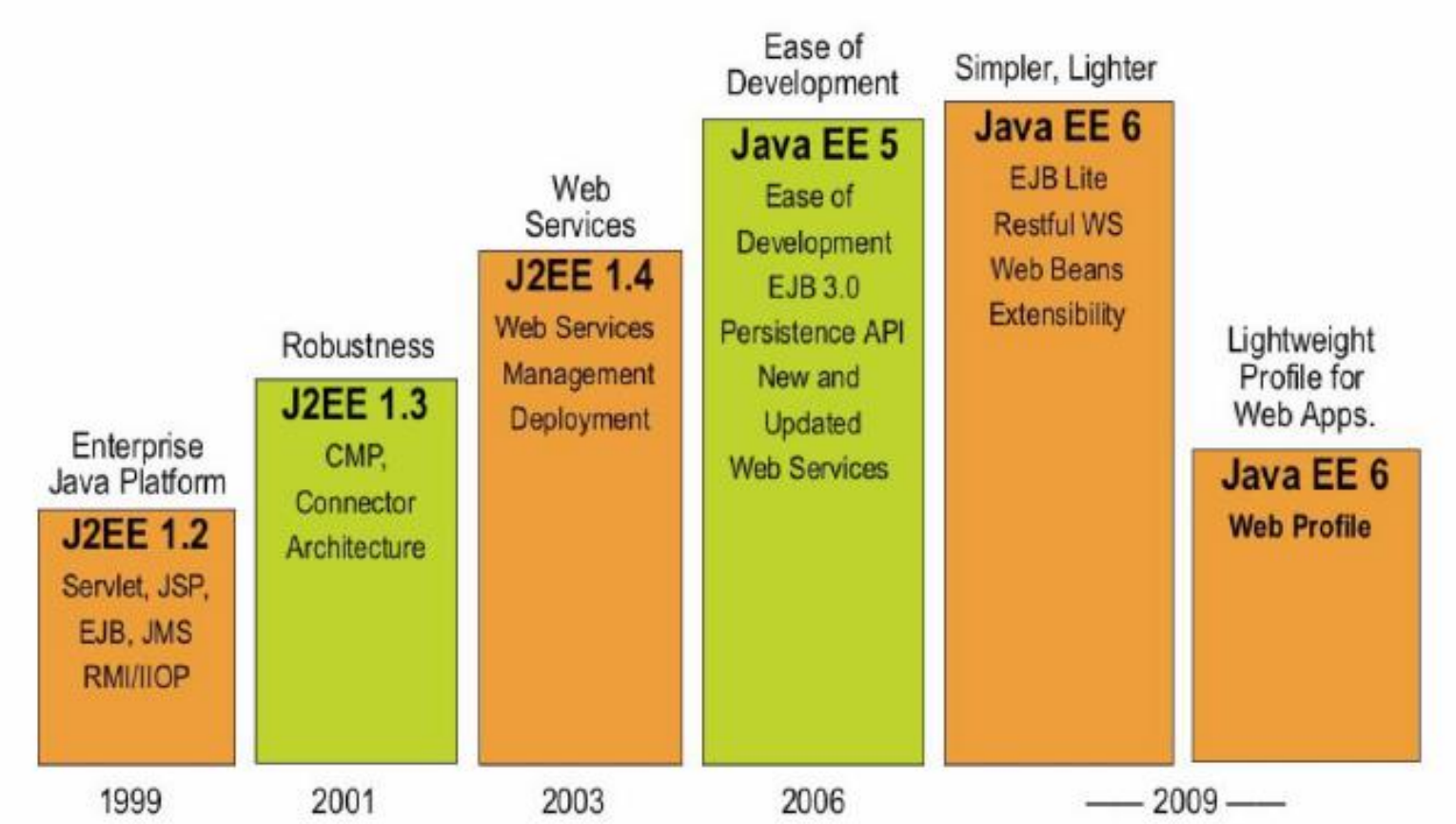
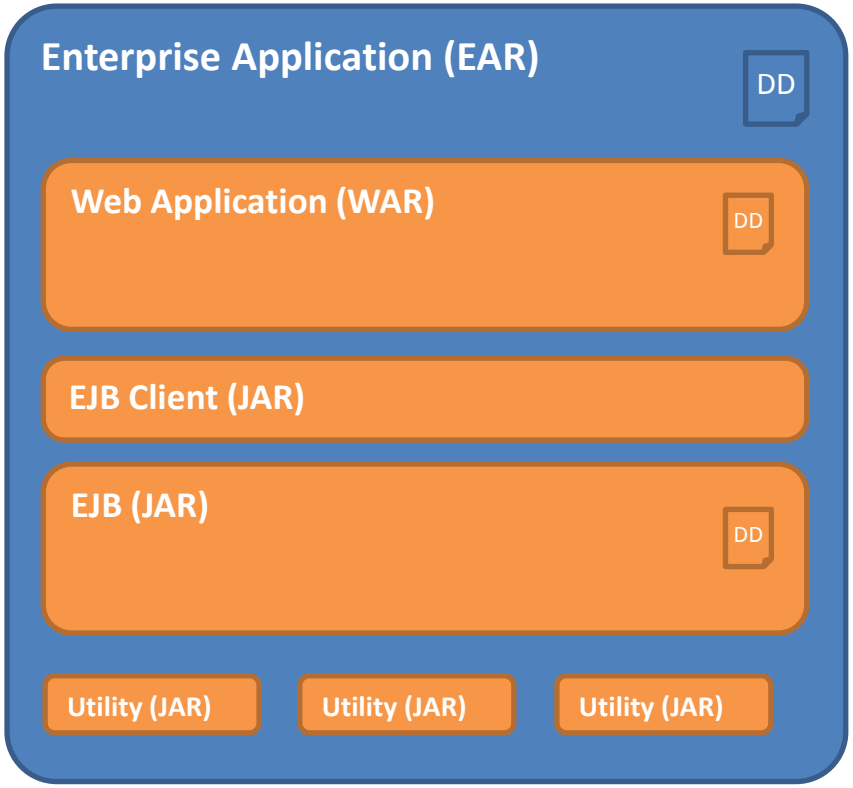


Figure 1: The main theme for the Java EE 6 Platform is a more flexible environment.

Whitepaper: Introduction to the JavaEE 6 Platform, Sun Microsystems Inc. December 2009 (https://www.sun.com/offers/details/java_EE6_overview_paper.xml)

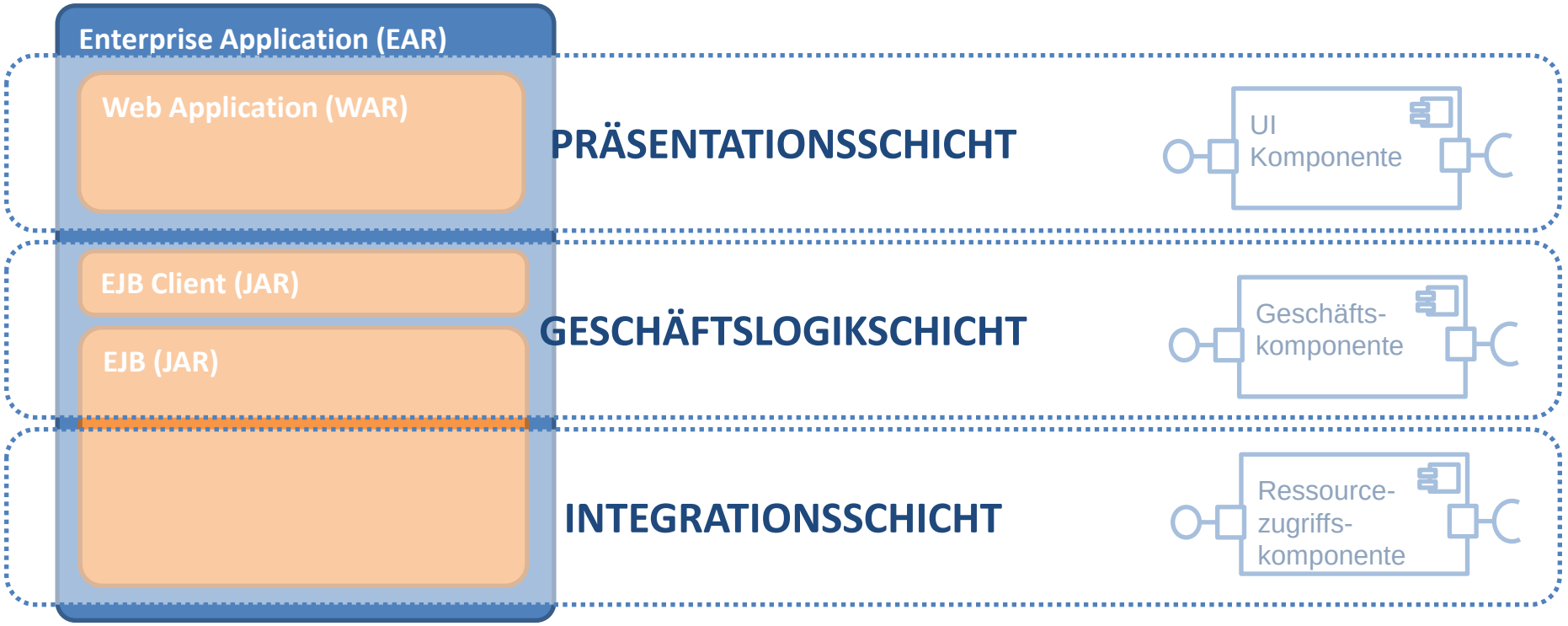
ANATOMIE EINER JAVAEE-APPLIKATION

Module einer Java EE Applikation



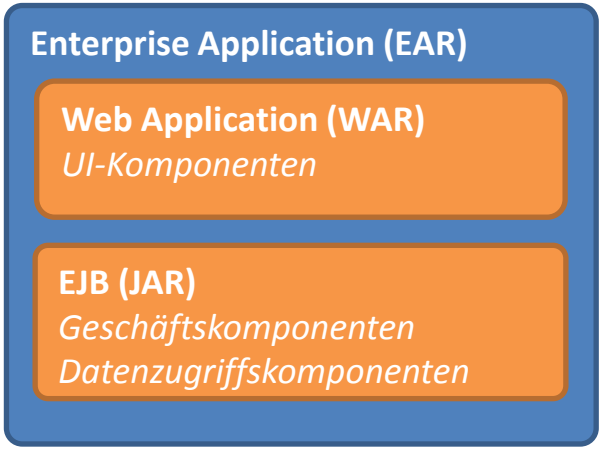
- Eine typische Java EE Applikation wird durch ein Enterprise Application Archive (EAR) repräsentiert
  - ZIP-Datei mit standardisiertem Inhalt
  - In sich vollständige Installationseinheit
- Ein EAR besteht im allgemeinen
  - aus einer Webapplikation (WAR), die die Benutzeroberfläche repräsentiert
  - aus einem EJB JAR mit Enterprise Java Beans, die die Businesslogik der Applikation repräsentieren
  - aus einem EJB Client JAR, welches die Interfaces zu den EJBs zur Verfügung stellt
  - aus mehreren Utility JARs, die Querschnittsfunktionalität für alle Module zur Verfügung stellt (Frameworks, Security, Logging…)

Module, Schichten, Komponenten



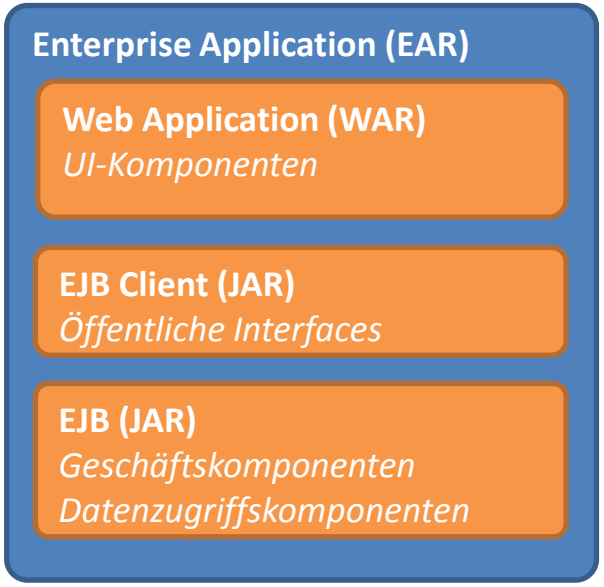
- Alle UI-Komponenten aus der Präsentationsschicht befinden sich im Web-Modul
- Alle Geschäftskomponenten aus der Geschäftslogikschicht befinden sich im EJB-Modul
- Die öffentlichen Interfaces der Geschäftskomponenten befinden sich im EJB-Client-Modul
- Die Ressourcezugriffskomponenten aus der Integrationsschicht befinden sich im EJB-Modul

Modulvariationen



Einfache Komplexität:

- Kein EJB-Client Modul
- EJB-Modul enthält alle Geschäftskomponenten und Ressourcezugriffs-komponenten



Mittlere Komplexität:

- EJB-Client Modul beinhaltet die Interfaces der Geschäftskomponenten und die Domänenobjekte
- EJB-Modul enthält alle Geschäftskomponenten und Ressourcezugriffs-komponenten

Hohe Komplexität:

- Wie *mittlere Komplexität* aber alle Ressourcezugriffs-komponenten werden in ein eigenes EJB-Modul verpackt
- Mehrere Module pro Schicht möglich

GEMEINSAME PATTERNS UND PRINZIPIEN

Gemeinsame Patterns und Prinzipien

DEPENDENCY INJECTION

Inversion of Control (IoC)

- Don´t call us, we call you!
- Kontrolle wandert von der Applikation in das verwendete Framework:

> "One important characteristic of a framework is that the methods defined by the user to tailor the framework will often be called from within the framework itself, rather than from the user's application code. The framework often plays the role of the main program in coordinating and sequencing application activity. This ***inversion of control*** gives frameworks the power to serve as extensible skeletons. The methods supplied by the user tailor the generic algorithms defined in the framework for a particular application."
>
> *Designing Reusable Classes* von Ralph E. Johnson und Brian Foote
> Journal of Object-Oriented Programming Juni/Juli 1988, S. 22-35ff
> http://www.laputan.org/drc/drc.html

Dependency Injection (DI)

- Präzisierung des Inversion of Control-Patterns u.a. von Martin Fowler
- Ein sog. *Assemblierer* löst die Abhängigkeiten zwischen Objekten auf
- Lose gekoppelte POJOs leben in einem *Inversion of Control-Container*
  - Container bestimmt Lebenszyklus der Objekte
  - Container löst als Assemblierer die Abhängigkeiten auf

Consumer benötigt Service

- Implementierungsklasse des Service

```
public class ServiceImpl implements Service
    public void doSomething() {
        // Hier wird etwas gemacht
    }
}
```

- Interface des Service

```
public interface Service {
    public void doSomething();
}
```

- Consumer ist abhängig vom Service-Interface

```
public class Consumer {
    private Service service;
    ...
    public void useService() {
        this.service.doSomething();
    }
```

Wie kommt der Consumer an den Service?

Traditionell

- Consumer erzeugt Objekt von *ServiceImpl* und initialisiert damit Feld *service*:

```
public class Consumer {
  private Service service =
        new ServiceImpl();
```

☹ Consumer wird abhängig von konkreter Implementierung

Mit Dependency Injection

- Consumer markiert Feld *service* als Ziel für Dependency Injection:

```
public class Consumer {
  @Inject
  private Service service;
```

- Container erkennt annotiertes Feld, erzeugt ein Objekt von *ServiceImpl* und injiziert Referenz in Feld *service*

☺ Consumer kennt nicht die konkrete Implementierung

Wie kommen die Objekte in den Container?

- Alle vom Container zu verwaltenden Klassen werden mit Annotationen markiert

```
@Named
public class Consumer { … }
```

```
@Named
public class ServiceImpl { … }
```

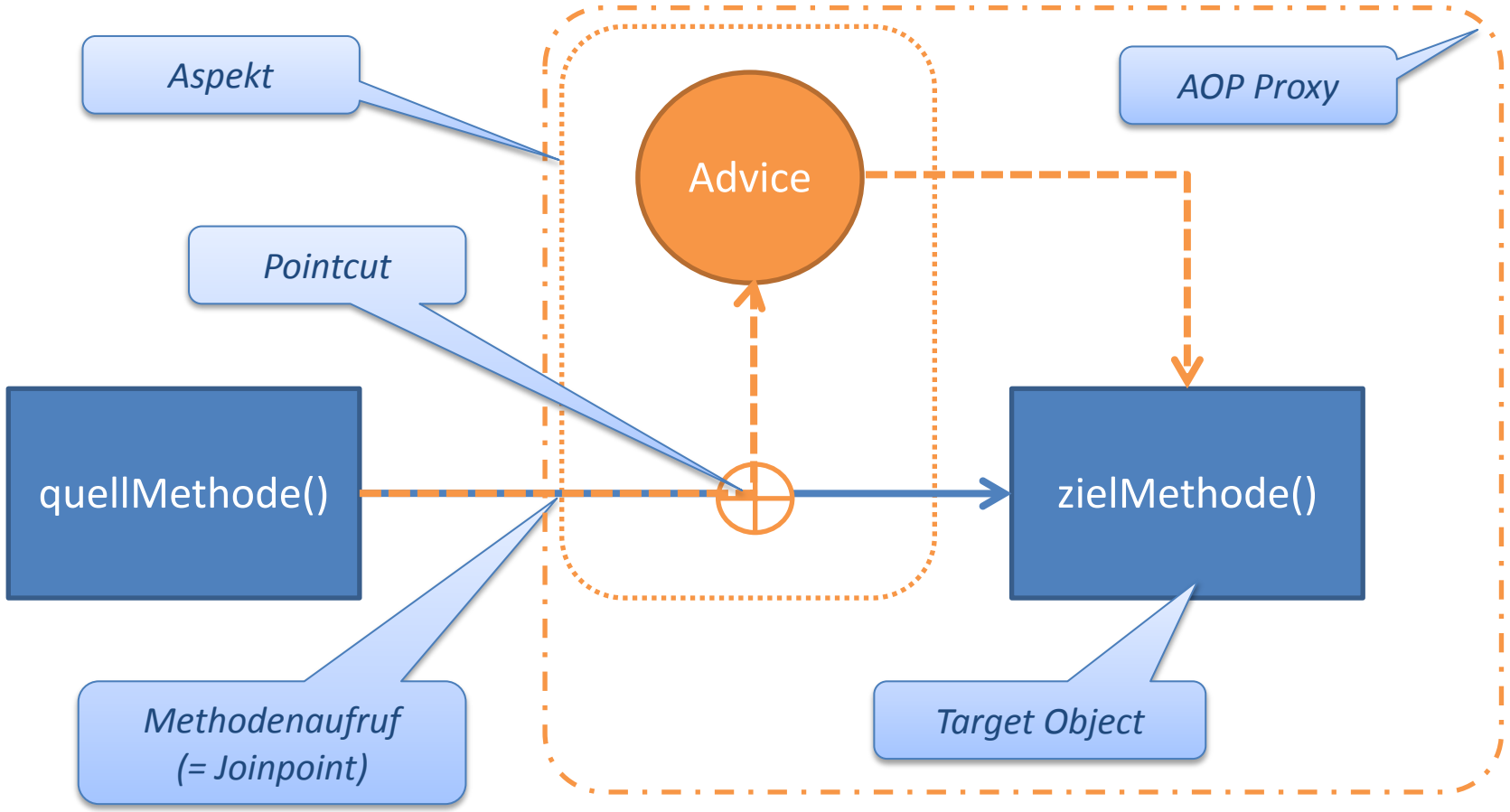
- Container scannt alle Klassen im Klassenpfad
- Objekte der markierten Klassen werden nach Bedarf erzeugt
- Beziehungen zwischen den Objekten werden über Dependency Injection aufgelöst

Gemeinsame Patterns und Prinzipien

ASPECT ORIENTED PROGRAMMING

Aspect Oriented Programming (AOP)

- Modularisierung von ***cross-cutting concerns***



Beispiele für AOP

- EJB-Container verwendet AOP für Transaktionsmanagement, Zugriffskontrolle und Remoting
- JAX-WS verwendet AOP für das Mapping von Methodenaufrufen mit Parametern auf SOAP-Nachrichten und umgekehrt
- JPA verwendet AOP für die Zustandsüberwachung von Entities

Gemeinsame Patterns und Prinzipien

CONVENTION OVER CONFIGURATION

Convention over Configuration (CoC)

- Einfaches Prinzip zur Erleichterung der Programmierung übernommen aus RUBY
- Standardmäßig verwendet die Laufzeitumgebung sinnvolle Voreinstellungen
- Nur im davon abweichenden Fall muss eine Konfiguration vorgenommen werden

Beispiele für CoC

- Stateless Session Beans oder Managed Beans werden unter dem einfachen Klassennamen registriert, falls nichts anderes angegeben
- Methodenaufrufe eines Enterprise Java Beans laufen immer in einem transaktionalen Kontext (TRANSACTION_REQUIRED)
- Die Java Persistence Architecture (JPA) mappt automatisch Klassennamen von Entitäten auf Tabellennamen und Feldnamen auf Spaltennamen

Fragen?

ANHANG

Quellen

- Eric Jendrock et. al.: ***The Java EE 7 Tutorial***
  http://docs.oracle.com/javaee/7/tutorial/doc/
  Oracle September 2013
- Adam Bien: ***Real World Java EE Patterns: Rethinking Best Practices***
  press.adam-bien.com September 2012; ISBN 978-0-300-14931-6
- Eric Evans: ***Domain Driven Design:***
  Tackling Complexity in the Heart of Software
  Addison Wesley 2004; ISBN 0-321-12521-5
- Rod Johnson: ***J2EE Design and Development***
  Wrox Press 2002; ISBN 1-86100-784-1
- Martin Fowler: ***Patterns of Enterprise Application Architecture***
  Addison Wesley 2003; ISBN 0-321-12742-0

Kontakt

Michael Theis
Lehrbeauftragter Hochschule München

email michael.theis@hm.edu
mobile + 49 170 5403805
web http://www.tschutschu.de/Lehrauftrag.html