Java EE meets Microservices ▸ 32

JAVA Mag

Deutschland €9,80 Österreich €10,80 Schweiz sFr 19,50 Luxemburg €11,15 4.2016

Java magazin

Java • Architekturen • Web • Agile

www.javamagazin.de

BPM

Kann mehr, als man denkt

BPM abgespeckt – ein Praxisbericht

Vom gescheiterten BPMS zum Open-Source-Eigenbau

Business Process Management (BPM) ist schon lange kein Hype mehr, sondern zählt in vielen Softwareentwicklungsabteilungen bereits zum täglich Brot. Nur leider haben sich viele BPM-Systeme am Markt etabliert, die den modellgetriebenen Ansatz nicht nur für die Prozessmodelle verfolgen, sondern auch für den Rest der Geschäftslogik. Dieser „Zero-Code-Ansatz" mag in manchen Fällen gut funktionieren, jeder Entwickler weiß aber, dass ab einem bestimmten Komplexitätsgrad schnell die eine oder andere Codezeile notwendig wird. Doch es können auch noch ganz andere Probleme auftreten.

von Michael Scholz

Wir befinden uns in der IT-Abteilung eines deutschen Automobilzulieferers vor einigen Jahren. Zur Erarbeitung der IT-Strategie für die nächsten Jahre wurde ein großes Beratungshaus konsultiert. Eine der Empfehlungen lautete: Automatisierung von Geschäftsprozessen mit BPM. Zur Umsetzung wurden drei kommerzielle BPM-Systeme (BPMS) als geeignet und strategisch sinnvoll empfohlen. Eines davon wurde danach per PoC (Proof of Concept) und Entscheidungsmatrix sorgfältig ausgewählt. Die ersten umzusetzenden Prozesse stammten aus dem Bereich Materialwirtschaft und überspannten mehrere Fachbereiche und IT-Systeme (BS2000 Mainframe, E-Sourcing und MDM sowie Engineering-Systeme). Mit der Entwicklung wurde ein Entwicklerteam beauftragt – zusammengesetzt aus Consultants des Herstellers und auf Kundenseite dem Autor des Artikels, der bisher hauptsächlich Erfahrungen mit Java (EE) hatte. Später wurden weitere interne Kollegen hinzugezogen, um für einen langsamen Wissenstransfer zu sorgen und den Consultingaufwand schrittweise zurückfahren zu können. Unglücklicherweise kamen mit diesem System nach und nach immer mehr Probleme auf. Deren übergreifende Ursache war letztendlich die Komplexität der Gesamtlösung.

Die Probleme häufen sich

Das gekaufte BPMS war über Jahre gewachsen und aus verschiedenen Einzelprodukten und Komponenten zusammengeschweißt: einer Prozess-Engine, einem System zur Aufgabenverwaltung, einem Enterprise Service Bus, einem Portal und weiteren. Natürlich waren diese Komponenten miteinander auf verschiedene Art und Weise über diverse Schnittstellen und Technologien verbunden. Doch es fehlte ein zentrales Konzept. Das führte einerseits zu einem wirren Konfigurationsdschungel und andererseits oftmals zu Inkompatibilitäten nach dem Einspielen von Patches oder Updates für einzelne Komponenten oder auch des Gesamtsystems. Das Patchmanagement musste deshalb schließlich sogar per Outtasking dem Hersteller übergeben werden.

Nahezu jede Komponente des Stacks arbeitete mit eigenen Entwicklungstools, die teils unterschiedliche Ideen bezüglich Coding, Testen, Versionierung, Build Management und Deployment verfolgten. Einige Tools erlaubten nur „Live Editing" auf einem gemeinsamen

Development-Server – es gab also keine isolierte Umgebung für den einzelnen Entwickler. Das ließ natürlich kaum einen kontrollierten Entwicklungsprozess zu. Eine gemeinsame Versionierung oder ein gemeinsamer Build der diversen Einzelartefakte war ebenso nicht oder nur eingeschränkt möglich, was jeden Test oder Go-Live immer wieder zu einer eigenen Herausforderung machte.

Die entkoppelte Systemarchitektur und der Mangel an entsprechendem Tooling machte es den Entwicklern schwer, die Auswirkungen ihrer Codeänderungen abzuschätzen. Dementsprechend mühsam gestalteten sich Änderungen an bestehendem Code. Und um so etwas wie Refactorings versuchte man einen weiten Bogen zu machen. Zwar hätte man z. B. im ESB teilweise auf Ebene der einzelnen Artefakte mit Versionierung arbeiten, also alle Änderungen einfach in einen Klon der alten Logik mit höherer Versionsnummer einbauen können, doch auch hier war die Toolunterstützung für die Verwaltung der dadurch entstehenden Abhängigkeiten so schwach ausgeprägt, dass dies mit wachsender Codebasis zu einer nicht mehr überschaubaren oder wartbaren Anwendungsarchitektur geführt hätte.

Auch automatisiertes Testen war technisch unmöglich – es konnten also kein Continuous-Integration-Server oder andere Tools zur Testautomatisierung genutzt werden. Die einzige Möglichkeit waren manuelle Tests auf den Development- und Staging-Systemen sowie Smoke-Tests nach jedem Live Deployment – mit gekreuzten Fingern.

Neben allen technischen Belangen sollte natürlich auch erwähnt werden, dass durch hohe Lizenz-, Wartungs- und, bedingt durch die zuvor genannten Probleme, Entwicklungskosten ein positiver ROI auch nach zwei Jahren noch in weiter Ferne lag.

Eine neue Lösung muss her

Aufgrund dieser immerwährenden Probleme entschied sich das Unternehmen nach etwa zwei Jahren, nochmal von vorne zu beginnen. Es wurde ein komplett neuer Auswahlprozess mit drei Kandidaten gestartet. Aufgrund der Empfehlungen des bestehenden BPM-Entwicklerteams war darunter auch ein Open-Source-basierter Vertreter neben zwei klassisch kommerziellen. Der Open-Source-Kandidat war allerdings kein einzelnes Produkt, stattdessen wurde nach kurzer Marktstudie durch den Autor des Artikels ein Stack aus Komponenten ins Rennen geschickt.

- JBoss Enterprise Application Platform 6 (basierend auf JBoss 7 Application Server)
- Liferay Portal Server 6 EE (deployt im JBoss)
- Camunda BPM Platform 7 EE (integriert in JBoss)
- Eclipse IDE als Entwicklungstool
- Best-of-Breed-Open-Source-BPM-Stack

Wie zu sehen, wurde bei allen Serverkomponenten die „Enterprise Edition“ ausgewählt, um kommerziellen Support zur Verfügung zu haben. Das war ein wichtiges Argument für das Management und eine wichtige Absicherung für das Entwicklerteam, das übrigens inzwischen ausschließlich aus internen Mitarbeitern bestand. Dennoch sind natürlich jeweils auch kostenfreie Communityversionen verfügbar, die sich ideal zum Kennenlernen und Herumspielen eignen – so geschehen bei der Zusammenstellung des Stacks.

Nachdem es nun ausreichend BPM-Erfahrung mitbringen konnte, erstellte und verwendete das BPM-Entwicklerteam für die Evaluierung eine sehr feingranulare, gewichtete Entscheidungsmatrix mit über 100 Kriterien, um den besten Kandidaten auszuwählen. Der Vergleich erfolgte über inhouse durchgeführte PoC-Sessions mit den Anbietern oder Consultingpartnern, die alle denselben BPM-Referenzprozess inklusive webbasierter Oberflächen implementieren sollten. Die Evaluierung ergab schlussendlich, dass der Open-Source-Kandidat den besten Erfüllungsgrad vorweise konnte – sowohl aus funktionaler als auch archi-

Praktisch überall, wo das alte System Schwächen zeigte, konnte die neue Open-Source-Lösung ordentlich punkten.

tekturaler Sicht. Aufgrund der erwähnten Fokussierung auf Enterprise Subscriptions stimmte das IT-Management schließlich der Entscheidung zu.

Der Firma gelang es in nur etwa drei Monaten, die gesamte neue Umgebung aufzusetzen und einen ersten BPM-Prozess von der alten Plattform zu migrieren und live zu bringen. Während dieser Zeit wurde außerdem ein kleines Framework zur Entwicklung von Prozessapplikationen auf der neuen Plattform geschaffen, ein Template für entsprechende Entwicklungsprojekte sowie eine Dokumentation mit Programmiermodell und Best Practices für alle BPM-Entwickler.

Die Resonanz von Entwicklern und dem Artikelautor war durchwegs positiv – alle waren sehr zufrieden mit ihrer maßgeschneiderten und produktiven Entwicklungsumgebung. Die Plattform war leichtgewichtig, erlaubte einen schnellen Einstieg und hat sich bis heute bewährt. Praktisch überall, wo das alte System Schwächen zeigte, konnte die neue Lösung ordentlich punkten. Darüber hinaus gab es allerdings noch einige weitere Vorteile.

Integrierte und leichtgewichtige Komponentenarchitektur

Alle Komponenten des neuen Systems, wie Prozess-Engine, Portal oder Applikationen, sind ausschließlich per Java implementiert. Das war zuvor nur teilweise der Fall. Und sie laufen gemeinsam in einem Java EE Application Server. Dadurch können alle direkt über die JVM miteinander kommunizieren, es gibt keine externen Schnittstellen mehr oder gar unterschiedliche Protokolle oder Technologien. Der vorherige Konfigurationsdschungel, über den die einzelnen Komponenten zusammengeschaltet wurden, ist komplett obsolet geworden. Und durch ein gemeinsames Abhängigkeitsmanagement über Maven sind Updates und Patches der Plattform oftmals nicht mehr als eine Minutensache.

State-of-the-Art-Entwicklungsprozess

Jeder Entwickler kann nun einfach seine eigene Installation isoliert auf seiner Maschine betreiben. Deren Setup erfolgt automatisch über einfache Scripts. Es besteht kein Zwang mehr, auf zentralen gemeinsamen Systemen zu entwickeln und zu testen. Dadurch wird Code komplett lokal entwickelt, nach Fertigstellung ins SCM eingecheckt und ein dahinter geschalteter Continuous-Integration-Server fährt laufend Builds und informiert Entwickler zeitnah über Fehler. Es gibt eine zentral verwaltete Eclipse-Installation, die mit nur wenigen Plugins aufgerüstet ist und es den Entwicklern erlaubt, alle Artefakte von BPM-Prozessapplikationen, wie Prozesse, UIs, Business oder Logik, über dasselbe Tool zu erstellen und zu pflegen.

Komfortables Dependency Management

Die Nutzbarkeit des Java-Compilers und die üblichen Analyse- und Refactoring-Möglichkeiten von Eclipse erlauben den Entwicklern nun das schmerzfreie Durchführen von Änderungen und Refactoring und einen schnellen Überblick über mögliche Auswirkungen. Das Risiko von Seiteneffekten wurde minimiert, die Qualität damit deutlich erhöht und die Aufwände für Entwicklung und Testing stark reduziert. Die Maven-basierte Projektverwaltung und die Möglichkeiten des isolierten Class Loadings im Java-EE-Server reduzierten die Komplexität bei der Verwaltung von Abhängigkeiten beim Build und Deployment von Anwendungen noch weiter.

Hohe Testautomatisierung

Tests können nun einfach auf verschiedenen Ebenen implementiert werden – vom Unit Test zum automatisierten Integrationstest bis zum manuellen End-to-end-Test. Die Möglichkeit der Nutzung moderner Testing-Frameworks, wie *Unit, Arquillian und Mockito sowie die Möglichkeit, die Camunda Engine auch In-Memory betreiben zu können, erlauben es nun, BPMN-Prozesse komplett automatisiert zu testen.

Finanzielle Aspekte

Wie schon erwähnt, wurden alle verwendeten Komponenten in der Enterprise-Variante ausgewählt, um entsprechenden Support und auch das eine oder andere Zusatzfeature zur Verfügung zu haben. Obwohl also auch hier natürlich entsprechende laufende Kosten anfallen, ließ sich der Gesamtkostenaufwand gegenüber dem Vorgängersystem deutlich reduzieren. Die technische Reife der neuen Lösung und die Möglichkeit, gewohnte Entwicklungstools und -prozesse nutzen zu können, konnten außerdem auch die Entwicklungskosten verringern.

Wiederwendung von Wissen

Die Entwicklung einer BPM-Prozessapplikation unterscheidet sich nun kaum mehr von der Entwicklung einer normalen Java-Enterprise-Applikation. So konnte das im Team vorhandene Wissen in hohem Maße wiederverwendet werden. Zugleich hatten neue Teammitglieder eine flache Lernkurve und mussten nur ein paar wenige Dinge komplett neu dazulernen. Schließlich ist das notwendige Wissen generell besser zugänglich, da mit Java EE ein weit verbreiteter Standard genutzt wird und sowohl Camunda als auch Liferay beide eine starke Community haben. Früher musste bei fast allen fortgeschrittenen Problemen erst Consulting ins Haus geholt werden.

Hoher Reifegrad der genutzten Technologien

Eben wegen der genannten Gründe ist für die genutzten Technologien Wissen nicht nur schnell und einfach verfügbar, ebenso ist der Reifegrad dieser Technologien hoch. Dies mag sicher an den großen Communitys oder der Vielzahl an Kunden und Herstellern liegen, die alle Einfluss auf die Entwicklung der Produkte nehmen.

Nicht zufällig konnte sich der Open-Source-basierte Stack in der Evaluierung technisch gegenüber den rein kommerziellen Lösungen hervorheben. Da es auf einem Standard-Java-Application-Server arbeitet, kann das Entwicklerteam beispielsweise nun auch das gesamte Spektrum an Technologien des Java-EE-Standards nutzen, wie ORM-basierte Persistenz (JPA), einfache oder automatische Verwaltung von Transaktionen und Connections (JTA und JCA), Dependency Injection (CDI), Messaging (JMS), XML-Binding, Web Services und REST Services (JAXB, JAX-WS und JAX-RS) sowie einige mehr.

Die alte Suite stellte für viele dieser Anforderungen keine Lösung bereits. ORM fehlte beispielsweise komplett. Oder sie enthielt proprietäre Mechanismen, die meistens schwächer oder schwieriger zu benutzen waren.

Transparenz und Mitgestaltung

Obwohl die Bausteine der neuen Lösung aus Enterprise Subscriptions bestehen, sind sie immer noch Open Source. Das reduziert den Vendor-Lock-in für die Kunden und ist natürlich von großem Vorteil für die Entwickler. Denn diese können sich bei Problemen oder Unklarheiten durch einen Blick in den Sourcecode viel besser selbst helfen.

Außerdem sollte die Möglichkeit der Mitgestaltung an der Entwicklung der genutzten Produkte nicht außer Acht gelassen werden. Warum in der eigenen Applikation einen Workaround implementieren und ihn über Jahre mitschleppen, wenn man mit dem vielleicht selben Aufwand das fehlende Feature direkt im genutzten Produkt implementieren kann und dafür dann künftig sogar offiziellen Support erhält? Hierbei spielt die Nähe der Hersteller zur Community aber natürlich eine entscheidende Rolle.

Developer-friendly BPM

Der Umstieg auf einen selbst zusammengestellten Open-Source-BPM-Stack war also offensichtlich die richtige Entscheidung. Es gibt keinen Aspekt, in der man der alten Lösung auch nur eine Träne nachweint. Nun sind das beschriebene Projekt und die dort gemachten „Lessons learned“ keineswegs Einzelfälle. Auch in Fachkreisen gewinnt dieser leichtgewichtige Ansatz, BPM zu betreiben, unter dem Namen „Developer-friendly BPM“ zunehmend Anerkennung [1]. Hier rückt die bereits zu Beginn erwähnte Erkenntnis in den Vordergrund, dass BPM-Anwendungen nichts weiter als normale Geschäftsanwendungen sind, bei denen ein Teil der Geschäftslogik über BPM(N) ausgedrückt und eine BPM(N) Engine ausgeführt wird. Um also auch hier effizient sein zu können, sollte das Rad nicht neu erfunden werden und die Entwicklung sich deshalb nicht in das Korsett einer proprietären Suite zwängen, sondern stattdessen das BPM-System sich einfach und leicht in bestehende Architekturen integrieren lassen.

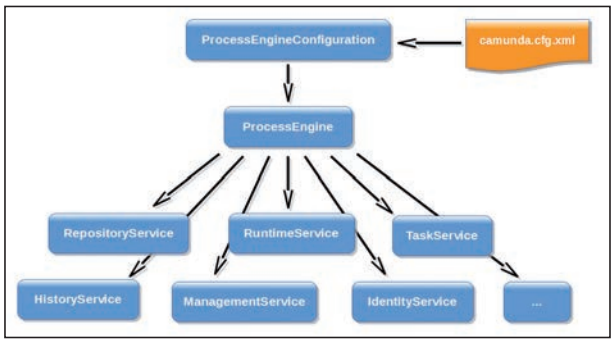

Abb. 1: Aufbau des Camunda-Java-API

Leichtgewichtige Prozessentwicklung am Beispiel „Camunda BPM“

Um ein Bild davon vermitteln zu können, warum eine BPM-Engine wie Camunda leichtgewichtig ist und warum sie sich gut in bestehende Java-(EE)-Architekturen einfügen lässt, betrachten wir das Produkt mit ein paar kurzen Codebeispielen etwas mehr aus der Nähe. Wer danach Lust auf mehr hat, sollte unbedingt einen Blick in die offizielle Dokumentation [2] werfen.

Integrationsfähigkeit ist bei Camunda BPM ein wichtiger Aspekt. So gibt es verschiedene Möglichkeiten [3], die Engine zu betreiben – beispielsweise „container managed“ innerhalb aller gängigen Java EE Application Server (für JBoss 7 gibt es beispielsweise ein eigenes Subsystem), über Spring oder einfach Standalone und In-Memory per POJO.

Das Java-API ist einfach und trotzdem mächtig, hat praktisch keine Abhängigkeiten und ist klar strukturiert. Wer nicht über Java arbeiten kann oder will, hat natürlich auch noch andere Möglichkeiten, wie beispielsweise REST. In diesem Artikel wird aber ausschließlich ein Blick auf das Java-API geworfen.

Ausgangspunkt aller Aktionen im Java-API ist das Process-Engine-Objekt, das auf unterschiedliche Weise erzeugt wird – je nach Art der Integration von Camunda in die Applikation. Im vorgestellten Projekt wird die Engine durch den JBoss Application Server zentral verwaltet. Das bedeutet, es gibt eine Engine-Instanz für alle Anwendungen. Durch Nutzung der CDI-Erweiterungen von Camunda lässt sich das Objekt dann ganz einfach per CDI injizieren:

```
@Inject ProcessEngine engine;
```

Andere Möglichkeiten zum Bootstrapping der Process Engine kann man unter [2] nachschlagen.

Hat man eine Process Engine erzeugt oder eingebunden, geht es praktisch immer unabhängig von der Umgebung weiter. Das Objekt dient als Factory für diverse

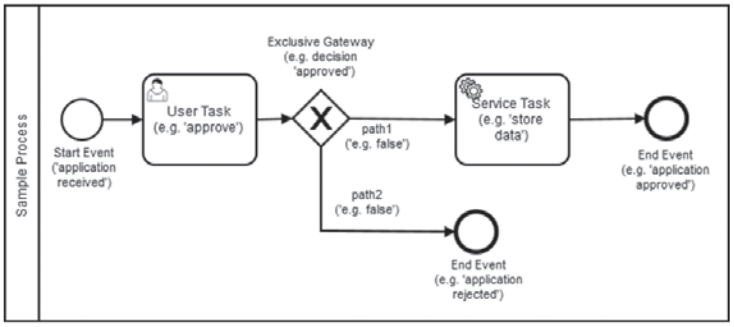

Abb. 2: Ein Beispiel für ein BPMN-Modell

Services, die wiederum themenbezogen Zugriff auf die verschiedenen API-Funktionalitäten bieten. So erlaubt uns beispielsweise der *RuntimeService* das einfache Starten von Prozessinstanzen:

```
engine.getRuntimeService().startProcessInstanceByKey("SampleProcess");
```

Der Prozess mit dem Namen *SampleProcess* muss dabei als Teil einer BPMN-Datei mit im Deployment (WAR, EAR etc.) vorhanden sein. Camunda scannt beim Starten der Java-EE-Anwendung automatisch deren Klassenpfad und registriert alle dort enthaltenen BPMN-Prozesse automatisch in der Engine. In den anderen Betriebsvarianten der Engine kann man die Prozesse auch per XML-Deskriptor oder API-Aufruf registrieren. Ein Prozess könnte beispielsweise aussehen wie in **Abbildung 2**.

Dieses Diagramm wurde mit dem BPMN Modeler von Camunda erstellt. BPMN ist prinzipiell aber ein XML-Format und lässt auch proprietäre Erweiterungselemente zu, mit dem Engine-Provider spezifische Metadaten im Prozessmodell hinterlegen können. Diagramme lassen sich deshalb auch mit anderen Modellierungstools vorbereiten, nur die technische Anreicherung muss dann entweder in den Camunda-Modellierungswerkzeugen oder in irgendeiner anderen Form auf der XML-Ebene erfolgen.

Zurück im API bieten nun die meisten der genannten Services neben dem Ausführen von Aktionen in der Engine weiterhin noch Möglichkeiten zur Suche von prozessrelevanten Objekten in der Datenbank. Wenn der Prozess beispielsweise einem menschlichen Prozessteilnehmer eine Aufgabe einstellt (User Task; **Abb. 2**), so wird in der Datenbank eine Taskinstanz erzeugt. Tasks lassen sich dann nach verschiedenen Filterkriterien über den *TaskService* suchen:

```
List<Task> tasks = engine.getTaskService().createTaskQuery()
  .processDefinitionKey("SampleProcess")
  .taskCandiateGroup("sales")
  .list();
```

Mit dem Ergebnis der oberen Suche könnte dann z. B. ein UI gefüttert werden, das einem User alle seine anstehenden Aufgaben anzeigt. Das Taskobjekt stellt alle dafür notwendigen und nützlichen Informationen per Getter zur Verfügung.

Beendet ein Benutzer die Arbeit an einer Aufgabe, so muss dies der Prozess-Engine natürlich signalisiert werden, damit die Prozessinstanz weiter laufen kann. Dies erfolgt ebenfalls über den *TaskService*:

```
engine.getTaskService().complete(task.getId());
```

Natürlich werden über das API aber auch viele andere Use Cases abgebildet. So kann ein Task beispielsweise einfach einem Bearbeiter fest zugeordnet werden:

```
engine.getTaskService().claim(task.getId(), "mscholz");
```

Bestimmte Metadaten eines Tasks werden über das Taskobjekt selbst manipuliert und müssen dann ebenfalls über den *TaskService* zurück in die Datenbank geschrieben werden:

```
task.setDueDate(dateTomorrow); // Muss morgen erledigt werden
task.setPriority(100);  // Hat maximale Priorität
engine.getTaskService().saveTask(task); // Update in der DB
```

Arbeitsweise von Camunda

Bei den letzten beiden Beispielen ist es für das Verständnis der Arbeitsweise von Camunda wichtig, dass die gezeigten Veränderungen an den Metadaten einer Aufgabe (Fälligkeitsdatum und Priorität ändern) zunächst keine weiteren Folgen haben. Es wird also dadurch nichts in der Engine ausgelöst. Es werden lediglich die entsprechenden Felder in der Datenbank aktualisiert.

Wie diese Daten nun interpretiert werden sollen, liegt ganz im Ermessen des Entwicklers der Anwendung – meistens werden sie jedoch im UI visualisiert oder als Suchkriterien verwendet. So könnte man beispielsweise Tasks mit hoher Priorität im UI rot markieren oder einem Benutzer nur diejenigen Tasks anzeigen, für die er als Bearbeiter eingetragen ist. Aufgaben lassen sich so nach Fälligkeitsdatum sortieren und auflisten, solche mit einem übertretenen Fälligkeitsdatum lassen sich mit einem entsprechenden Markersymbol versehen.

Camunda schränkt den Entwickler hier in keiner Weise ein. Es konzentriert sich auf ein minimales und sinnvolles Set an Metadaten, die in der Datenbank verwaltet und über das API abgefragt und manipuliert werden können. Gleichzeit gibt es natürlich die Möglichkeit über Variablen beliebige eigene Informationen hinzuzufügen. Dieses Konzept zieht sich wie ein roter Faden durch das gesamte API und erlaubt dem Entwickler eine einfache Integration der Engine in seine bestehende Anwendungslandschaft.

Web-UIs mit Plug-ins unterstützen

Trotzdem zwingt einen solch ein hoher Freiheitsgrad im Umkehrschluss natürlich dazu, immer selbst etwas entwickeln zu müssen, was manchmal, z. B. bei Evaluierungen, PoCs oder vielleicht sogar in der Produktion,

durchaus von Nachteil sein kann. Deshalb entwickelt Camunda seit Jahren zwei Webapplikationen, „Camunda Tasklist“ und „Camunda Cockpit“, die kostenlos als Add-on zur Prozess-Engine verfügbar sind. Die Tasklist dient dem bequemen Starten von Prozessinstanzen mit initialer Datenbefüllung, sowie der Verwaltung von Aufgaben und deren Bearbeitung durch den Endbenutzer. Das Cockpit dient der Administration und dem Monitoring von Prozessen. Beide sind inzwischen sehr umfangreich und bieten eine wirklich umfassende Grundfunktionalität. Sie lassen sich aber auch über Plug-in-Konzepte entsprechend erweitern, anpassen und wiederverwenden. Somit kann eine hohe Produktivität auch ohne zu starkes Customizing und Entwicklungsaufwände erreicht werden.

Variablen einfach handeln

Mit Variablen bietet Camunda die Möglichkeit, quasi beliebige Daten im Scope einer Prozessinstanz oder Teil-Scopes davon, abzulegen. Ihre Lebensdauer ist also an die des Prozesses gebunden. Camunda stellt außerdem die Persistenz dieser Daten sicher, auch wenn der Prozess unterbrochen wird, beispielsweise bei Human Tasks. Das API stellt diverse Möglichkeiten zur Verfügung, um Variablen zu speichern oder zu laden. Innerhalb einer Human Task geschieht dies beispielsweise wieder über den *TaskService*:

```
// Speichern
engine.getTaskService().setVariable(task.getId(), "customer", "john");
// Laden
String customer = engine.getTaskService().getVariable(task.getId(),
                                                       "customer");
```

Das API bietet noch viele weitere Möglichkeiten, um Variablen verwalten zu können, z. B. aus Callbacks heraus, per EL-Ausdruck, aus Inlineskripte, die direkt im Prozess eingebunden werden können, oder in Datenmappings, die deklarativ an bestimmten Prozessschritten definiert werden können. Eine ausführliche Dokumentation hierzu findet man unter [4].

Engine-Callbacks sind kein Problem

Natürlich bestehen die meisten BPM-Prozesse nicht ausschließlich aus Human Tasks, also Schritten, bei denen ein Benutzer manuell ein Aufgabe abarbeiten muss. Meistens ist es beispielsweise auch notwendig, Code in der Anwendung aufzurufen, entweder um Daten abzugreifen oder Daten zu übermitteln. Hier rufen also nicht wir aus unserem Code Funktionalität der Engine auf, sondern umgekehrt. In Camunda gibt es hier prinzipiell drei verschiedene Möglichkeiten:

- Delegation Code [5]: Dies sind eigentlich klassische Listener. Man schreibt also eine Klasse, die ein be-

stimmtes Interface implementiert und registriert die Klasse an geeigneter Stelle im Prozessmodell. Der Zugriff auf Variablen oder andere Daten ist über ein mitgeliefertes Contextobjekt verfügbar.
- Expression Language [6]: Hier kann man über typische EL-Ausdrücke, wie man sie aus JSP/JSF kennt, einfache Berechnungen durchführen – Variablen sind auch implizit über ihren Namen verfügbar – oder beispielsweise im Java-EE-Container auf CDI Beans zugreifen.
- Scripting [7]: Reichen einfache EL-Ausdrücke nicht aus, kann komplexere Logik auch direkt inline im Prozess kodiert werden. Auch hier ist schneller Zugriff auf Variablen direkt über deren Namen möglich

Das folgende Codebeispiel zeigt exemplarisch ein Java Delegate, das als Callback für einen *ServiceTask* hinterlegt werden kann.

```
public class MailNotificationDelegate implements JavaDelegate
{
  public void execute(DelegateExecution execution)
  {
    String customer = execution.getVariable("customer");
    MyMailService.getInstance().sendCustomerNotification(customer);
  }
}
```

Da *Delegate*-Klassen aktuell bei Camunda per Reflection instanziiert werden, kann im gezeigten Beispiel der *MyMailService* nicht per *@Inject*-Klausel injiziert werden. Allerdings kann man die Klasse einfach durch eine *@Named*-Annotation per EL zugreifbar machen und über diesen Weg ansprechen. Dann funktionieren auch *Injects*. In Zukunft wird aber der Reflection-basierte Zugriff auf Delegates um einen CDI-basierten Weg erweitert, dann ist nicht einmal das *@Named* notwendig.

Sehr nützlich bei Camunda ist außerdem, dass EL-Nutzung und teilweise auch Scripting nicht nur bei *ServiceTasks* oder ähnlichen Callback-Elementen eines BPMN-Prozesses möglich sind, sondern an vielen anderen Stellen im Prozess. Dies erlaubt eine große Dynamik und Flexibilität sowie eine einfache Integration von Geschäftslogik an diversen Stellen des Prozesses. So kann eine Expression beispielsweise genutzt werden, um Folgendes zu ermitteln:

- Benutzer oder Gruppe, die eine User-Task bearbeiten können
- Priorität oder Fälligkeitsdatum einer User-Task
- welcher Pfad oder welche Pfade bei einer Verzweigung im Prozess genommen werden soll
- Wartezeit bei Timer-Events

Die Möglichkeiten, Callbacks einzusetzen, sind zahlreich und umfangreich – ebenso wie beim Umgang mit Variablen. Zum Glück zählt aber die Onlinedokumentation der Engine sicherlich zum Besten und Umfangreichsten, was man im Open-Source-Umfeld so finden kann. Ein Besuch der referenzierten Links ist also definitiv einen Blick wert.

Fazit

Business Process Management muss nicht kompliziert oder schwergewichtig sein. Sicherlich gibt es ein paar BPM-Systeme, mit denen man mit rein deklarativen und modellgetriebenen Ansätzen und ohne viel Code tolle Prozesse bauen kann. Aber ist es kein Geheimnis, dass bei solchen Lösungen auch sehr schnell das Ende der Fahnenstange erreicht sein kann, vor allem, wenn Zahl und Komplexität der Anforderungen zunehmen.

Erst wenn BPM-Anwendungen als klassische Geschäftsanwendungen verstanden und von entsprechenden Teams entwickelt werden, kann die Technologie ihre vollen Stärken ausspielen. Oberstes Gebot hierbei sollte aber sein, dass BPM nicht als einzelne Technologie die Architektur des gesamten Systems bestimmen sollte. Deshalb bieten die neuen leichtgewichtigen Ansätze hier im Normalfall bessere Voraussetzungen als die großen schwergewichtigen Suites.

Und mit Case Management und Business Rules gibt es starke artverwandte Partnertechnologien, die sich mit BPM ideal ergänzen und zusammen immer noch genug „Model-driven“ mitbringen, damit auch Fachabteilungen in sinnvollem Maße bei der Gestaltung von Anwendungen mitwirken können.

Michael Scholz ist BPM-Architekt bei einem mittelständischen deutschen Automobilzulieferer. Er beschäftigt sich seit ca. zwölf Jahren mit XML, Java und Java-EE-Technologien und ist Koautor des Buchs „Java und XML“. In Firma war er jahrelang für die Eigenentwicklung eines Java-EE-basierten EAI-Systems zuständig. Seit ca. sechs Jahren beschäftigt er sich dort nun auch intensiv mit BPM und hat in den letzten drei Jahren eine entsprechende Plattform basierend auf Java EE und Open-Source-Technologien etabliert und betreut.

Links & Literatur

[1] Developer friendly BPM: http://bit.ly/1Vvcp9W

[2] Camunda BPM Manual: http://bit.ly/1PJzxDn

[3] Process Engine Bootstrapping: http://bit.ly/1SNPjOn

[4] Process Variables: http://bit.ly/1VvcwlQ

[5] Delegation Code: http://bit.ly/1nCFXZO

[6] Expression Language: http://bit.ly/1JLx4Gn

[7] Scripting: http://bit.ly/1VvcGty