KaffeeKlatsch
Das Magazin rund um Software-Entwicklung

ISSN 1865-682X
09/2011

# JEE-Komponenten reloaded

## Contexts and Dependency Injection (CDI) im Überblick

von Werner Eberling

# JEE-Komponenten reloaded

## Contexts and Dependency Injection (CDI) im Überblick

von Werner Eberling

Noch ein Komponentenstandard? Im Rahmen der JEE 6 hat mit *Contexts and Dependency Injection* ein neuer Standard das Licht der Welt erblickt. Doch es ist nicht einfach ein zusätzliches Komponentenmodell, das uns hier präsentiert wird, sondern der Versuch, die bestehenden Ansätze zu vereinheitlichen. Darüber hinaus verspricht der neue Standard mehr Typsicherheit und wartet unter anderem mit einem eigenen *Eventing*-Modell auf. Dieser Artikel gibt einen Überblick über die wichtigsten Neuerungen, die CDI bietet.

*Ich stosse mich immer an den spitzen Klammern* – das war die Antwort von Gavin King, als er bei einer ersten Vorstellung der Ideen zu einem neuen JEE-Standard, der damals noch *WebBeans* hiess, auf der JBoss World in Orlando gefragt wurde, warum er keine XML-Deskriptoren mag und daher alles per Annotation ausdrücken möchte.

*Context and Dependency Injection* (kurz: CDI), wie „der Neue" inzwischen genannt wird – auf den Wegfall der XML-Deskriptoren zu reduzieren, wäre töricht, nicht nur weil es sich hierbei lediglich um ein technisches Randdetail handelt. Wer allerdings eine ausgeprägte Abneigung gegen Annotationen besitzt, sei bereits am Anfang dieses Artikels gewarnt, denn ohne dieses, inzwischen nicht mehr wirklich neue Java-Feature, geht inzwischen kaum noch etwas und im Rahmen von CDI genau genommen gar nichts mehr. Da man sich aber an alles, also auch an den exzessiven Umgang mit Annotationen gewöhnt, soll uns das nicht davon abhalten einen genaueren Blick auf das zu werfen, was uns die aktuelle JEE-Version mit CDI beschert hat.

### Die Idee

Server-seitige Komponenten sind nun wirklich nichts Neues innerhalb der JEE. Die Web-Schicht hat ihre Komponenten in Form von *Java Servlets* oder *JSF-ManagedBeans*, die Businesslogik-Schicht in Form von *Enterprise JavaBeans* (EJB). Doch genau hier beginnt bereits das Problem. Denn für eine EJB ist ein *Servlet* (oder eine JSF-ManagedBean) im Grossen und Ganzen ein Client, wie eine entfernte *Swing*-Applikation. Das heisst, selbst wenn Web-Anwendung und EJB-Anwendung im selben *Application-Server* existieren, könnten sie aus Sicht des Programmiermodells kaum weiter voneinander entfernt sein.[1]

Unter anderem diese Diskrepanz wird von CDI adressiert, indem ein neuer, allgemeinerer Komponentenbegriff eingeführt wird: die *CDI-Bean*. Dabei handelt es sich um ein sogenanntes *managed object*, also ein Objekt, das innerhalb eines Containers existiert.[2] Dabei wird im ersten Schritt von der konkreten Implementierung abstrahiert. Das bedeutet, eine solche Komponente kann als EJB, als JSF-ManagedBean oder als *Plain Old*

[1] Der JEE-erfahrene Leser mag nun argumentieren, dass sich Web-Komponenten, im Gegensatz zu entfernten Clients, ja dem Konzept der Dependency Injection bedienen können und so einfacher auf EJB-Komponenten zugreifen können. Das ist zwar soweit richtig, allerdings nur ein kleiner technischer Kniff, der einen *JNDI-Lookup* überflüssig macht, aber nicht wirklich zur besseren Integration von EJB-Komponenten in die Web-Schicht beiträgt.

[2] Genau genommen handelt es sich auch bei einer CDI-Bean um eine *Managed-Bean*. Um allerdings die Verwechslung mit *JSF ManagedBeans* zu vermeiden, wird in diesem Artikel weiter nur von CDI-Beans die Rede sein.

*Java Object* (POJO) implementiert werden. Hier werden also bestehende Komponentenansätze und deren Container wiederverwendet, anstatt das Rad wieder einmal neu zu erfinden.

Bei CDI handelt es sich also erst einmal um einen Ansatz zur Zusammenführung bestehender Komponentenansätze. Zu beachten bleibt dabei, dass es sich per Definition immer um server-seitige Komponenten handelt. Das heisst, auch wenn CDI-Beans als POJOs implementiert werden können, wird immer noch ein JEE-Container als Laufzeitumgebung benötigt.

## Der Context

Ein einheitlicher Komponentenansatz über die verschiedenen Schichten hinweg, das hört sich gut an, aber was bedeutet es im Detail? Verhalten sich JSF-ManagedBeans nicht anders als EJBs?

Die Antwort auf diese Frage ist ein klares *Jaein*. Nicht ohne Grund handelt es sich hier um verschiedene Komponententypen, denn die Web-Schicht adressiert andere Probleme als die Businesslogik-Schicht. So sind die Methoden von EJBs automatisch transaktional; wohingegen ManagedBeans einen Kontextbezug besitzen und über *ExpressionLanguage*-Ausdrücke (EL) referenzierbar sind. Hier bringt CDI eine gewisse Vereinheitlichung mit sich. Um zum Beispiel direkt aus Web-Anwendungen heraus referenzierbar zu sein, besitzt jede CDI-Bean (egal ob JSF-ManagedBean oder EJB) einen Kontextbezug, also eine klar definierte Lebenszeit, und einen Namen über den sie per EL-Ausdruck angesprochen werden kann.

| Kontext | Passivierbar? |
|---|---|
| *Request* | Nein |
| *Conversation* | Ja |
| *Session* | Ja |
| *Application* | Nein |

## In CDI verfügbare Kontexte

Wie sieht das nun im Code aus? Wie bereits erwähnt, kann eine CDI-Bean als POJO realisiert werden. Dazu muss die entsprechende Klasse, die den JavaBeans-Konventionen entsprechen muss, lediglich mit der Annotation *@Named* versehen werden, die sie für EL-Ausdrücke zugänglich macht. Wird dabei kein weiterer Parameter angegeben, wird der Name der Klasse zum EL-Namen der Bean. Zur Festlegung des Kontexts in dem die Bean existieren soll, wird eine weitere Annotation, im folgenden Beispiel *@SessionScoped*, verwendet. Sobald die Bean in einem passivierbaren Kontext, wie dem gewählten *SessionScope*, abgelegt werden soll, muss sie zusätzlich serialisierbar sein.

```
@javax.enterprise.context.SessionScoped
@javax.inject.Named
public class User implements Serializable{

   public User(){...}
   public User(String name, String vorname){...}
   public String getName(){...}
   public void setName(String name){...}
   // ...
}
```

Wird die Klasse *User* im Rahmen einer Web-Applikation deployed, kann sie aus einer JSF-Seite heraus über einen EL-Ausdruck referenziert werden:

```
<h:outputText value=
   "Angemeldet als #{user.nachname}, #{user.vorname}"/>
```

Die CDI-Bean verhält sich also wie eine JSF-ManagedBean, auch wenn sie nicht mit den bekannten JSF-Annotationen versehen wurde. Doch wenn die Klasse nicht passend annotiert wurde, wie kann sie als JSF-ManagedBean verfügbar sein? Hierfür sorgt die CDI-Implementierung, die in jedem JEE6-Server verfügbar sein muss. Sie reagiert auf die gesetzten Annotationen und macht die Klasse für JSF verfügbar. Dies funktioniert allerdings nur, wenn das Web-Archiv, das die JSF-Seite und die Klasse enthält, im *WEB-INF*-Verzeichnis einen zusätzlichen, in diesem Fall leeren Deskriptor mit dem Namen *beans.xml* enthält.[3]

So weit so gut. Aber wieso eine neue Notation für etwas, was mit JSF ManagedBeans schon möglich war? Der vereinheitlichte Komponentenansatz erscheint im ersten Moment eher als ein akademisches Argument, das im weiteren Verlauf dieses Artikels aber noch genauer beleuchtet werden wird. An dieser Stelle sollen erst einmal andere Vorzüge des CDI-Standards betrachtet werden.

## Dependency Injection

Als erstes springt hier der mächtigere Dependency Injection-Ansatz ins Auge. Auch wenn dieses Konzept schon aus JEE5 bekannt ist, so war es doch recht ein-

[3] Auf eine sarkastische Bemerkung bezüglich spitzer Klammern und der Einführung eines neuen XML-basierten Deskriptors wurde an dieser Stelle explizit verzichtet. Hier möge sich der geneigte Leser selbst eine Meinung bilden.

geschränkt, da nur auf spezielle JEE-Komponenten und Infrastrukturobjekte beschränkt. Mit CDI sieht das nun anders aus. Auch hier gilt die Daumenregel: Alles was eine CDI Komponente ist, kann injiziert werden. Soweit erstmal kein allzu grosser Unterschied zum bekannten Vorgehen. Doch in CDI gibt es deutlich mehr Freiheiten zur Implementiernung von Komponenten. So ist es möglich POJOs als CDI-Komponenten zu deklarieren und sie so zu injizieren. Diese Komponenten müssen dabei nicht zwingend einen Namen besitzen, da die Dependency Injection über den Typ der zu injizierenden Komponente erfolgt.

```
@javax.enterprise.context.RequestScoped
@javax.inject.Named
public class Registration{
  @javax.inject.Inject
  private User user;

  public String register(){
  //...
  someObject.createdBy(user);
  //...}
  // …
}
```

Ein weiterer Vorteil dieses erweiterten Dependency Injection-Mechanismus ist die Vielzahl an möglichen Punkten, an denen Objekte injiziert werden können (*InjectionPoints*).

Neben der im Beispiel verwendeten *Field-Injection*, können auch Parameter von Konstruktoren oder Methoden Ziele von Injizierungen sein:

```
@javax.enterprise.context.RequestScoped
@javax.inject.Named
public class Registration{

@javax.inject.Inject
 public String register(User user){
  //...
  someObject.createdBy(user);
  //...}
 // …
}
```

## Erzeugung und Initialisierung

Sobald Dependency Injection zum Einsatz kommt, stellt sich die Frage nach *Lifecycle-Callback*-Methoden, da der Konstruktor einer Komponente, die sich Referenzen injizieren lässt, nur noch eingeschränkt zu Initialisierung verwendet werden kann. Hier kommen die aus dem *Common Annotations*-Standard bekannten Annotationen *@PostConstruct* und *@PreDestroy* zum Einsatz.

Das nachfolgende Beispiel nutzt den *Callback* nach der Komponentenerzeugung um die CDI-Bean zu initialisieren. Der injizierte *javax.security.Principal* ist ein spezielles Objekt, das in einer CDI-Umgebung zur Verfügung steht, sobald ein *Login* stattgefunden hat:[4]

```
@javax.enterprise.context.SessionScoped
@javax.inject.Named
public class User implements Serializable{
 @javax.inject.Inject
 private Principal principal;
 public User(){...}
 @javax.annotation.PostConstruct
  void initUser(){
  computeAndSetNameAndFirstname(principal);
 }
 // …
}
```

In den bisherigen Beispielen wurde implizit davon ausgegangen, dass ein CDI-Bean bei Bedarf per Konstruktoraufruf erzeugt wird. Dies ist die einfachste Variante der Komponentenerzeugung. Dabei kann der *Default-Konstruktor* oder genau ein Konstruktor mit Parametern verwendet werden, der in diesem Moment mit *@Inject* als *InjectionPoint* gekennzeichnet werden muss. Weitere Konstruktoren dürfen in beliebiger Zahl und Form existieren, werden aber vom Container nicht zur Erzeugung verwendet.

In speziellen Fällen, in denen die Erzeugung einer Komponente ein komplexerer Vorgang ist, kann es sich anbieten *Factory*-Methoden zu verwenden. Auch dieser Ansatz wird von CDI unterstützt. Hierzu muss die jeweilige Methode mit der Annotation *@Produces* versehen werden. Zusätzlich trägt sie die Annotationen zur Definition des *Scopes* und zur Angabe des Namens der zu erzeugenden Komponente.

```
@javax.enterprise.context.ApplicationScoped
public class UserInfo{
 @javax.enterprise.inject.Produces
 @javax.enterprise.context.SessionScoped
 @javax.inject.Named
 public User createUser(Principal principal){...}
 // …
}
```

[4] Dabei spielt es keine Rolle, ob die Authentifizierung in der Web-Schicht (via http) oder in der EJB-Schicht (via JAAS) stattgefunden hat, ein weiteres Beispiel für die Vorteile eines einheitlichen Komponentenansatzes bzw. der Kapselung technischer Details.

Parameter solcher Erzeugermethoden sind automatisch *InjectionPoints*. Somit wird das *Principal*-Objekt im oberen Beispiel vom Container zur Verfügung gestellt. Hierbei ist darauf zu achten, dass die Methode nicht(!) zusätzlich mit *@Inject* annotiert wird!

Erfolgt die Komponentenerzeugung durch eine andere Klasse als die Komponentenklasse selbst, so finden sich alle Annotationen, die die Komponente definieren, in der erzeugenden Klasse und nicht in der eigentlichen Komponentenklasse:

```
public class User implements Serializable{...}
```

Es muss aber nicht immer eine Methode sein. Auch Attribute einer Klasse können zur Erzeugung von Komponenten herangezogen werden:

```
@javax.enterprise.context.ApplicationScoped
public class Marquee{
 @javax.enterprise.inject.Produces
 @javax.enterprise.context.SessionScoped
 @javax.inject.Named
 private List<Registration> registrations;
 // …
}
```

## Qualifier und Stereotypes

Bei dieser ganzen Injiziererei stellt sich die Frage, was passiert, wenn es mehrere Komponenten des selben Typs gibt. Die bisher verwendete User-Komponente könnte zum Beispiel in zwei Varianten existieren: eine für den gerade angemeldeten User und eine für einen allgemeinen Gast-User.

Nachdem die Auflösung der Abhängigkeiten bisher über genau diesen Typ lief, wird der Container die Situation mit einer Fehlermeldung quittieren. Abhilfe schaffen hier die sogenannten *Qualifier*. Sie dienen zur genaueren Spezifizierung des zu injizierenden Objektes und können nach Bedarf, in Form eigener Annotationen, definiert werden:

```
@javax.inject.Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target(
  {ElementType.FIELD, ElementType.METHOD,
    ElementType.PARAMETER, ElementType.TYPE}
)
public @interface LoggedIn {}
```

Um nun zu bestimmen, welche Variante eines Komponententyps erzeugt wird bzw. injiziert werden soll, muss der entsprechende *Qualifier* sowohl an der Komponente (oder der erzeugenden Methode) als auch am *InjectionPoint* angegeben werden.

```
@javax.enterprise.context.ApplicationScoped
public class UserInfo{
  @javax.enterprise.inject.Produces
  @javax.enterprise.context.SessionScoped
  @javax.inject.Named
  @LoggedIn
  public User createLoginUser(Principal principal
    {…}

  @javax.enterprise.inject.Produces
  @javax.enterprise.context.SessionScoped
  @javax.inject.Named
  @Guest
  public User createGuestUser(Principal principal
    {…}
  // …
}

@javax.enterprise.context.RequestScoped
@javax.inject.Named
public class Registration{
 @javax.inject.Inject
 @LoggedIn
 private User user;
 //...
}
```

Wem das dann nun doch zu viele Annotationen auf einmal waren, der sei noch einmal an das Zitat von Gavin King am Anfang dieses Artikels erinnert. Und sich an „runden“ @-Zeichen zu stossen, ist vielleicht doch nicht ganz so schlimm wie an spitzen Klammern. Allerdings hat auch CDI einen Mechanismus der Annotationsflut Herr zu werden. Gibt es Kombinationen aus verschiedenen CDI-Annotationen die immer wieder zur Definition von CDI-Beans verwendet werden, so können diese zu einer Annotation zusammengefasst werden. Diese Annotation wird als *Stereotype* bezeichnet.

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@javax.enterprise.context.ApplicationScoped
@javax.inject.Named
@javax.enterprise.inject.Stereotype
public @interface SingletonService {..}
```

Hierbei steht natürlich nicht im Vordergrund die Anzahl der bei der Definition einer CDI-Bean zu setzenden

Annotationen zu reduzieren, sondern architekturelle Konzepte in Annotationen zu giessen. Im Code wird nur noch definiert, dass eine Komponente ein *Singleton-Service* ist. Was dies konkret bedeutet, kann zentral definiert werden, wodurch die Wahrscheinlichkeit, dass *SingletonServices* die erwarteten Eigenschaften nicht besitzen, weil vergessen wurde sie zu annotieren, deutlich reduziert wird.

```
@SingletonService
public class Registration {..}
```

## Persistenz und Transaktionssicherheit

In den bisherigen Betrachtungen ging es darum CDI-Beans zu erzeugen und zu verwenden, ohne sich Gedanken um Aspekte wie Persistenz zu machen. Wenn es darum geht in der Web-Schicht erzeugte Objekte zu speichern, kommt sehr schnell der Ruf nach einer EJB-Komponente auf, die diese Aufgabe im Zusammenspiel mit JPA einfach und transaktionssicher abhandeln kann. Zwar wäre es auch möglich JPA direkt aus einem POJO-basierten CDI-Bean zu verwenden, aber die Einbindung einer EJB-Komponente soll hier als Beispiel für den schichtenübergreifenden Ansatz von CDI dienen.

Aus einer POJO-basierten CDI-Bean ein persistentes Objekt zu machen ist kein Hexenwerk. Alles was dafür notwendig ist, ist die Bean als JPA-Entität zu definieren. Neben den CDI-Annotationen tauchen jetzt also auch JPA-Annotationen in der Komponenten-Implementierung auf:

```
@javax.enterprise.context.RequestScoped
@javax.inject.Named
@javax.persistence.Entity
public class Customer {
 @javax.persistence.Id
 @javax.persistence.GeneratedValue
 private int id;
 private String name;
 private String firstname;
 //...
}
```

Soweit noch nicht allzu spannend. Doch da die CDI-Bean mit der Annotation *@Named* versehen ist, kann sie direkt aus einer JSF-Seite heraus referenziert und mit Daten gefüllt werden. Es entfällt also die technische Notwendigkeit zur Trennung des Datenmodells der Web-Schicht und des Persistenzmodells.[5]

Zur Persistierung (inklusiv deklarativer Transaktionssteuerung) wird nun noch eine EJB benötigt, die die JPA-Entität unter Zuhilfenahme eines *EntityManagers* persistiert. Auch diese Bean wird als CDI-Komponente annotiert und kann somit direkt aus einer JSF-Seite aufgerufen werden, wodurch die bisher notwendige Zwischenschicht aus JSF-ManagedBeans entfällt.

```
@javax.enterprise.context.Dependent
@javax.inject.Named
@javax.ejb.Stateless
public class Registration implements IRegistration {...}
```

```
<h:form>
  <h:inputText value="#{customer.name}"/>
  <h:inputText value="#{customer.firstname}"/>
  <h:commandButton
    value="Registrieren"
    action="#{registration.register(customer)}"/>
</h:form>
```

## Eventing

Der einfache Zugriff aus JSF-Seiten in die EJB-Schicht ist ein gutes Beispiel für den integrativen Ansatz von CDI. Doch CDI geht noch einen Schritt weiter. Mit einem neu eingeführten *Eventing*-Modell ist es möglich CDI-Komponenten nach dem *Observer-Pattern* lose miteinander zu koppeln. Nachdem diese Komponenten sowohl in der Web- als auch in der EJB-Schicht angesiedelt werden können, ist es möglich in der Businesslogik-Schicht *Events* zu erzeugen, die in der Präsentationsschicht empfangen und verarbeitet werden. Ein Vorgehen, dass dem ein oder anderen aus dem Umfeld des *MVC-Patterns* bekannt sein dürfte.

Wie bei allen Neuerungen des CDI-Standards, ist auch die Verwendung des Eventings überraschend einfach. Alles was eine Komponente benötigt um Events zu verschicken, ist ein POJO, welches das Event repräsentiert und ein CDI-Objekt, über das dieses Event verschickt werden kann. Das Verschicken von Events wird durch die Klasse *javax.enterprise.event.Event* implementiert, die per Dependency Injection an ein beliebiges Attribut der Komponente gebunden werden kann, die Events verschicken möchte. Der Typ des Events wird dabei als

[5] Ungeachtet dieser technischen Möglichkeit, sei hier darauf hingewiesen, dass es auch weiterhin eine Reihe guter Argumente für die Trennung dieser beiden Modelle gibt. Lediglich der technische Zwang ist dank CDI verschwunden, so dass in einfachen Anwendungsfällen die JPA-Entität direkt verwendet werden kann.

Typparameter angegeben. Eine genauere Beschreibung des Events ist durch den Einsatz von *Qualifiern* möglich.

```
@SingletonService
public class Registration {
  @Inject @Registered
  private Event<Customer> newCustomerEvent;

  public String register(Customer customer) {
    newCustomerEvent.fire(customer);
  //...
  }
}
```

Events können mit jeder beliebigen Methode empfangen werden. Diese Methode muss lediglich den Event-Typ (Typ des POJOs, nicht *javax.enterprise.event.Event*) als Parameter erwarten und diesen mit der Annotation *@Observes* als Empfänger für Events dieses Typs kennzeichnen. Auch hier ist eine weitere Einschränkung über *Qualifier* möglich.

```
@SingletonService
public class Marquee {
  public void addRegistration(
@javax.enterprise.event.Observes
  @Registered Customer customer) {...}
  //...
}
```

## Weiterführende Themen

Neben den bisher vorgestellten Features wartet CDI mit einer Reihe weiterer Neuerungen auf. Diese umfassen unter anderem den einfachen Austausch von Komponenten-Implementierungen durch Alternativen oder Spezialisierungen, ein erweitertes *Interceptor*-Konzept und die Möglichkeit zur Implementierung von *Dekoratoren*. Zusätzlich führt CDI mit der *Conversation* einen neuen Gültigkeitsbereich ein, der mehrere *Requests* umfassen und dessen Gültigkeitsdauer programmatisch beeinflusst werden kann. Wem das nicht reicht, der hat auch die Möglichkeit einen eigenen Context zu implementieren.

Diese weiterführenden Themen im Detail zu beleuchten, würde den Rahmen dieses Artikels sprengen, weswegen sie hier der Vollständigkeit halber nur kurz angesprochen wurden. Der an mehr Details interessierte Leser, sei an dieser Stelle auf eines der zahlreichen CDI-Tutorien [2], die Dokumentation der Referenzimplementierung [3] oder die Beispielanwendung zu diesem Artikel [4] verwiesen, die alle im Internet verfügbar sind.

## Fazit

Contexts and Dependency Injection bietet einen einheitlichen Komponentenansatz, der die Integration von Web- und EJB-Schicht deutlich vereinfacht. Auch wenn die Gründe für die Einführung unterschiedlicher Schichten in grossen Projekten immer noch Ihre Gültigkeit haben, so ermöglicht CDI die Eliminierung unnötiger *Glue-Code*-Schichten, deren einzige Daseinsberechtigung in der Integration mit einer „fremden" Komponententechnologie bestand. Darüber hinaus adressiert der neue Standard eine Reihe wunder Punkte der Java Enterprise Edition.

Wer Vorteile aus der flexiblen Komponentenerzeugung, dem Eventing-Modell oder den weiterführenden Features wie *Interceptoren* oder *Conversations* ziehen kann, ist bei CDI genau richtig.

Wenn es „nur" darum geht eine reinrassige JSF- oder EJB-Anwendung zu schreiben, welche die neuen Features nicht benötigt, sollten auch die CDI-Annotationen außen vor bleiben. Denn *@ManagedBean* und *@Stateless* funktionieren noch immer wie gewohnt und der Wiedererkennungseffekt des Bekannten ist in Projekten nie zu unterschätzen.

## Ein ausführlicheres Beispiel

Die in diesem Artikel vorgestellten Code-Ausschnitte sind Teil einer grösseren CDI-Beispiel-Anwendung. Diese enthält auch Beispiele für die hier nur kurz angesprochenen „weiterführenden Themen" und steht im Internet zum Download zur Verfügung [4].

### Referenzen

[1] JSR-299 *Contexts and Dependency Injection*
http://jcp.org/en/jsr/detail?id=299

[2] Andy Gibson *CDI Tutorial*
http://www.andygibson.net/blog/tutorial/getting-started-with-jsf-2-0-and-cdi-in-jee-6-part-1

[3] Weld *CDI-Refernzimplementierung*
http://docs.jboss.org/weld/reference/1.0.0/en-US/html

[4] Eberling, Werner *CDI-Beispielanwendung*
http://www.werner-eberling.de/download/CDI-Registration.zip

### Kurzbiographie

Werner Eberling ist Principal Consultant und Technical Lead bei der MATHEMA Software GmbH in Erlangen. Seit 1999 beschäftigt er sich mit verteilten Systemen, sein Fokus liegt dabei auf CORBA und der JEE. Sein aktuelles Steckenpferd ist die Anwendungsentwicklung für mobile Endgeräte bzw. deren Anbindung an Enterprise-Systeme. Neben seiner Projekttätigkeit hält er Technologie-Trainings in Deutschland und der Schweiz und ist als Sprecher auf nationalen und internationalen Konferenzen anzutreffen.

Werner Eberling ist unter anderem Autor des Buches *Enterprise JavaBeans 3.1 – Das EJB-Praxisbuch für Ein- und Umsteiger*, erschienen im Hanser Verlag.