jsf-academy.com

JavaServer Faces und CDI

Andy Bosch
JSF-Academy.com
Stuttgart

Schlüsselworte

JavaServer Faces, CDI, JavaEE 6, JBoss Weld, Apache Open WebBeans

Einleitung

JSF und CDI ist das neue Traumpaar der Software-Szene. Während JSF ein klares Bekenntnis auf das User Interface vorweist, ist CDI hier nicht allzu festgelegt, sondern versteht sich eher als allgemeines Komponentenmodell. Ein wichtiges Feature in CDI sind die Kontexte, von denen es etliche mehr gibt, als dies die JSF Spezifikation vorsieht. Speziell die Möglichkeit, Conversations zu verwenden, ist für JSF eine willkommene Ergänzung. Doch nicht nur damit kann CDI bei JSF punkten. Auf die vielen sich ergebenden Vorteile der Kombination wird im Folgenden näher eingegangen.

Die Grundlagen von CDI

Da ich davon ausgehe, dass JSF mittlerweile bekannt sein dürfte (schließlich gibt es das Framework schon ein paar Jahre), gehe ich lediglich auf die Grundlagen von CDI ein, um anschließend verschiedene Szenarien in der Kombination mit JSF aufzuzeigen.

Fangen wir zunächst bei dem Namen an. CDI steht für den Ausdruck "Context and Dependency Injection". Der Name weist direkt auf die zwei Hauptaufgaben dieser neuen Spezifikation hin: Alles hat irgendwie immer mit Kontexten zu tun, und zusätzlich steht noch die Möglichkeit der Dependency Injection zur Verfügung.

Entstanden ist CDI als JSR-299, der ursprünglich unter dem Namen Webbeans firmierte. Im JSR-299 sind viele Ideen eingeflossen, die ursprünglich aus der JBoss Seam Ecke stammten. In JBoss Seam war es damals schon möglich, nicht nur Injection zu betreiben, sondern auch Outjection. Auch andere Frameworks (wie u.a. das Spring Framework) haben sich mit dem Thema DI beschäftigt. CDI hat allerdings den großen Vorteil, zum großen JavaEE Standard zu gehören. Damit muss jeder Application Server, der sich als JavaEE kompatibel bezeichnen möchte, CDI unterstützen.

Erste Schritte in CDI

Schauen wir uns zunächst ein erstes etwas komplexeres HalloWelt Beispiel an. Wir schreiben ein Interface zur Berechnung einer monatlichen Prämie für eine Kfz-Haftpflichtversicherung. Das Interface schaut wie folgt aus:

```
public interface IHaftpflichtCalculator {

   public BigDecimal calculateMonthlyCosts(
```

```
            int carPrice,
           int noOfDrivers,
           boolean carport );

}
```

Fachlich ist das sicherlich ein wenig von der Realität entfernt, aber die lassen wir doch mal außen vor. Wir haben hiermit ein Interface, das eine Berechnung aufgrund einiger Eingabedaten vornimmt. Basierend auf diesem Interface gibt es eine erste Implementierung:

```
public class SimpleHaftpflichtCalculator
             implements IHaftpflichtCalculator {

   @Override
   public BigDecimal calculateMonthlyCosts(
               int carPrice, int noOfDrivers,
               boolean carport) {
      return new BigDecimal(490);
   }
}
```

Ok, auch die Implementierung zeugt wenig von fachlichem Verständnis der Versicherungsbranche, aber das ist ein anderes Thema. Diesen Service möchten wir aus einem UI-Controller heraus aufrufen. Der UI-Controller soll dabei der erste Ansprechpartner sein, wenn aus einer JSF-Seite heraus die Berechnung angestossen wird. Somit ist die JSF-Aktionsmethode, die beispielswiese hinter einem CommandButton liegt, im UI-Controller zu finden. Dieser UI-Controller soll anschließend an den fachlichen Service delegieren.

```
@Named
@RequestScoped
public class KfzUIController {

   @Inject
   private IHaftpflichtCalculator kfzService;

   public String calculateBudget() {
      BigDecimal result = kfzService.calculateMonthlyCosts( 100, 3, true );
      // TODO hier das Ergebnis in das Viewmodell stellen
      return "result";
   }


}
```

Der UI-Controller ist ein erstes einfaches CDI-Bean. Er weist einige typische Eigenschaften auf. Zum einen liegt das Bean in einem Scope, im @RequestScope um genau zu sein. Damit übergeben wir der CDI Runtime die Kontrolle, Objekte unseres UI-Controllers anzulegen und auch wieder zu zerstören. Damit unser CDI Bean später auch über JSF aufrufbar ist (genauer gesagt über die Expression Language), benötigen wir noch eine weitere Annotation: @Named. Damit bekommt das Bean per Konvention den Namen "kfzUIController" und kann so in der JSF-Seite aufgerufen werden. Da wir im UI-Controller den fachlichen Service aufrufen möchten (natürlich ohne harte Abhängigkeiten), lassen wir uns die konkrete Abhängigkeit hereinreichen, also injecten. Dies erfolgt über die Anweisung @Inject. Die CDI Runtime sucht automatisch nach einer passenden Implementierung und reicht uns diese herein. Fertig ist unser CDI-Bean, das auch in JSF verwendet werden kann.

```
<h:form>
   ...
   <h:commandButton value="Berechnen"
       action="#{kfzUIController.calculateBudget}" />
</h:form>
```

In obigem JSF Listing wird direkt die Aktionsmethode im UI-Controller aufgerufen. Dies ist möglich, da der Controller mit @Named in der Expression Language bekannt gemacht wurde.

Bei den Imports sollte beachtet werden, das Package javax.enterprise.context zu erwischen, denn die Namen sind (leider) nicht immer eindeutig. Und einmal das falsche Package erwischt, beschert einem lange Debugging-Nächte.

Einsatz von Qualifiers

Wir haben basierend auf einem Interface eine Implementierung erzeugt. Angenommen, wir wollen eine weitere Implementierung verwenden. Wir hätten dann zwei Implementierungen für das gleiche Interface. Im UI-Controller wird jedoch beim Injection-Point ein Interface-Type verwendet. Somit kann die CDI-Runtime eine eindeutige Zuordnung nicht mehr vornehmen. Dementsprechend erhalten wir beim Start der Anwendung eine Fehlermeldung.

Wie können wir jetzt aber eine Unterscheidung einführen? Die CDI Antwort hier lautet: Qualifiers.

```
@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.FIELD})
public @interface ExtendedCalculator {
}
```

Mit wenigen Zeilen Code können wir so an einer Implementierungsklasse einen Qualifier zusätzlich verwenden. Diesen Qualifier können wir als zusätzliches Unterscheidungsmerkmal betrachten. Beim Injection Point kann dieser Qualifier ebenfalls mit angegeben werden und die CDI Runtime kann eine eindeutige Zuordnung vornehmen.

```
@Inject @ExtendedCalculator
IHaftpflichtCalculator kfzService;
```

Eine Kombination von Qualifiers ist durchaus möglich bzw. oftmals auch sehr sinnvoll.

Stereotypen

Qualifier sind durchaus sehr hilfreich. Doch schnell kommt man an die Stelle, an der "plötzlich" zu viele Annotationen verwendet werden. Hier wäre es sinnvoll, Annotationen zusammenzufassen zu können. Und genau das bietet CDI in Form von Stereotypen. Es gibt bereits Standard-Stereotypen wie z.B. @Model, aber man kann fast genauso leicht eigene Stereotypen bauen.

```
@UIController
public class KfzUIController {
```

Im obigen Beispiel ist unser UI-Controller lediglich mit der Annotation @UIController versehen. Dahinter verbirgt sich ein Stereotype, der neben der @Named Angabe auch gleich noch den Standardscope (@RequestScoped) setzt. Der Scope könnte bei einer Verwendung auch überschrieben werden. Ist jedoch keine Angabe vorhanden, wird der Scope aus dem Stereotype verwendet.

Conversation Scope

Eines der am meisten diskutierten Features, die sich JSF-Entwickler häufig wünschen, ist der Conversation Scope. Die Http-Session ist schon überstrapaziert genug, der RequestScope aber viel zu kurzlebig. Zwar liefert JSF 2 mit dem ViewScope eine ganz nette Alternative, die aber auch nicht immer passend ist. Genau an diesem Punkt bietet CDI einen ConversationScope, der sich nahtlos in JSF integrieren lässt.

```
@Named
@RequestScoped
public class KfzUIController {
   ...
   @Inject
   private Conversation conversation;
   ...
   public void checkConversation( SystemEvent event ) {
      if ( conversation.isTransient() ) {
         conversation.begin();
      }
   }
}
```

In obigem Listing des KfzUIControllers sieht man zunächst, wie die Conversation per Injection in den UI-Controller injiziert wird. Mittels diesem Objekt kann eine neue Conversation gestartet werden. Damit die verwendeten Daten auch tatsächlich pro Conversation abgelegt werden, muss diese Angabe auch an der Klasse hinterlegt sein:

```
@Named
@ConversationScoped
public class HaftpflichtData { ... }
```

Somit lässt sich mit überschaubaren Aufwand der Conversation Scope in JSF ergänzen. Ich denke, dieses Beispiel sollte auch den letzten "Kritiker" überzeugen, dass die Kombination JSF+CDI sehr gewinnbringend ist und sich damit sehr viele Szenarien perfekt abbilden lassen.

Kontaktadresse:

Andy Bosch
JSF-Academy.com
Barchetstraße 19
D-70569 Stuttgart

Telefon:	+49 (0) 711 - 46 97 28 70
Fax:	+49 (0) 711 - 88 25 321
E-Mail	andy.bosch@jsf-academy.com
Internet:	www.jsf-academy.com