KaffeeKlatsch
Das Magazin rund um Software-Entwicklung

ISSN 1865-682X
08/2013

Höher, schneller, weiter

Erweiterungen für CDI entwickeln

von Werner Eberling

Höher, schneller, weiter

Erweiterungen für CDI entwickeln

von Werner Eberling

Der inzwischen nicht mehr ganz so neue CDI-Standard kommt mittlerweile in immer mehr Projekten zum Einsatz. Doch was tun, wenn die *out-of-the-box*-Funktionalität des „Neuen“ nicht ausreicht? Für solche Situationen bietet CDI einen eingebauten Erweiterungsmechanismus, der dem Entwickler interessante Möglichkeiten zum Ausbau des Frameworks bietet. Wie dies vonstatten geht, soll in diesem Artikel anhand zweier Beispiele genauer betrachtet werden.

Im Rahmen der *JEE6* hat mit *Contexts and Dependency Injection* ein neuer Standard das Licht der Welt erblickt. Seitdem ist nun schon einige Zeit ins Land gegangen und so findet „der Neue“ inzwischen den Weg in immer mehr *Java EE*-Projekte. Doch wie es immer ist, finden sich neben jeder Menge schöner neuer Features auch oft Themen, die im Standard (noch) keine Berücksichtigung gefunden haben. Doch anstatt zu grollen, heißt es in diesem speziellen Fall „selbst Hand anzulegen“.

Anders als so manch anderer JEE-Standard ist CDI von Anfang an auf (funktionale) Erweiterung ausgelegt. Bereits die Spezifikation [1] [2] spricht auf einer der ersten Seiten von „portable Extensions“ und auch die Dokumentation der Referenzimplementierung bringt es am Anfang des entsprechenden Kapitels auf den Punkt: „CDI is intended to be a foundation for frameworks, extensions and integration with other technologies“ [3].

Extensions

Das Mittel der Wahl, wenn es darum geht CDI zu erweitern, ist also die *Portable Extension*. Diese ist als Teil des *Service Provider Interfaces* (SPI) nach eigenen Angaben dafür zuständig:

- Eigene *CDI Beans*, *Interceptoren* und *Dekoratoren* bereitzustellen.
- *Dependency Injection* in eigene Objekte zu realisieren.
- Eigene Kontexte zu implementieren.
- Die annotierten Metadaten von CDI-Beans zu erweitern oder abzuändern.

Aber wie implementiert man nun so eine Erweiterung? Das *CDI-SPI* sieht dafür das leere Service-Interface *javax.enterprise.inject.spi.Extension* vor. Eine Klasse, die dieses Interface implementiert, kann über den Service Provider-Mechanismus als Erweiterung registriert werden. Hierzu muss der vollqualifizierte Klassenname der Erweiterung in einer Datei mit dem Namen *META-INF/services/javax.enterprise.inject.spi.Extension* hinterlegt werden.

Das leere Service-Interface bringt selbst erst einmal keine Funktionalität mit sich. Die Arbeit in einer *Extension* wird in einer oder mehreren *Callback*-Funktionen geleistet, die spezielle vom CDI-Container gefeuerte Events abfangen.[1] Nachdem dies über den CDI-eigenen leichtgewichtigen Eventing-Mechanismus vonstatten geht, ist man hier in der Wahl der Methodennamen grundsätzlich vollkommen frei. Es hat sich allerdings

[1] Auf eine komplette Liste all dieser Events wird an dieser Stelle aus Gründen der Übersichtlichkeit verzichtet. Der geneigte Leser sei hierfür z. B. auf Kapitel 16 der Dokumentation der Referenzimplementierung (Weld) verwiesen [3].

eingebürgert die Namen der jeweiligen Events dafür zur verwenden, um den Wiedererkennungswert zu steigern.

```
class EineExtension implements Extension {
  void beforeBeanDiscovery(
    @Observes BeforeBeanDiscovery bbd
  ) {
    Logger.global.debug(
      "Die Suche nach CDI-Komponenten startet..."
    );
  }
  <T> void processAnnotatedType(
    @Observes ProcessAnnotatedType<T> pat
  ) {
    Logger.global.debug("Analysiere: " +
      pat.getAnnotatedType().getJavaClass().getName()
    );
  }
  void afterBeanDiscovery(
    @Observes AfterBeanDiscovery abd
  ) {
    Logger.global.debug(
      "Suche nach CDI-Komponenten beendet."
    );
  }
}
```

Nun stellt sich natürlich die Frage, was man mit einem solchen Erweiterungsmechanismus sinnvolles anstellt. Die folgenden beiden Beispiele sollen hierfür eine gewisse Anregung geben.

Die Startup-Extension

Eine Problemstellung, die immer wieder in CDI-Projekten auftaucht, ist das Starten von *ApplicationScoped*-Komponenten bereits beim Hochfahren der Anwendung. Zwar gibt es die *@Startup*-Annotation für *Singleton SessionBeans*, aber nur für diese Anforderung die Komponenten gleich als *EJBs* zu implementieren erscheint doch etwas übertrieben. Hier stellt eine *Extension* die deutlich leichtgewichtigere Lösung dar oder etwas plakativer ausgedrückt: „Sie haben ein Problem? Schreiben Sie sich einfach eine passende Annotation!“

Zeit zu coden

Der erste Teil der Aufgabe ist schnell erledigt. Nachdem die *Startup*-Annotation implementierungsneutral gehalten werden soll, kann das aus *EJB* bekannte *@Startup* nicht verwendet werden, es muss also eine neue Annotation her:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Startup {
}
```

Natürlich ist das erst die halbe Miete, denn die Annotation dient nur zur Kennzeichnung der Komponenten, die beim Start des CDI-Containers erzeugt werden sollen. Jetzt wird noch das Stück Code benötigt, das die eigentliche Arbeit übernimmt. In diesem Fall wäre das eine *CDI-Extension*, die sich an zwei Stellen in den Startprozess des Containers einhängt:

1. Im Rahmen der CDI-Bean-Verarbeitung werden die Beans identifiziert, die gestartet werden sollen (Event: *ProcessBean*).
2. Nachdem sichergestellt ist, dass alle Abhängigkeiten erfüllt sind (Event: *AfterDeploymentValidation*), werden die benötigten Instanzen der identifizierten Beans erzeugt.

```
public class StartupExtension implements Extension {
  private final Set<Bean<?>> startupBeans =
    new LinkedHashSet<Bean<?>>();
  <X> void processBean(
    @Observes ProcessBean<X> event
  ){
    if (
      event.getAnnotated().isAnnotationPresent(
        Startup.class
      )
      &&
      event.getAnnotated().isAnnotationPresent(
        ApplicationScoped.class
      )
    ){
      startupBeans.add(event.getBean());
    }
  }
  void afterDeploymentValidation(
    @Observes AfterDeploymentValidation event,
    BeanManager manager
  ){
    for (Bean<?> bean : startupBeans){
      manager.getReference(
        bean, bean.getBeanClass(),
        manager.createCreationalContext(bean)
      ).toString();
    }
  }
}
```

Wird jetzt noch diese *Extension* als *ServiceProvider* eingetragen, so wird diese beim Start des Containers erzeugt und sorgt dafür, dass alle mit *@Startup* und *@ApplicationScoped* versehenen CDI-Komponenten direkt instantiiert werden.

Das nächste Level – ein eigener Scope

Nach der Einführung einer eigenen semantischen Annotation, wird es nun Zeit für die nächste Stufe: die Implementierung eines eigenen Kontextes. Als Beispiel soll dafür ein anwendungsweiter Kontext dienen, der seine

Instanzen für einen vordefinierten Zeitraum (z. B. 5 min) hält.[2]

Auch hier wird als erstes wieder eine eigene Annotation benötigt, die Komponenten an den neu zu definierenden Kontext bindet. Da es sich hierbei de facto um die Definition eines neuen *Scopes* handelt, muss diese Annotation zusätzlich mit *@NormalScope* versehen werden.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@NormalScope
  public @interface TimeScoped {
}
```

Wie bereits bei *@Startup* muss nun auch hier noch der Code geschrieben werden, der die eigentliche Funktionalität realisiert. Im Falle eines Kontextes ist dies die Verwaltung der Bean-Instanzen, die ihm zugeordnet werden. Hierzu enthält die CDI-SPI das Interface *javax.enterprise.context.spi.Context*. Dieses Interface definiert vier Methoden:

- **public** <T> T get(Contextual<T> *arg0*)

  Diese Methode muss die im Kontext hinterlegte Bean-Instanz vom Typ *T* zurückliefern. Ist keine Instanz vorhanden, wird null zurückgeliefert. Eine genauere Beschreibung welche Instanz angefragt wird (z. B. deren Bean-Name), wird durch den Übergabeparameter geliefert.

- **public** <T> T get(Contextual<T> *arg0*, CreationalContext<T> *arg1*)

  Diese Methode verhält sich wie *get(Contextual<T>)* mit dem einzigen Unterschied, dass sie die angefragte Instanz bei Bedarf über den mitgelieferten *CreationalContext* erzeugt.

- **public** Class<? **extends** Annotation> getScope()

  Liefert den Typ der *Scope*-Annotation zurück, für den dieser Kontext zuständig ist.

- **public boolean** isActive()

  Gibt an ob der Kontext aktiv ist oder nicht.

```
public class TimedScopedContext
  implements Context
{
  private static class TimeBox<T>{ … }
  Map<String, TimeBox> instances =
    new ConcurrentHashMap<String, TimeBox>();
  public <T> T get(Contextual<T> arg0) {
    TimeBox<T> box =
      instances.get(((Bean<T>)arg0).getName());
    if(box != null && !box.isTimedOut()){
      return (T)box.getBean();
    }
    return null;
  }
  public <T> T get(
    Contextual<T> arg0, CreationalContext<T> arg1
  ) {
    T bean = get(arg0);
    if(bean == null){
      bean = ((Bean<T>)arg0).create(arg1);
      TimeBox<T> box = new TimeBox<T>(bean);
      instances.put(((Bean<T>)arg0).getName(), box);
    }
    return bean;
  }
  public Class<? extends Annotation> getScope() {
    return TimeScoped.class;
  }

  public boolean isActive() {
    return true;
  }
```

Alles was jetzt noch fehlt ist das Bekanntmachen des neuen Kontextes. Hierfür wird wieder eine *Extension* benötigt, die nach der *BeanDiscovery*-Phase (Event: *AfterBeanDiscovery*) eine Instanz des Kontexts erzeugt und dem Container bekannt gibt:

```
public class TimedScopeContextExtension
  implements Extension
{
  public void afterBeanDiscovery(
    @Observes AfterBeanDiscovery event,
    BeanManager manager
  ){
    event.addContext(new TimedScopedContext());
  }
}
```

Auch diese *Extension* muss wieder über den Service Provider Mechanismus registriert werden. Ist dies geschehen, kann der neue *Scope* in der Anwendung verwendet werden.

```
@TimeScoped
@Named
public class TimedScopeSample {...}
```

Entdecke die Möglichkeiten

Die beiden genannten Beispiele zeigen natürlich nur einen kleinen Ausschnitt der Möglichkeiten, die dem Entwickler mit dem *Extension*-Mechanismus an die Hand gegeben werden. Der interessierte Leser sei an dieser Stelle noch einmal auf die Dokumentation der Refe-

[2] Die gewählte Semantik des neuen Kontextes legt keinen Wert auf gesteigerte Sinnhaftigkeit, vielmehr lag das Augenmerk auf Einfacheit bei gleichzeitiger Vollständigkeit der Implementierung (also Verzicht auf eine einfache Erweiterung und Abänderung eines bestehenden Kontextes).

renzimplementierung [3] verwiesen, mit dem speziellen Hinweis auf das *BeanManager-* bzw. das *InjectionTarget-*Interface.

Third Party Extensions

Es müssen aber nicht immer selbstgeschriebene Erweiterungen sein. Auch für *CDI-Extensions* gibt es inzwischen eine ganze Reihe von Anbietern. So bietet z. B. das Apache-Projekt *Deltaspike* [4] eine ganze Sammlung von Erweiterungen an. Diese sind in verschiedenen Modulen organisiert und reichen u. a. von Kernfunktionalitäten (z. B. I18N und *Exception-Handling*) über Themen wie Sicherheit und *BeanValidation-Integration* bis hin zur erweiterten Container-Kontrolle (z. B. explizites Starten und Stoppen von Kontexten im Rahmen von Unit-Tests). Hier finden sich jede Menge funktionale Erweiterungen, die sicher einen Blick Wert sind.

Fazit

CDI bietet mit den *portable Extensions* einen mächtigen Mechanismus zur Erweiterung des „Grundframeworks" an. Damit gibt der Standard dem Entwickler die Möglichkeit Funktionalitäten, die er vermisst, auf eine wohl definierte Art und Weise in das Gesamtsystem zu integrieren.

Dank der – wie immer sehr aktiven – Java-Entwicklergemeinde ist es teilweise nicht einmal mehr notwendig selbst Hand anzulegen. Unter Umständen reicht es bereits aus einfach die passenden *Extensions* zu „er-googeln".

Happy Coding

Insofern gilt ab sofort: nicht über fehlende Funktionalität im Framework schimpfen, sondern einfach passend erweitern! In diesem Sinne: Happy CDI-Coding!

Referenzen

[1] Java Community Process *JSR-299 – Contexts and Dependency Injection (V1.0)*
http://jcp.org/en/jsr/detail?id=299

[2] Java Community Process *JSR-346 – Contexts and Dependency Injection (V1.1)*
http://jcp.org/en/jsr/detail?id=346

[3] SeamFramework.org *Portable Extensions*
http://docs.jboss.org/weld/reference/latest/en-US/html/extend.html

[4] Apache Deltaspike *home*
http://deltaspike.apache.org

Kurzbiographie

Werner Eberling (@Wer_Eb) 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. Er ist u. a. Autor des Buches „Enterprise JavaBeans 3.1 Das EJB-Praxisbuch für Ein- und Umsteiger", erschienen im Hanser Verlag.