Wie man Workflows mit Activiti als Open Source Process Engine leichtgewichtig in die Java-Welt bringt

Standards im Duett: BPMN 2.0 und Java EE 6

Bisher ist uns als Entwickler das Thema BPM allzu oft unter zwei Gesichtspunkten begegnet: Einerseits gab es da das High-Level-PowerPoint-Heile-Welt-BPM auf Businessebene. Andererseits waren die BPM-Plattformen, mit denen wir uns dann am Ende des Tages abquälen, unflexibel und unhandlich. Aber BPM geht auch anders. Wir zeigen im Folgenden, wie sich mit der Open Source Process Engine Activiti einfache Workflows in Java EE 6 umsetzen lassen, mit großem Spaßfaktor bei der Entwicklung und dank BPMN 2.0 trotzdem „Business-aligned".

von Daniel Meyer und Bernd Rücker

In diesem Artikel wollen wir zeigen, wie man mit Activiti BPMN-2.0-Prozesse auf Basis von Java EE 6 umsetzen kann. Dazu haben wir uns als Anwendungsbeispiel die Bearbeitung einer Anfrage für ein Versicherungsangebot herausgesucht. Der Prozess ist in BPMN 2.0 modelliert und in **Abbildung 1** zu sehen. Er beginnt mit dem Ereignis, dass ein Kunde ein Angebot für eine Versicherungspolice beantragt hat. Der nächste Schritt im Prozess ist dann eine Service Task, also eine automatisierte (wir denken: „Java"-)Tätigkeit, in der versucht wird, anhand der vom Kunden erhaltenen Daten ein Angebot zu erstellen. Dem Kunden wird dann die Aufgabe zugeordnet (UserTask), das Angebot zu akzeptieren oder abzulehnen. Mit dieser Aufgabe wird eine Frist verknüpft (siehe „angeheftetes Zeitereignis"). Läuft diese Frist ab, wird dem Kunden die Aufgabe entzogen, und der Prozess wird beendet. Falls der Kunde das Angebot annimmt, wird ein Vertrag erstellt und dem Kunden postalisch zugestellt.

Zur Modellierung des Prozesses lässt sich im Prinzip jedes BPMN-2.0-kompatible Tool verwenden. Das Activiti-Projekt stellt dazu ein eigenes Eclipse Plug-in bereit, den „Activiti Designer". Verwenden wir diesen, entsteht eine Datei mit der Endung *.bpmn20.xml*. Diese XML-Datei ist das ausführbare Artefakt, das wir später an die Process Engine Activiti geben werden. Das Format dieser Datei wird durch den BPMN-2.0-Standard vorgegeben. Bevor wir den Prozess aber von Activiti ausführen lassen können, ist noch einiges an Implementierungsarbeit zu leisten. Wieso? Die Process Engine kann man sich wie eine Art „transaktionalen Zustandsautomaten" vorstellen. Das heißt konkret: Wenn wir den Prozess ausführen, erlaubt Activiti es uns, einzelne Instanzen (= Durchläufe) von dem Prozess zu initiieren, die jeweils mit einer eindeutigen Identifikation (Process Instance ID) versehen werden. Der Zustand dieser Instanzen wird dann von Activiti für uns in einer Datenbank verwaltet. Wenn wir jetzt von Prozessimplementierung reden, dann reden wir von der Entwicklung dessen, was noch darum herum passieren muss, also zum Beispiel Eingabemasken bzw. Formulare, Services usw. Für diese Implementierung haben wir uns einen weiteren Standard ausgesucht, Java Enterprise Edition 6. Wir gehen an dieser Stelle also davon aus, dass der Leser zumindest mit Grundlagen von JSF und CDI vertraut ist.

Los geht's: Wir starten den Prozess mit einem JSF-Formular

Um zu zeigen, wie elegant das geht, steigen wir gleich ein. Wir hatten festgelegt, dass der Prozess durch das Ereignis gestartet wird, dass ein Kunde ein Angebot angefordert hat. Bei der Prozessimplementierung geht es jetzt darum, dieses logische „Ereignis" zu konkretisieren. Eine Möglichkeit wäre es, die benötigten Daten mithilfe einer Eingabemaske vom Kunden abzufragen,

zu sammeln und dann eine neue Instanz des Versicherungsprozesses zu starten, die diese Daten als Eingabe erhält. Wir entscheiden uns, diesen Ansatz mithilfe eines einfachen JSF-Formulars umzusetzen, das in Listing 1 gezeigt ist.

Offensichtlich haben wir die Möglichkeit, Prozessvariablen in einer vorgefertigten CDI Bean namens *processVariables* zu sammeln. Diese Bean wird für uns von der Activiti-CDI-Integration zur Verfügung gestellt. Die Bean ist vom Typ *Map<String,Object>* und erlaubt es uns, Objekte mit einem Namen zu versehen (Key der *Map*) und als Prozessvariablen an Activiti beim Starten einer neuen Prozessinstanz zu übergeben. Prozessvariablen werden dann von Activiti automatisch persistiert und für die Lebensdauer einer Prozessinstanz vorgehalten. Die im Formular gesammelten Daten (z. B. der Kaufpreis des Fahrzeugs) würden also auch den nachfolgenden Tasks im Prozess wieder zur Verfügung stehen und können so z. B. benutzt werden, um das Angebot zu generieren, selbst wenn dies erst Stunden später passieren sollte.

Aber wie kommen die Daten genau in die Prozessinstanz? Der Schlüssel dazu liegt in der letzten EL-Expression in Listing 1. Hier wird eine weitere zur Verfügung gestellte Bean benutzt, die *businessProcess* Bean. Diese erlaubt es uns, neue Prozessinstanzen zu starten und mit bestehenden Instanzen zu interagieren. In diesem Fall starten wir eine neue Instanz des Beispielprozesses, dem wir den Namen *versicherungsabschluss* gegeben haben. Dieser Aufruf startet eine neue Prozessinstanz und übergibt die Variablen, die wir in der *processVariables* Bean gesammelt haben, an diese Instanz. Hinter den Kulissen wird die *processVariables* Bean mit der aktuellen CDI Conversation bzw. dem aktuellen Request verknüpft. Wenn wir also ein mehrseitiges Formular hätten und auf der ersten Seite eine CDI Conversation starten würden, hätten wir die Möglichkeit, über mehrere Requests hinweg Variablen in der *processVariables* Bean zu sammeln, die wir dann auf der letzten Seite an die neu gestartete Instanz übergeben.

Verwendet man kein JSF, kann man natürlich das gleiche Resultat erzielen, wenn man in irgendeiner Weise die Prozessvariablen sammelt und dann den Prozess startet. Wenn man also z. B. ein UI mit HTML, JavaScript und REST bauen möchte, würde man typischerweise die Variablen clientseitig cachen und dann an einen JAX-RS Service übergeben, der sie dann wieder an die entsprechenden CDI Beans übergeben könnte.

Da wir jetzt in der Lage sind, den Prozess zu starten, können wir dazu übergehen, die erste ServiceTask (*Angebot automatisch generieren*) zu implementieren.

Prozess ruft Bean: Wir implementieren eine „ServiceTask“

Der nächste Schritt im Prozess ist die ServiceTask. Eine ServiceTask mit Activiti zu implementieren heißt, dass man die benötigte Funktionalität in Java implementiert und dann den geschriebenen Java-Code von Activiti aufrufen lassen kann. Dazu gibt es verschiedene Möglichkeiten. Wir können z. B. ein von Activiti bereitgestelltes Interface (*JavaDelegate*) implementieren und unsere Klasse von Activiti instanziieren lassen. Eine andere Möglichkeit ist es, direkt eine CDI Bean aufrufen zu lassen (Listing 2).

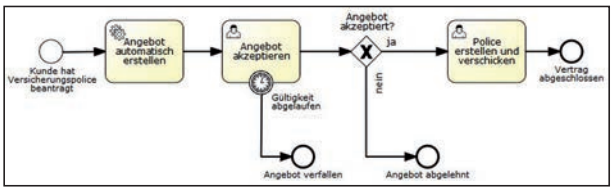

Abb. 1: Ein einfacher Prozess in BPMN 2.0 modelliert

Das Erste, was auffällt, ist, dass man mit Activiti aus BPMN-Prozessen auf die gleiche Art und Weise CDI Beans aufrufen kann wie aus JSF. Activiti erlaubt es uns, im BPMN 2.0 XML Erweiterungen im Activiti

Listing 1

```
<h1>Antrag fuer die Kraftfahrtversicherung</h1>
<h:form>
 <table>
  <tr>
   <td>Ihr Vorname:</td>
   <td><h:inputText value="#{processVariables['vorname']}" /></td>
  </tr>
  <tr>
   <td>Ihr Nachname:</td>
   <td><h:inputText value="#{processVariables['nachname']}" /></td>
  </tr>
  <tr>
   <td>Baujahr des Fahrzeuges:</td>
   <td>
    <h:inputText value="#{processVariables['kfzBaujahr']}">
     <f:convertDateTime pattern="dd/MM/yyyy"/>
    </h:inputText> (TT/MM/JJJJ)
   </td>
  </tr>
[...]
  <tr>
   <td />
   <td><h:commandButton value="Fertig"
    action="#{businessProcess.startProcessByKey('versicherungsabschluss')}" />
   </td>
  </tr>
 </table>
</h:form>
```

Listing 2

```
<serviceTask id="angebotErstellen"
    activiti:expression="${angebotsService.angebotErstellen()}"
    activiti:async="true" name="Angebot automatisch erstellen">
</serviceTask>
```

Namespace zu benutzen. Hier können wir dann den EL-Namen einer *@Named* CDI Bean angeben. Die in Listing 3 gezeigte Java-Klasse ist dann die CDI Bean, die von Activiti aufgerufen wird. Wenn wir uns diese Bean anschauen, wird auch augenblicklich klar, wieso das eine gute Sache ist. Da wir hier eine Java-EE-Komponente in der Hand haben, können wir natürlich auch alle Dienste nutzen, die die Java EE Platform uns bietet, z. B. Dependency Injection, Interceptors, JPA oder auch EJBs.

Hier finden wir jetzt auch wieder die *processVariables* Bean wieder. Mithilfe dieser Bean können wir jetzt auf die Prozessvariablen, die wir beim Prozessstart übergeben haben, zugreifen und weitere hinzufügen. Wir hätten zusätzlich die Möglichkeit, uns bestimme Prozessvariablen mithilfe eines CDI Qualifiers direkt injizieren zu lassen:

```
@Inject @ProcessVariable Object firstname;
```

Oder wir könnten auch an die aufgerufene Methode eine Instanz von dem von Activiti zur Verfügung gestellten *DelegateExecution* als Parameter übergeben lassen. Der Charme der *processVariables* Bean besteht darin, dass wir keine zusätzlichen Activiti Imports brauchen, unsere Klassen also nicht von Activiti abhängig sind und es sich für Unit Tests gut mocken lässt.

Wie wir wissen, können CDI Beans gewissen Scopes zugeordnet werden. Die in Listing 3 gezeigte Bean hat keinen Scope ist somit implizit *@Dependent*. Das stellt sicher, dass für jeden Aufruf der Bean eine neue Instanz erzeugt wird und wir uns um Themen wie Thread-Sicherheit keine Gedanken machen müssen. In diesem Fall würde jede Instanz unseres Prozesses ihr eigenes Exemplar der Bean bekommen. Es gibt jedoch auch Situationen, in denen man auf globalem Zustand arbeiten will. Hier könnte man eine *@ApplicationScoped* Bean benutzen, oder, weil EJBs ja auch CDI Beans sind, eine *@Singleton* EJB mit Concurrency Control für Thread-Sicherheit. Die gute Nachricht ist also, dass man mit Blick auf den Prozess die Mächtigkeit des Java-EE-Programmiermodells zur Verfügung hat, auf diese Art und Weise viele Probleme elegant lösen kann und vor allem kein proprietäres Activiti-Know-how erlernen muss.

Unter der Haube geht's mal wieder um Transaktionen

Die in Listing 3 gezeigte Bean lässt aber noch auf eine tiefergehende Integration hindeuten. Wer genau hinschaut, sieht, dass wir hier mit einem JPA Entity Manager ein neues Angebot persistieren, und weiß, dass wir dafür eine Transaktion brauchen. Da es sich bei der in Listing 3 gezeigten Bean um eine einfache Managed Bean handelt (also nicht um eine EJB), wird hier keine Transaktion für uns aufgemacht. Es existiert allerdings trotzdem eine, und die wird von der Prozessmaschine kontrolliert.

Activiti selbst benötigt Transaktionen, um den Zustand der Prozessinstanzen und -variablen in einer Datenbank zu persistieren. Im Rahmen einer solchen Transaktion führt Activiti auch die ServiceTask aus und ruft synchron unsere Bean auf. Da es sich dabei um eine über den Application Server (oder genauer JTA) gesteuerte Transaktion handelt, kann JPA an der gleichen Transaktion teilnehmen. Dadurch erfüllen wir quasi automatisch Konsistenzanforderungen zwischen Prozesszustand und Anwendungsdaten (in diesem Fall das Angebot), denn wenn Activiti seine Transaktion aus irgendeinem Grund nicht erfolgreich beenden kann, wird auch das Angebot nicht persistiert. Das Gleiche gilt übrigens auch anders herum: Wenn wir in der Applikation, die Activiti aufruft, bereits eine Transaktion geöffnet haben, nimmt Activiti an dieser Transaktion auch teil. Bräuchten wir mehr Flexibilität, hilft der Griff in den Java-EE-Werkzeugkasten, z. B. die *@TransactionAttribute(REQUIRES_NEW)*-Annotation aus EJB.

Das erklärt auch, wieso wir eine einfache CDI Bean für die ServiceTask verwenden können und nicht notwendigerweise eine EJB brauchen. Prozess-Engines,

Die Zukunft liegt im „Less Code BPM"

In der Welt des Business Process Management (BPM) geistert häufig der Begriff „Zero Coding" herum. Gemeint ist damit, dass man keine Entwickler mehr benötigt, um einen Prozess zu automatisieren. Erfahrungsgemäß funktioniert dies allerdings in komplexen Szenarien nicht. Soll beispielsweise eine Anforderung in der Oberfläche umgesetzt werden, die der eingebaute Formulardesigner nicht unterstützt, so kann man anfangen, nach hässlichen Workarounds zu suchen.

Daher propagieren wir schon länger einen anderen Ansatz, den wir als „Less Coding" bezeichnen: Die Entwickler bekommen die Umgebung, in der sie maximal produktiv sind, und wir bringen „lediglich" die Prozesse in diese Welt hinein. Dieser Artikel zeigt, wie einfach dies mit Activiti und Java EE 6 funktioniert, wobei die leichtgewichtige Java Process Engine noch weitaus mehr Trümpfe für Java-Entwickler aufzubieten hat: einfache Testbarkeit, Integrierbarkeit auch in Spring oder OSGi und Wahlfreiheit der Oberfläche, sodass auch GWT, JSP, Swing oder Portlets zum Einsatz kommen können.

Es müssen also keine Aspekte der Prozesssteuerung manuell geschrieben werden, und man kann auf vorgefertigte Funktionalität zurückgreifen, wie beispielsweise langlaufende Prozesse, Versionierung, Aufgabenmanagement oder Eskalation und Zeitereignisse. Es stehen einem alle Möglichkeiten der Java-Welt offen, und man kann auf einen großen Pool von Java-Profis zurückgreifen. Und dank der Mächtigkeit von Java EE 6 können in der Kombination mit BPMN, Activiti und camunda fox maximal produktiv Prozesse automatisiert werden. Ohne Zero-Code, aber dafür funktionierend.

die auf diese Art und Weise an den Applikationstransaktionen teilnehmen können, nennen wir „Embeddable Process Engines“.

Asynchronität

Ein letztes Detail aus Listing 2 soll nicht unerwähnt bleiben: *activiti:async="true"*. Was wir hier machen, nennt sich „Asynchronous Continuation“. Das Konzept ist einfacher als der Name: Wir weisen Activiti an, vor der Ausführung der ServiceTask einen „Safepoint“ zu erstellen, also die Transaktion zu committen, den aktuellen Thread freizugeben und asynchron in einem anderen Thread weiterzumachen. Wir erinnern uns noch einmal an Listing 1: Hier wurde die Prozessinstanz gestartet. In puncto Thread heißt das: Der HTTP Thread, der den JSF Request vom Browser entgegennimmt, wird von Activiti benutzt, um die Prozessinstanz zu starten. Activiti führt dann aber die nachfolgende ServiceTask nicht mehr in diesem Thread aus. Ohne *async* würde auch die ServiceTask in diesem Thread ausgeführt werden.

Wieso wollen wir das? Meistens aus Gründen der Fehlertoleranz und Skalierung. Diese Strategie erlaubt es uns, einerseits Anfragen des Clients (in diesem Fall der Browser) schneller zu beantworten und besser mit Lastspitzen sowie Ausfällen von Systemen umzugehen. Da die tatsächliche Bearbeitung der Anfragen im Hintergrund passiert, können wir bei Lastspitzen über die Datenbank Anfragen sehr einfach puffern und automatisch zu einem späteren Zeitpunkt bearbeiten. Andererseits können wir nun anders mit Fehlern umgehen. Falls bei der Bearbeitung der Anfrage Fehler auftreten, greift ein automatischer Retry-Mechanismus von Activiti, und wir versuchen es nach einer gewissen Zeit noch einmal. Dieses Konzept ist vergleichbar mit dem transaktionalen Zustellen einer Message in JMS. Das Ganze über die Prozess-Engine zu erledigen, hat allerdings den Vorteil, dass die so genannten Jobs (= Messages) im Monitoring immer eindeutig einer Prozessinstanz zugeordnet und bei Fehlern einfacher auf den fachlichen Vorgang gemappt werden können.

Nochmal JSF: wir implementieren eine „UserTask“

Der BPMN-2.0-Standard erlaubt es, Aufgaben zu definieren, die menschlichen Teilnehmern zugeordnet werden. Das grundlegende Konzept ist immer gleich: Es existieren Aufgabenlisten, die von der Process Engine verwaltet werden und aus denen wir uns eine Aufgabe herauspicken können, die wir bearbeiten und dann abschließen.

In diesem Fall wollen wir die UserTask *angebotAkzeptieren* dem Kunden zuordnen, der die Anfrage gestellt hat. In einem echten Szenario würden wir uns dazu zum Beispiel den angemeldeten Benutzer aus JAAS merken und dann mit diesen Informationen die Task-Listen aufbauen. In diesem Beispiel nehmen wir der Einfachheit halber an, dass Vor- und Nachname den Kunden eindeutig identifizieren. Die entsprechende UserTask sieht dann in BPMN-2.0-Notation wie in Listing 4 gezeigt aus.

Listing 3

```
@Named
public class AngebotsService {

  @PersistenceContext
  private EntityManager em;

  @Inject @Named
  private Map<String, Object> processVariables;

  public void angebotErstellen() {
    Angebot angebot = new Angebot();
    angebot.setFirstname((String) processVariables.get("vorname"));
    angebot.setLastname((String) processVariables.get("lastname"));
    // berechne Monatsbeitrag (vielleicht irgendwann mit Drools)
    BigDecimal montasbeitrag = new BigDecimal(100);
    angebot.setMonatsbeitrag(montasbeitrag);

    em.persist(angebot);
    em.flush();
    processVariables.put("angebotsId", angebot.getId());
  }

  @Produces @Named
  public Angebot angebot() {
    return em.find(Angebot.class, processVariables.get("angebotsId"));
  }
}
```

Listing 4

```
<userTask id="angebotAkzeptieren"
activiti:assignee="${vorname}${' '}${nachname}"
activiti:formKey="angebotAkzeptieren.jsf"
name="Angebot akzeptieren" />
```

Listing 5

```
@Named
public class TaskListController {

  @Inject
  private TaskService taskService;

  public List<Task> getTaskList(String user) {
    return taskService.createTaskQuery()
      .taskAssignee(user)
      .list();
  }
}
```

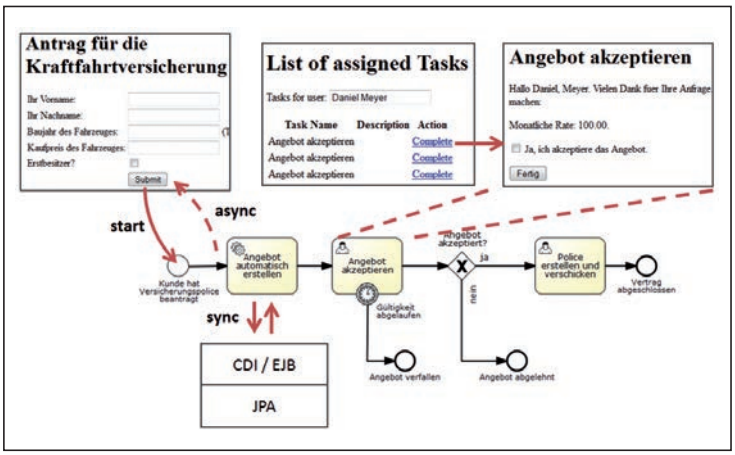

Abb. 2: Prozess und Implementierungsartfakte

Listing 6

```
<h:dataTable
  value="#{taskListController.getTaskList(currentUser.username)}" var="task">
  <h:column>
    <f:facet name="header">Task Name</f:facet>
    #{task.name}
  </h:column>
  <h:column>
    <f:facet name="header">Description</f:facet>
    #{task.description}
  </h:column>
  <h:column>
    <f:facet name="header">Action</f:facet>
    <h:outputLink value="#{formService.getTaskFormData(task.id).formKey}">
      Complete
      <f:param name="taskId" value="#{task.id}"></f:param>
    </h:outputLink>
  </h:column>
</h:dataTable>
```

Listing 7

```
<f:metadata>
  <f:viewParam name="taskId" />
  <f:event type="preRenderView"
        listener="#{businessProcess.startTask(taskId, true)}" />
</f:metadata>
[...]

<h1>#{task.name}</h1>
Hallo #{processVariables['vorname']}, #{processVariables['nachname']}.
Vielen Dank fuer Ihre Anfrage. Wir koennen Ihnen folgendes Angebot machen:
<p>Monatliche Rate: #{angebot.monatsbeitrag}.<p>
<h:form>
  <h:selectBooleanCheckbox value="#{processVariables['angebotAkzeptiert']}" />
  Ja, ich akzeptiere das Angebot.
  <p/>
  <h:commandButton value="Fertig" action="#{businessProcess.completeTask(true)}" />
</h:form>
```

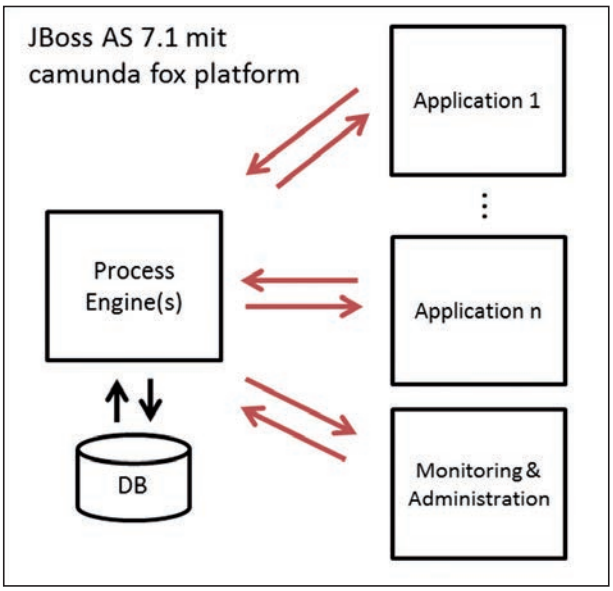

Abb. 3: Activiti als zentrale Komponente im JBoss AS 7

Activiti stellt zum Zugriff auf die Engine diverse APIs zur Verfügung. Eines davon ist der TaskService zur Verwaltung von Aufgaben. Dieser erlaubt es uns, eine Aufgabenliste mithilfe einer einfachen Query abzufragen (Listing 5).

Jetzt können wir die Liste mithilfe von JSF ausgeben, wie in Listing 6 gezeigt. Jede Task enthält eine Zeile in der Tabelle. In der letzten Spalte generieren wir jeweils einen Link, der zum Task-Formular führt. Das Task-Formular ist dann die wirkliche „Implementierung" der UserTasks. Listing 7 zeigt, wie das aussehen könnte. Bevor wir die Seite darstellen, lesen wir die Task-ID aus dem Request URL. Diesen Wert übergeben wir dann wieder an die *businessProcess* Bean. Ab dem Zeitpunkt merkt sich diese Bean, dass wir in der aktuellen Konversation an dieser bestimmten Task arbeiten. Das erlaubt es uns auch, wieder mithilfe der *processVariables* Bean auf die Prozessvariablen zuzugreifen.

Und woher kommt das Angebot? Also wieso funktioniert *#{angebot.monatsbeitrag}*? Wir nehmen noch einmal unser komplettes CDI-Wissen zusammen und schauen in Listing 3: Hier existiert ein CDI Producer, der aus den Prozessvariablen die Angebots-ID liest und damit die JPA-Entity aus der Datenbank lädt.

Soweit unsere kleine Einführung. **Abbildung 2** zeigt die bisher besprochenen Artefakte im Überblick. Wir wollen uns jetzt noch damit beschäftigen, wie man ein Projekt mit Activiti und Java EE 6 aufsetzt.

Projektsetup mit Activiti und camunda fox

Als „Embeddable Engine" kann Activiti sehr eng mit einer Anwendung integriert werden. Konkret haben wir gesehen, dass die Engine an Transaktionen teilnehmen oder auf andere Ressourcen der Anwendung wie CDI Beans zugreifen darf. Das hat den Vorteil, dass man die Infrastruktur sehr schlank und transparent halten kann. Die in diesem Artikel besprochene Anwendung kann so z. B. einfach als WAR-Archiv gepackt und zu-

sammen mit Activiti deployt werden. Das bedeutet, dass ein *activiti-engine.jar* im *WEB-INF/lib*-Ordner liegt, zusammen mit dem *activiti-cdi.jar*, das die Java-EE-6-Integration herstellt. Die Prozess-Engine kann dann wie andere Libraries auch in der Anwendung direkt benutzt werden. Zwei einfache Maven Dependencies und etwas Konfiguration sind bereits ausreichend. Wie das genau aussieht, kann im den Artikel begleitenden Beispielprojekt nachgesehen werden, wenn man das richtige Maven-Profil auswählt.

Dieser schnelle und einfache Weg hat jedoch auch Nachteile: Stellen wir uns vor, dass wir mehrere solcher Prozessanwendungen entwickeln. Dann möchten wir einerseits nicht in jeder Anwendung eine eigene Prozess-Engine (mit eigener Datenbank) mitbringen. Aber vor allem möchten wir andererseits vermutlich anwendungsübergreifendes Monitoring oder eine globale Task-Liste für alle Prozessanwendungen realisieren. Daher haben wir in der „camunda fox platform" einen alternativen Weg eingeschlagen: Anstatt Activiti mit jeder Anwendung separat zu deployen, installieren wir die Process Engine zentral im Application Server, sodass sie allen Applikationen als Infrastrukturkomponente zu Verfügung steht, wie in **Abbildung 3** verdeutlicht. Auf diese Weise können wir zum zentralen Monitoring und Controlling einfach eine weitere Anwendung deployen, die sich mit der Process Engine verbindet. Der Clou ist aber, dass sich für die Prozessanwendung quasi nichts ändert, denn die Infrastruktur ist so gestrickt, dass Activiti, wenn nötig, in den Kontext der entsprechenden Prozessanwendung wechselt, womit der Zugriff auf lokale Ressourcen wie CDI Beans funktioniert, aber auch z. B. Sicherheitsmechanismen von Java EE normal greifen.

Mit der camunda fox platform stellen wir eine freie Version dieser Integration in den JBoss AS 7.1 bereit. Kommerziellen Support gibt es dann auch für weitere Java EE 6 Application Server wie Glassfish 3.1, WebSphere 8 oder WebLogic 12c. Das in diesem Artikel beschriebene Beispiel kann also als *war*-Archiv ohne Activiti im Bauch direkt auf dem vorbereiteten JBoss 7 deployt werden (Kasten: „Loslegen!"). Übrigens stehen sowohl Activiti als auch die camunda fox community edition unter der Apache-Open-Source-Lizenz, was eine einfache Verwendung in verschiedensten Szenarien ermöglicht.

Loslegen!

1. camunda fox platform community edition herunterladen [1], entpacken und den enthaltenen JBoss 7 hochfahren (*server/jboss-as-*/bin/standalone.bat* oder *.sh*); es wird standardmäßig die integrierte H2-Datenbank verwendet
2. Activiti Designer Eclipse Plug-in über Update-Seite [2] installieren oder Evaluierungsversion des leistungsfähigeren fox designer besorgen [3]
3. Beispiel zum Artikel aus dem Demo-SVN auschecken [4]
4. Projekt mit Maven bauen und entstandenes WAR in den *standalone/deployment*-Ordner des JBoss 7.1 kopieren
5. Die Anwendung steht unter [5] zur Verfügung
6. Weitere Infos im Activiti User Guide [6]

Fazit

Es bewegt sich viel an der BPM- und Java-(EE-)Front. Die Integrationen sind ausgereift und erlauben es, zügig Anwendungen zu entwickeln, wobei man hauptsächlich „normales" Java-Know-how benötigt. Die Open-Source-Projekte erlauben es, direkt loszulegen, wobei trotzdem kommerzieller Support und zusätzliches Tooling für den Enterprise-Einsatz verfügbar sind. Dieser Weg, Prozesse zu automatisieren, zeigt sich als in der Praxis sehr erfolgreich, da auch die Akzeptanz bei Java-Entwicklern hoch ist. Er macht einfach Spaß. Trotzdem kann man mit BPMN einen Schritt Richtung Business-IT-Alignment machen, auch wenn das noch weitere Überlegungen erfordert (siehe dazu auch [7]). Übrigens haben wir aufgrund des begrenzten Platzes darauf verzichtet, auf SOA als Anwendungsfall einzugehen. Auch das kann in dieser Umgebung grundsätzlich praktiziert werden, was wir uns aber besser für einen zukünftigen Artikel aufheben.

Bernd Rücker (bernd.ruecker@camunda.com) ist einer der Gründer von camunda und verfügt über umfangreiche Projekterfahrung im Bereich BPM sowie Java EE. Bernd ist langjähriger Committer im jBPM- sowie Activiti-Projekt und bloggt unter www.bpm-guide.de.

Daniel Meyer (daniel.meyer@camunda.com) arbeitet als Berater, Trainer und Entwickler bei der camunda und ist Teil des Activiti-Core-Entwicklerteams und Lead der Activiti-Java-EE-6-Integration.

Links & Literatur

[1] http://www.camunda.com/fox/community/download/

[2] http://activiti.org/designer/update/

[3] http://www.camunda.com/fox/

[4] https://svn.camunda.com/fox/demo/fox/versicherungsantrag/

[5] http://localhost:8080/versicherungsantrag/

[6] http://www.activiti.org/userguide/

[7] Jakob Freund, Bernd Rücker: „Praxishandbuch BPMN 2.0", München: Carl Hanser, 2010