Event Driven Process Engine

Gruppe Processmaker
Lukas Rosentreter, Alexander Senger

Inhalt

Kontext und Motivation

Welche Bereiche sind betroffen?

Complex Event Proccessing

…oder kurz *CEP*. Dabei wir ein Fluss an Daten beziehungsweise Ereignissen (Events) aus einer oder mehreren Quellen zusammengeführt und in Echtzeit analysiert. Echtzeit deswegen, weil genau das es von herkömmlicher Analyse der Daten, nach dem aggregieren, abgrenzt.
Wissen, dass aus der Kombination mehrerer Ereignisse gewonnen wird, sind Komplexe Ereignisse. Ereignisse sind nicht weiter spezifiziert oder limitiert und können von Wetter-Daten über Börsentrends bis zu Netzwerk-Logs reichen.
Bei der Analyse spielen sowohl die konkreten Werte, als auch der Unterschied zwischen ihnen (Delta) eine Rolle.
Um Zusammenhänge und mögliche Schlüsse verlässlicher aufdecken, respektive treffen zu können, ist eine kritische Menge an Daten von Nöten. Ein kontinuierlicher Strom an

zusammenhängenden Daten wird daher als Voraussetzung angenommen.

Es lassen sich alleinig aus der Beschreibung von CEP keine Rückschlüsse auf potentielle Nutzer oder Anwendungsbereiche ziehen, denn relativ große Mengen an Daten, lassen sich, mindestens durch Logs, heutzutage sehr leicht produzieren oder aggregieren. Ausgehend davon ist es theoretisch allen Menschen möglich, ihre Daten durch eine CEP zu schleusen und Schlüsse über ihr System, ihre Prozesse oder Strategien zu ziehen.

Process Execution

Als Process Execution Engine bezeichnet man Software, die darauf abzielt (Geschäfts-) Prozesse zu unterstützen und Teile davon gegebenen Falls zu automatisieren. Diese Prozesse sind im Vorfeld modelliert und bilden in der Regel häufig auftretende Abläufe ab. Zur Modellierung wird eine besondere Sprache verwendet, in unserem Fall *Business Process Model and Notation* (BPMN). Alternativ gibt es weitere Sprachen, wie z.B. *Business Process Execution Language* (BPEL). Ein Prozess selber kann unter anderem verschiedene Ereignisse und Aktivitäten beinhalten, sowie Schnittstellen nach außen. Ob diese zu Menschen oder Maschinen sind, Daten senden oder empfangen spielt dabei keine Rolle. Die Engine hilft dabei, die vorhandenen Prozesse aus- und durchzuführen, aber auch zu analysieren und Möglichkeiten zur Verbesserung aufzuzeigen.

Profess Execution Engines können zum Beispiel benutzt werden, um eine Bestellung in einem Onlineshop oder eine Supportanfrage zu unterstützen und umzusetzen. Ein Prozess kann dabei wie folgt modelliert sein.

Ein Kunde legt in einem Webshop ein Produkt in den Warenkorb und klickt auf "Jetzt bestellen". Der Prozess beginnt damit, zu überprüfen ob der Warenkorb nicht leer ist. Ist die Bedingung erfüllt geht es weiter zur Kasse. Der Shop bietet mehrere Zahlungsmöglichkeiten und der Kunde wählt seine aus. Der Prozess wartet nun, bis er Bestätigung der Zahlung hat. Ob diese sofort (Kreditkarte) erfolgt oder erst einige Tage später (Kauf auf Rechnung) ist dabei egal. In dem Moment in dem die Bestätigung erfolgt, wird geprüft ob und wenn ja in welchem Lager die Waren verfügbar sind. Angenommen die Waren sind alle im gleichen Lager vorrätig, bekommt ein Mitarbeiter eine Liste in welchem Regal welches Produkt liegt und kann anfangen das Paket zu packen. Sobald der aufgedruckte Barcode auf dem Paket gescannt wurde, bekommt der Kunde eine E-Mail mit der Versandbenachrichtigung. Bei Ankunft des Pakets wird der Barcode erneut gescannt und der Prozess mit der Unterschrift des Kunden abgeschlossen.

EventDriven-BPM

Durch die sehr offenen use cases der beiden betroffenen Bereiche, ergeben sich auch in der Schnittstelle der beiden noch eine Unzahl verschiedener Anwendungen. Laut Theorie wäre es auch dem Hobby-Tüftler für seine Heim-Automation möglich Event-Driven-Process-Models zu verwenden, jedoch scheint der Aufwand kaum in irgendeiner Relation zum Nutzen zu stehen. Wesentlich wahrscheinlicher ist die Anwendung ab kleinen bis mittelständischen Unternehmen oder auch Start-Ups. Der Hauptaufwand für Unternehmen, die noch keine Prozess-Modelle nutzen besteht dann darin, die Abläufe des Tagesgeschäfts in solche zu übersetzen. Start-Ups hingegen haben die Möglichkeit, gleich von Beginn, ihre Geschäfte Event-Driven anzulegen. In beiden Fällen gibt es den Unternehmen - ob etabliert oder nicht - die Möglichkeit ihre Prozesse zu analysieren, anzupassen und zu optimieren.
Zum Beispiel kann bei einer Bestellung bei Zalando auf einen Event Stream für Verkehrsdaten innerhalb des Business Process zugegriffen werden und anhand der Auswertung entschieden werden, an welches Lager die Bestellung weitergeleitet wird. Umgekehrt können auch die Lieferfahrzeuge wiederum Events für die CEP Engine produzieren, die dann bessere Einsicht in die Verkehrslage ermöglichen.

Wieso ist es interessant sich damit zu beschäftigen?

BPM beschäftigt sich mit einzelnen Prozessen. CEP schafft es Zusammenhänge zwischen großen Mengen an Daten automatisiert herzustellen. Mehr Daten als es für Menschen eventuell möglich wäre. Das Zusammenspiel der Beiden, ermöglicht ein ganzheitlicheres Bild auf Prozesssabläufe und Zusammenhänge zwischen Diesen aufzuzeigen. So kann CEP dabei unterstützen, den Prozess in sich selber zu optimieren und innerhalb der Abläufe neue, bessere oder - neutral gesprochen - andere Zusammenhänge herzustellen. Darüber hinaus bietet CEP aber auch die Möglichkeit zwei verschiedene, in sich abgeschlossene Prozesse zu verknüpfen. Das kann zum sowohl auf der inhaltlichen Ebene passieren, aber auch auf der Daten-getriebenen Ebene. Das Ändern eines bestehenden, Verknüpfen zweier alter oder das Erstellen eines neuen Prozesses braucht dabei noch den Eingriff eines Menschen.

Problemstellung

Auf einfachster Ebene gesprochen, verbindet die entwickelte Software zwei Anwendungen. Auf der einen Seite eine CEP-Engine, auf der anderen eine BPM-Software. Beides sind Open-Source Projekte. Die CEP-Engine, Unicorn, entwickelt vom Hasso-Plattner-Institut in Potsdam wurde in Java geschrieben und läuft auf einem Apache Tomcat Server. ProcessMaker, entwickelt durch ProcessMaker Inc., ist hauptsächlich in PHP geschrieben. Ein Server für ProcessMaker ist nicht vorgegeben, doch Apache HTTP Server wird empfohlen und in der Dokumentation verwendet. Durch die Server Architektur bringen beide Komponenten REST-APIs mit sich, sodass eine Middleware die naheliegende Entscheidung war.

Schon von der Terminologie her, bringen sowohl Complex Events als auch BPMN *Events* mit. In Unicorn können Queries erstellt werden, die auf eine Abfolge von eingehenden Events reagieren. Ein solches, aus mehreren Granulaten bestehendes, Event nennt man Komplexes Event. Die Daten der einzelnen Events und den Typ und die ID des Komplexen Events, können dann an einen vorbestimmten Ort (z.B. via REST-API) übergeben werden. Events in BPMN repräsentieren Dinge, die sich Ereignen, wie zum Beispiel, das Eingehen einer Bestellung. BPMN Events teilen sich in drei Typen auf: Start, Stopp und Intermediate. Events können dabei jeweils fangend (Catching) oder werfend (Throwing) sein. Darüber hinaus gibt es weitere Klassifizierungen, die aber hier keine weitere Rolle spielen sollen.

Das Ziel war, die Events beider Anwendungen zu verknüpfen und insbesondere für die BPMN-Seite Schnittstellen für jeden Typen, sowohl Catching als auch Throwing zu haben. Die schlussendliche Funktionalität sollte beinhalten, ein Complexes Event an ProcessMaker zu senden (und dadurch einen Prozess zu starten, zu beenden oder zwischendurch zu beeinflussen) und von ProcessMaker aus ein Event an Unicorn zu senden und es in den Event-Fluss einzugliedern.

Verlauf des Projektes

Planung

Die Milestones waren den gesamten Verlauf des Projekts über sehr grob gehalten. Nicht zuletzt, da wir als einziges Projekt Management Tool GitHub hatten und mit den dortigen

Werkzeugen *Wiki* und *Issues* gearbeitet haben. So gab es zu Beginn einen sechsstufigen Plan im Wiki, der wie folgt aussah.

> 1. **Setting up the two parallel systems** *Until 15.05.*
>
>   - Create (Docker) Image, preferably Debian
>   - Install Dependencies and Unicorn & Processmaker
>   - Setup coding environment
>
> 2. **Identifying Interfaces of the two systems** *Until 22.05.*
>
>   - Make familiar with interface documentation
>   - Simple tryout with small scripts and examples
>
> 3. **Normalising the data and rudimentary communication between systems** *Until 31.05.*
> 4. Eliminating the kinks, testing, debugging
> 5. Sophisticating the communication and broadening of the possibilities
> 6. Further Testing, Debugging

https://github.com/hupda-edpe/p/wiki/Project

Entstanden ist dieser Plan vorwiegend aus den bereitgestellten Folien mit der Aufgabenstellung und dem generellen Grundgerüst, das für fast jedes Setup von Applikationen notwendig ist. Während des Projekts wurde lediglich drei bis vier Male mit der konkreten Planung gearbeitet, aber die einzelnen Schritte und die Abfolge dieser, stimmten mit der Realität überein. Im Verlaufe der einzelnen Phasen, sind diese nach Bedarf präzisiert und arbeitsteilig abgearbeitet worden.
Auffällig ist, dass in der Planung keine Entscheidungen beziehungsweise konkreten Aussagen zum Endgültigen Produkt sind (z.B. Middleware oder die Nutzung von APIs). Architektur und Scope wären zu Beginn sicherlich auch mit relativer Genauigkeit vorhersehbar gewesen, schien aber zweitrangig gegenüber dem vergleichsweise hohen Setup-Aufwand der beiden bestehenden Software Projekte. Erst nach erfolgreichem zum-laufen-bringen von Unicorn und ProcessMaker ging es um die effizienteste Variante, wie diese zu verknüpfen sind.
Diese Herangehensweise ist prototypisch für viele weitere Planungs-Entscheidungen des Projekts. So kamen weitere Wiki-Seiten dazu, die sowohl den theoretischen Ablauf für das Arbeiten mit Aktivitäten, als auch den API-spezifischen Ablauf dokumentierten. Je weiter das Projekt fortschritt, desto klarer wurde die Aufgabenteilung im Team und die eigentliche Planung und Dokumentation trat in den Hintergrund, sodass gegen Ende des Projektes durch direkte Kommunikation die anstehenden Aufgaben leicht abgearbeitet werden konnten.

Workflow

Vorausgesetzt war die Arbeit mit GitHub und gegeben waren unterschiedliche Betriebssysteme im Team. Um möglichst viel Overhead zu vermeiden und flexibel zu bleiben, schienen Container eine sinnvolle Wahl für die Entwicklung. Aufgrund von Vorerfahrung wurde Docker beziehungsweise Docker-Compose als System ausgewählt.
Container erlauben unter anderem automatisiertes Setup und schnelles Neustarten der Systeme, was mehrere Vorteile gegenüber nativen Installationen oder Virtuellen Maschinen mit sich bringt. Native Installation von Unicorn und ProcessMaker auf mehreren verschiedenen Betriebssystemen bringt mehrfachen Aufwand mit sich, ist nicht wirklich sauber, was Kommunikation und Abtrennung untereinander anbelangt und bringt keine Sicherheit, dass alle Systeme gleich funktionieren.
Mit Virtuellen Maschinen zu arbeiten hätte sehr viel Voraussicht bedeutet, da das Setup einmalig am Anfang passieren muss, bevor die VM an alle Teammitglieder verteilt wird. Nachträgliche Änderungen und deren Nachverfolgung in einem nicht-lokalen Repository sind nicht praktikabel.
Containerisierung erlaubt kleine Konfigurationsdateien, ausgelagerten Code und garantiert identische Systeme über verschiedene Betriebssysteme hinweg. Nach einmaligem Setup können wenige, kleine Dateien über GitHub zur Verfügung gestellt werden und bei allen Teammitgliedern leicht installiert werden. Updates im Setup benötigen lediglich das austauschen dieser Dateien und das Neuinstallieren erfolgt via einzeiligem Befehl im Terminal. So sind auch tiefgreifende System-Änderungen mit Leichtigkeit in der Versionsverwaltung enthalten und nachvollziehbar.

Der Quellcode wurde ebenso über GitHub verwaltet. Da Unicorn, in Java geschrieben, lediglich Kompiliert und Deployed werden musste, gab es hier keinen Bedarf für eine IDE. Trotz einiger schwerwiegenderer Änderungen in ProcessMaker, PHP, benötigte es auch hier keiner IDE, sodass jedes Teammitglied mit dem Editor ihrer Wahl arbeiten konnte. Die Kommunikation im Team erfolge von Anfang an informell wie Threema und gelegentlichen E-Mails. Persönliche Treffen fanden meist vor den Veranstaltungen statt, je nach Bedarf länger oder kürzer.

Setup

Die Container zu Beginn aufzusetzen war um einiges umständlicher als ursprünglich geplant, nicht zuletzt, da es eine weitere potentielle Fehlerquelle darstellt. In Kombination mit der nicht ausgereiften ProcessMaker Software, war nicht immer klar, ob der Fehler an der Container Konfiguration oder im Quellcode von ProcessMaker lag.
Das Setup von ProcessMaker ist mit Docker Compose keine Schwierigkeit per se. Die Voraussetzungen spezifizieren ein Linux-Apache-MySQL-PHP (LAMP) Stack. Um die Architektur möglichst flach, also im Sinne der Container, zu halten, werden Server und Datenbank aufgeteilt. Ein Image mit einer Linux Installation und Apache mit PHP ist so fertig verfügbar, genauso wie eine MySQL Installation. Via Docker Compose werden die beiden Container verknüpft. Zwar fehlen in den von ProcessMaker gelieferten Systemvoraussetzungen einige Apache Module und Abhängigkeiten, diese sind jedoch dank Stack Overflow, mit einiger Recherche herausgefunden und nachinstalliert. Ebenso stimmte die Angabe des MySQL Treibers nicht. (`mysqlnd` ist angegeben, `pdo` wird aber im Code tatsächlich verwendet.)

Ein weiteres großes Problem mit Docker war, dass Docker bis vor einigen Monaten auf OS X anders funktionierte als auf Linux. Docker hat auf OS X eine virtuelle Debian Maschine via VirtualBox installiert, auf welcher die Container ausgeführt wurden. Unter Linux laufen diese direkt auf dem System. Normalerweise führt das selten zu Problemen, in unserem Fall aber waren die Zugriffsrechte auf Ordner zwischen Host (OS X) und Client (Debian via VirtualBox) relevant. (Der Code sollte dynamisch geladen werden, um den Container nicht bei jeder Änderung im Quellcode neu initialisieren zu müssen.) Da Docker aber nur Nutzer mit der `uid` `1000` auf dem Host lesen und schreiben lässt, Apache aber den Nutzer `www-data` braucht, war der Fix dem `www-data` -User die `uid` `1000` zu geben.

Den Unicorn Container haben wir vorerst nicht selber aufgesetzt, sondern den im *Shared* Repository vorhandenen, verwendet. Da der Deployment-Prozess unter OS X, aus unbekannten Gründen, fehlschlug, die Kompilierung aber einwandfrei verlief, haben wir die beiden Abschnitte getrennt und eine neue Umgebung mit vorkompilierter `.war` -Datei aufgesetzt. Ein weiterer Vorteil davon war, dass bei einer Neuinstallation des Unicorn-Containers nicht der ganze Kompilationsprozess erneut durchlaufen werden musste, sondern

lediglich die `.war`-Datei in einen Shared Folder eingebunden wurde.

Somit existierten zwei separate Umgebungen für Unicorn und ProcessMaker die es nun galt zu verknüpfen.

Kommunikation mit ProcessMaker

BPMN in ProcessMaker

Unicorn modelliert EventTypes und Events praktischerweise bereits mit BPMN. Die Prozesse, einschließlich Tasks und Events in ProcessMaker sind sowieso mit BPMN modelliert. Es lag daher nahe, EventTypen zu modellieren, die für den Austausch zwischen Unicorn und PM gedacht sind. BPMN sieht Erweiterung dieser Art bereits vor und bietet spezifikationskonforme Lösungen an. Es war zudem ein Bestreben diese BPMN Erweiterungen zwischen den drei Projektgruppen auszutauschen, so dass eine Interkompatiblität entsteht. Der Versuche eigene BPMN Erweiterungen in ProcessMaker einzuspeisen schlug allerdings fehl. Tatsächlich war es nicht einmal möglich standardkonforme Modelle zu importieren, lediglich eigens aus ProcessMaker exportierte Prozesse ließen sich wieder importieren.

Events

Nachdem dieser erste Ansatz keine Früchte trug, stellte sich dennoch die Frage, mittels ProcessMaker Events die Kommunikation mit Unicorn zu steuern. Auch hier wurden wir enttäuscht, da es nicht einmal gelang Events aus zwei verschiedenen Prozessen innerhalb von ProcessMaker zu verbinden. Events in ProcessMaker sind darüber hinaus auch nicht an die API angebunden. Da Events auch seitens ProcessMaker über einen Pull-Mechanismus implementiert waren, den man eigenständig von außen z.B. mit einem Cronjob auslösen musste, wäre man möglicherweise über diesen Ansatz zu einer Lösung gekommen. Da die Online Dokumentation aber nicht viele Informationen über die Funktionsweise des Mechanismus hergab schien auch dieser Weg nicht sehr Zielführend. Trial und Error und Reverse Engineering wären die besten Werkzeuge gewesen, welche nicht sonderlich effektiv schienen, nicht zuletzt da zu diesem Zeitpunkt bereits mehrere Fehler im Quellcode der Software aufgetaucht waren. Zeitrahmen und Arbeitsumfang ließen sich nicht abschätzen. Daher haben wir uns für die vielversprechendere Lösung über Tasks entschieden.

Das neue Ziel war weiterhin Komplexe Events von Unicorn an ProcessMaker und zurück zu senden, in ProcessMaker aber Aktivitäten zu nutzen um das Verhalten von Events nach außen zu simulieren.

Tasks

Tasks ist die Bezeichnung die ProcessMaker für BPMN Aktivitäten nutzt. Sie stellen zu erledigende Aufgaben dar und sind daher semantisch nicht ganz akkurat auf Komplexe Events abzubilden. Sie sind in ProcessMaker konkreten Nutzern oder Nutzergruppen des Systems zugeordnet und müssen entlang des Prozessablaufs geroutet werden. Sowohl das Abarbeiten als auch das routen sind via REST-API verfügbar. Diese ist online durchweg gut dokumentiert und die Vermutung, dass die ProcessMaker eigene Weboberfläche selber über diese API mit der Execution Engine kommuniziert, ist sogar naheliegend. Obwohl dies nicht der syntaktisch korrekte Ansatz ist, schien er der einzige der im Rahmen des Projektes als realistisch umsetzbar einzustufen war.

Eingehende Events lassen sich von der Middleware auffangen, an den Aktivitäten Endpoint pushen und von der Middleware automatisch weiter routen. Werfende Events zu simulieren ist nicht ohne weiteres möglich, nicht zuletzt, weil die API keine Callbacks ermöglicht. Aber eine geroutete Aktivität landet in der Inbox des zuständigen Nutzers und diese ist wiederum per API abzufragen. Somit polled die Middleware - ähnlich der Cronjobs - die Inbox eines designierten Unicorn-Nutzers und pushed gegebenenfalls ein Event in den Datenfluss von Unicron.

Prototyp für ProcessMaker

Da der generelle Ansatz zur Kommunikation mit ProcessMaker bereits eine Hürde war, haben wir uns dazu entschieden zunächst einen Prototyp zu implementieren, um damit die Möglichkeiten der API zu testen und einen Proof-of-Concept zu haben. Des Weiteren diente dies auch dazu, unseren Zwischenstand und das bis dahin gesammelte Wissen über unsere spezifische Process Execution Engine den anderen Gruppen vorzustellen und zu demonstrieren.

Um flexibel und interaktiv zu sein haben wir als Programmiersprache für den Prototyp `Python` gewählt. Da alle Teammitglieder die gleiche Version lokal hatten, schien es kein Problem, den Prototypen vorerst lokal zu verwenden. `Python` stellte sich als eine gute Wahl heraus, da wir so zügig zu einem funktionierenden Ergebnis kamen. Mit der `requests` Bibliothek ließ sich unkompliziert mit der API von ProcessMaker kommunizieren. Außerdem haben wir damit einen guten Grundstein für die eigentliche Middleware, die das Ziel des Projektes war, gelegt.

Django Middleware

Bis zu diesem Zeitpunkt war lediglich die Seite in Richtung ProcessMaker abgedeckt. Da man beim Registrieren von EventQueries in Unicorn einen HTTP Endpoint als Callback angeben musste, war klar, dass wir irgendeine Form von HTTP Server anbieten mussten. Wir wollten Teile vom Prototyp wiederverwenden und uns nicht damit aufhalten, ein Callback-Interface für Unicorn selber zu implementieren, also haben wir uns dafür entschieden einen Django-Server aufzusetzen. Das hatte außerdem den Vorteil, dass man die Middleware selber leicht über eine Weboberfläche bedienen konnte.

Schwierigkeiten

ProcessMaker

Im Verlaufe des Projekts kam es immer wieder zu Verzögerungen, meist durch Fehler in der Community Version von ProcessMaker. Abgesehen von den fehlenden Abhängigkeiten während des Setups, schienen einige MySQL-Operationen nicht getestet worden sein.
Nach erfolgreicher Installation von ProcessMaker, wird man als Admin durch ein paar Konfigurations-Schritte geleitet, die zwar funktionierten, aber nicht persistent waren. Bei der Initialisierung beim ersten Start wurde eine Tabelle nicht in die Datenbank geschrieben, weil die zuständige Query fehlschlug. Das Datumsformat war als `0000-00-00 00:00:00` eingetragen, obwohl die Datenbank `NULL` erwartet hat. Die zuständige Datei `project_root/processmaker/rbac/engine/data/mysql/insert.sql` sah wie folgt aus.

```
('00000000000000000000000000000006','PM_ALLCASES','2008-04-30 00:00:00',
    '2008-04-30 00:00:00',1,'00000000000000000000000000000002'),
# ...
('00000000000000000000000000000009','PM_SUPERVISOR','0000-00-00 00:00:00',
    '0000-00-00 00:00:00',1,'00000000000000000000000000000002'),
('00000000000000000000000000000010','PM_SETUP_ADVANCE','0000-00-00 00:00:00',
    '0000-00-00 00:00:00',1,'00000000000000000000000000000002'),
# ...
```

Gändert sah die Datei folgendermaßen aus.

```
('00000000000000000000000000000006','PM_ALLCASES','2008-04-30 00:00:00',
    '2008-04-30 00:00:00',1,'00000000000000000000000000000002'),
# ...
('00000000000000000000000000000009','PM_SUPERVISOR',NULL,NULL,
    1,'00000000000000000000000000000002'),
('00000000000000000000000000000010','PM_SETUP_ADVANCE',NULL,NULL,
    1,'00000000000000000000000000000002'),
# ...
```

Ein weiterer Fehler war die Query, die für die Registrierung von OAuth Apps zuständig war. Das besondere hierbei war, dass auch keine Fehlermeldung angezeigt wurde. Lediglich durch abfangen des AJAX Returns in der Konsole, ließ sich feststellen, dass es sich um ein Invalides MySQL Statement handelte. Dem dynamisch in PHP zusammengesetzte Query fehlte eine `GROUP` Column. Die Funktion

```
public function getAll($arrayFilterData = array(), $sortField = "",
    $sortDir = "", $start = 0, $limit = 25) {
    //...
    }
```

schlug fehl.

Hinzufügen der Zeile

```
$criteriaCount->addGroupByColumn("UPPER(". OauthClientsPeer::CLIENT_NAME .")");
```

hat das Problem, zumindest für unsere Zwecke, gelöst. Danach war es möglich neue OAuth Applications zu registrieren.

Unicorn

Auch Unicorn stellte einige Schwierigkeiten dar, nicht zuletzt, da die Dokumentation sehr spärlich war. Die im `shared` Repository bereitstehende Docker-Installation lief auf OS X nicht durch. Lediglich der Kompilationsprozess, nicht jedoch das Deployment. Das neue Setup, dass eine Vorkompilierte Version von Unicorn in einen gemeinsamen Ordner (zwischen Docker und Host) legte, stellte sich später als Vorteil heraus, als das Original GitHub Repository mit allem Quellcode verschwand.

Architektur der Middleware

Die von uns implementierte Middleware ist ein mit Django aufgesetzter Webserver, der die jeweiligen REST-APIs miteinander verbindet.

OAuth

Um auf die API von ProcessMaker zugreifen zu können, muss sich eine Anwendung erst von einem User registrieren lassen und erhält so eine Client ID und Secret. Als dieser User ist dann die Middleware nach erfolgreicher Authentifizierung eingeloggt. Es empfiehlt sich daher einen dezidierten User für diesen Zweck anzulegen, im Folgenden *Unicorn User*, dessen Login-Daten in *unicorn_config.py* eingetragen werden müssen. Mit den Login-Daten und o.g. Client ID und Secret lässt sich dann ein Token von ProcessMaker generieren lassen, mit dem sich die Anwendung bei HTTP Requests authentifizieren kann. Diese Funktionalität ist im Pythonmodul *pm_auth.py* implementiert. Das Token wird dabei lokal gespeichert und falls nötig aktualisiert oder neu generiert.

Für Unicorn waren keine Maßnahmen zur Authentifizierung nötig.

APIs

Die API von ProcessMaker gliedert sich in drei Bereiche:

> - Administration
> - Designer
> - Cases

Ersterer dient dazu User, Gruppen, Rollen und Systemeinstellungen von ProcessMaker zu verwalten. Dieser Aspekt wird von unserer Middleware nicht benutzt. Der Bereich Designer stellt Endpunkte bereit mit denen man Prozessmodelle verwalten und bearbeiten kann. Im Prototyp haben wir Teile davon verwendet, um die Tasks aufzulisten, mit denen man einen neuen Prozessablauf starten kann. Diese Funktionalität wird so nicht mehr benötigt, allerdings brauchen wir dafür die Möglichkeit die Beschreibung von einem

modellierten Task auszulesen, was sich über diesen Bereich der API realisieren lässt. Der letzte Bereich, Cases, ist der wohl wichtigste für unsere Anwendung. Hier sind alle Endpunkte veranlagt mit denen man Prozessabläufe steuern kann und Prozessvariablen ein- und auslesen kann. Im Modul *pm_wrapper.py* sind die von uns benötigten Endpunkte gekapselt.

Die API von Unicorn ist etwas simpler zu bedienen. Es lassen sich bequem Events posten, EventTypes und EventQueries erstellen, bearbeiten und löschen und für letztere eine Callback URL angeben. Implementiert ist die Schnittstelle im Modul *unicorn_wrapper.py*.

Für ProcessMaker gibt es im dazugehörigen Wiki eine extensive Dokumentation der Endpunkte, wohingegen wir für Unicorn nichts Vergleichbares finden konnten. Glücklicherweise sind in den Unit-Test von Unicorn Beispiele, die die Bedienung der API hinreichend erklären.

Konfiguration von ProcessMaker

In einem Prozessmodell müssen diejenigen Tasks, die als Schnittstelle zu Unicorn dienen, dem Unicorn User zugewiesen werden damit sie in dessen Inbox erscheinen. Ist ein solcher Task in einem Prozessablauf erreicht, kann die Middleware beim Pullen der offenen Cases somit feststellen, ob ein neues Event erzeugt werden soll.

Damit ein Mapping zwischen den Tasks und EventTypes möglich ist und die Middleware weiß, wie sie mit dem Case umgehen soll, ist es nötig Informationen bereits im Modell an den Task zu hängen. Dafür dient uns das Description-Feld von den Tasks, welches wir über die Designer-API auslesen können. Erwartet wird von unserer Anwendung, dass sich dort ein JSON befindet, das beispielsweise so aussehen könnte:

```
{
  "blocking": true,
  "event_type": "SomeType",
  "start_task": "1234",
  "start_process": "5678"
}
```

Die Einträge *blocking* und *event_type* müssen vorhanden sein. Ersterer gibt an, ob die Middleware nach dem Erzeugen des Events den Case direkt weiterleiten soll (nicht blockierend) oder in der Inbox belassen soll (blockierend). Der EventType des zu erzeugenden Events wird entsprechend dem zweiten Eintrag entnommen.

Ist ein Task der einen Prozess startet dem Unicorn User zugewiesen, kann dieser durch *start_task* und *start_process* identifiziert werden und bei einem Query Match von der Middleware gestartet werden. Diese beiden Parameter sind optional.

Das Pullen der offenen Cases erfolgt manuell. Auch das Pushen der erzeugten Events an Unicorn wird über die Weboberfläche gesteuert. Dafür haben wir Cases selber als Klasse modelliert um deren Status festzuhalten (neu, pushed, routed).

Konfiguration von Unicorn

Damit die Kommunikation klappt, müssen zunächst *EventTypes* und *EventQueries* erstellt werden. Um bei Queries eine Callback URL anzugeben, muss man diese über die REST API von Unicorn erstellen und da wir sowieso ein Webserver als Middleware gewählt haben, haben wir uns dafür entschieden, *EventTypes* und *EventQueries* über eine Weboberfläche zu verwalten und eigene Klassen dafür zu erstellen.

Dies hat noch einen weiteren Zweck. Um auch Business Variablen aus einem Prozessablauf in Unicorn Events zu übertragen (und umgekehrt), ist es nötig zu wissen, wie diese heißen. Dafür werden die einzelnen Variablennamen und deren Datentyp als Attribute eines EventTypes definiert. Wird nun ein Case als Event nach Unicorn gepusht, werden explizit die Variablen, die zu dem angegebenen EventType gehören, in ProcessMaker abgefragt. Das heißt auch, dass man bei der Benennung der Variablen und Erstellung der Prozessmodelle darauf achten muss, dass diese übereinstimmen.

Alle EventTypes haben automatisch die Felder *AppUid*, *ProUid* und *TasUid* definiert. Ersteres ist die ID des jeweiligen Case, wenn dieser in ein Event überführt wird und die anderen beiden sind für die o.g. *star_task* und *start_process* IDs vorgesehen.

Empfängt die Middleware ein Ergebnis von einer Query gibt es zwei mögliche Reaktionen. Ist der Key *AppUid* vorhanden, wird der entsprechende Case geroutet. Sind die Keys *ProUid* und *TasUid* im Ergebnis, wird den entsprechende Task gestartet. Unabhängig davon werden die restlichen Werte aus dem Ergebnis als Variablen an den existierenden oder/und an den neu erzeugten Case weitergegeben.

Reflektion

Nutzbarkeit

Die Middleware stellt mit Sicherheit einen validen Prototyp dar. Sie zeigt, dass die Kommunikation zwischen beiden Engines funktioniert, erfordert dabei aber sicherlich noch zu viel menschliche Interaktion. Ein wirklicher Einsatz in einer Produktivumgebung beim jetzigen Stand ist kaum vorstellbar. Durch das einfache Setup via Docker Compose stellt die Middleware, so wie sie momentan ist, eventuell einen guten Testkandidaten dar. Das heißt, sollte jemand in Erwägung ziehen Event-Driven Process Management einzusetzen, scheint unser Projekt ein valider Einstiegspunkt.

Schwächen und Potential

Bis die Software in Produktion genommen werden könnte, gibt es einige Features die noch implementiert, beziehungsweise Architekturentscheidungen, die getroffen werden müssten. Der ganze Themenkomplex Sicherheit, hat bei der Entwicklung bisher keine Rolle gespielt. So sind die meisten Passwörter im Klartext in der Konfiguration bzw. im Code und der Einfachheit halber sehr einfach gewählt. (`123456` in der Regel.) Zudem gibt es Sicherheitsschwachpunkte bei den Gemeinsamen Ordnern und den Lese- und Schreibberechtigungen. Ebenso ist der Großteil unseres Projekts auf basierend auf Kommunikation zwischen APIs und Containern. Beide haben sehr großes Potential in verteilten Umgebungen eingesetzt zu werden. Zusätzlich zur Netzwerk-Sicherheit fehlt hier ein kompletter Authentifizierungs-Mechanismus in der Middleware und jegliche Anforderungen an

Verlässlichkeit und Stabilität.

Ebenso sollte beim der Diskussion der Einsatzgebiete Effizienz eine Rolle spielen. Python ist bekanntlich keine gut skalierende Sprache, CEP arbeitet aber mit sehr hohem Datenaufkommen. Zwar können Container etwas entgegenwirken, dennoch sollte eine stichhaltige Aussage zu Performance getroffen werden können, bevor die Software in Produktion geht. ProcessMaker selber wurde auch nicht auf hohe Datenaufkommen getestet, sodass die Middleware im Zweifelsfall sogar als Puffer wirken müsste. Die nötige Funktionalität dafür ist bisher aus noch nicht vorhanden und im Zweifelsfalls nochmal ein eigenes Projekt in sich.

Nicht zuletzt um effizienter zu arbeiten, aber auch um Dopplungen in Code zu vermeiden, ist auch die Entscheidung ein Add-On statt einer Middleware zu bauen eine Überlegung wert. Die API von ProcessMaker zu erweitern und Endpunkte für Events, sowie Callbacks zu implementieren, um direkt mit Unicorn kommunizieren zu können, sollte zwar nicht einfacher, aber sicherlich auch ein legitimer weg zu Event-Driven Process Management sein. Vorteil der Middleware hingegen ist ein gewisses Maß an Flexibilität. Da die Software auf beiden Seiten mit Fassaden (Wrappern) arbeitet, ist es mit relativer Leichtigkeit möglich, eine der beiden Engines auszutauschen. Voraussetzung dafür ist, dass die austauschende Engine eine REST-API bietet. Wie umfangreich der Rück-Umstieg von Aktivitäten auf Events ist, hängt dabei viel von der gegebenen API ab. Sicherlich müssen die Abläufe innerhalb der Middleware geändert werden (denn das Routen fällt mit großer Wahrscheinlichkeit weg) aber die Zuordnungsfunktion von Komplexem Event zu Event in Instanziiertem Prozess bleibt erhalten.

Das Level an Interaktion, welches zur Kommunikation zwischen den beiden Engins nötig ist, müsste sicherlich auch Automatisiert werden. Denkbar ist, viel mehr Konfiguration in die Prozessmodellierung zu schieben, sodass bereits beim Anlegen des Projekts in ProcessMaker definiert wird, was beim erhalten eines Komplexen Events passieren soll.

Letztlich liegt noch Optimierungspotential in den Konfigurationsdateien. Zur gemeinsamen Kommunikation müssen die beiden Engines vieles voneinander wissen und bisher ist dieses Wissen an mehreren Stellen gleichzeitig abgelegt. Zusammenführung der Konfigurationsdateien und ein Automatisiertes Setup dieser scheint sinnvoll und gibt ein einheitlicheres Software Produkt.

Ablauf

Im Verlauf des Projekts gab es im Team keine größeren Beeinträchtigungen. Der Anfang war, wie erwartet und üblich, etwas langsam, da sich alle Teammitglieder einfinden mussten und zwischenzeitlich gab es kleiner zeitliche Schwierigkeiten, aber die Kommunikation und Absprache hat in neunundneunzig Prozent der Fälle hervorragend funktioniert.

Beispiel und Bedienung

An dieser Stelle möchten wir ein Beispiel dafür geben, wie man die implementierte Funktionalität verwenden könnte. Das dient außerdem dazu, die Bedienung besser zu erklären.

Nachdem die grundlegende Konfiguration vorgenommen ist, erstellen wir in ProcessMaker die User *Trainee* und *Supervisor*. Ersterer soll Texte erstellen erstellen können die der Supervisor dann überprüfen und ggf. abändern kann. Dafür sind zwei Prozesse nötig:

- Proposal
- Review

In beiden werden die Variablen ID und Text erstellt. Im Proposal Prozess gibt es dann einen Task der an den User *Trainee* zugewiesen ist gefolgt von einem Task der ein Unicorn Event erzeugen soll. Der Review Prozess wird von einem Unicorn-Task gestartet und diesem folgen dann ein Task für den *Supervisor* und einer für Unicorn.

Im Unicorn Task aus dem Proposal-Prozess tragen wir folgendes ein:
`json { "blocking": true, "event_type": "Proposal", "start_task": "<Start Task ID vom Review Prozess>", "start_process": "<Review Prozess`

und im intermediate Unicorn Task aus dem Review-Prozess tragen wir ein: `json { "blocking": false, "event_type": "Review" }`

Dann erstellen wir über die Weboberfläche von der Middleware die EventTypes Proposal und Review, beide mit den Variablen ID und Text. Und außerdem folgende EventQueries:

```
SELECT TasUid,ProUid,ID,Text FROM Proposal
```

und

```
SELECT A.AppUid AS AppUid, A.ID AS ID, B.Text AS Text
FROM Pattern[ every A=Proposal
-> B=Review(A.ID = B.ID)]
```

Erstellt nun der *Trainee* ein Textvorschlag wird dieser als Proposal Event an Unicorn weitergeleitet. Die erste Query sorgt dann dafür, dass der Review Prozess gestartet wird und der *Supervisor* bekommt so die Möglichkeit den Text zu bearbeiten. Lässt er dann den Prozess weiterlaufen wird ein Review Event mit der gleichen ID in Unicorn erzeugt. Das sorgt dann durch die zweite Query dafür, dass der neue Text in der Variable im ursprünglichen Proposal Prozess übernommen wird und der Case dort gerouted wird.