HAW HAMBURG

BACHELORTHESIS
Julius Schulz

Ein Generalisierter Ansatz für die Verarbeitung Digitaler Ereignisdaten

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Julius Schulz

Ein Generalisierter Ansatz für die Verarbeitung Digitaler Ereignisdaten

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Angewandte Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachter: Prof. Dr. Olaf Zukunft

Eingereicht am: 31. August 2021

Julius Schulz

Thema der Arbeit

Ein Generalisierter Ansatz für die Verarbeitung Digitaler Ereignisdaten

Stichworte

Complex Event Processing

Kurzzusammenfassung

Ereignisse bergen essenzielle Informationen über Geschäftsprozesse, und können für jede Firma eine wertvolle Ressource sein. Doch existierende Ereignisprozessoren benötigen meist so viel Fachwissen, dass Erstnutzer abgeschreckt werden. In dieser Arbeit wird ein Ereignisprozessor konzipiert, der durch seine einfach verständliche aber mächtige Konfiguration selbst kleinen Unternehmen hilft, Ereignisse zu verarbeiten.

Julius Schulz

Title of Thesis

A Generalized Approach On Complex Event Processing

Keywords

Complex Event Processing

Abstract

Events convey essential information about business processes and may prove to be a valuable asset to any company. However, existing event processors require so much intricate know-how that potential users might be scared away before even trying. In this abstract, an event processor is built that uses a simple but powerful configuration syntax to help even small business profit from event processing.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Motivation

Die abraxas Verlag GmbH ist ein kleiner IT-Dienstleister aus Lüneburg, Niedersachsen. Seit 1995 werden hier Hard- und Softwarelösungen im direkten Kontakt mit Kunden vertrieben. Tägliche Geschäftsprozesse werden manuell erfasst, um diese später den Kunden als Arbeitszeiten zu berechnen. Wegen der Komplexität der Erfassung soll diese teilweise durch den Einsatz von Ereignisprozessoren automatisiert werden. Vorhandene Lösungen verlangen durch ihre Komplexität jedoch hohe Zeitinvestitionen und erfordern so unverhältnismäßig hohe Kosten, die für kleine Betriebe nicht zu tragen sind.

Ereignisse gewinnen immer mehr an Bedeutung. Vor allem mit der Vernetzung von Geräten über das Internet of Things, wird die Verarbeitung und Erkennung von Mustern innerhalb der erzeugten Ströme aus Ereignissen immer wichtiger. Potenzial bieten hier sogenannte „Event Processing Engines“ – Applikationen, die speziell für die Erkennung solcher Muster zuständig sind. Sie können über vorher definierte Verarbeitungsregeln Ereignisströme in Echtzeit verwertbar machen [20].

Bisher werden Geschäftsprozesse durch die Mitarbeiter über eine manuelle Zeiterfassung aufgezeichnet. Dabei gehen viele Informationen durch die große Flut an Aufgaben im täglichen Geschäft verloren. Die automatisierte Unterstützung soll Mitarbeiter dazu motivieren, tatsächlich genutzte zeitliche Ressourcen in die Rechnungsstellung zu übergeben.

1.2 Ziel der Arbeit

Im Rahmen dieser Bachelorarbeit soll eine Applikation entworfen werden, welche die Verarbeitung von komplexen Ereignisdaten erlaubt. Verarbeitungsregeln sollen möglichst

einfach definierbar sein, sodass eine Integration in Umgebungen auch durch unerfahrenere Anwender möglich ist.

Zuerst sollen die Beschaffenheiten von sowohl Ereignissen, als auch Ereignisprozessoren abgesteckt werden. Da ein genereller Ansatz erforderlich ist, der, falls nötig, auch außerhalb der abraxas Verlag GmbH genutzt werden könnte, muss zunächst untersucht werden, welche Anforderungen die Bachelorarbeit zu erfüllen hat.

Anschließend sollen die Anforderungen in einen Entwurf eines Ereignisprozessors überführt werden. Dabei soll ein flexibler Ansatz für die Verarbeitung von Ereignisströmen gefunden werden, der möglichst viele Prozesse unterstützen und/oder automatisieren kann.

Zuletzt wird der implementierte Ereignisprozessor in die vorhandene Umgebung der abraxas Verlag GmbH integriert, um die Mitarbeiter bei der Erfassung von Arbeitszeiten zu unterstützen. Über eine Applikation sollen Ergebnisse des Prozessors einsehbar und verwertbar dargestellt werden. Hierdurch soll die Flexibilität und Resilienz der Applikation getestet werden.

2 Grundlagen

2.1 Begriffe

Für die Verarbeitung von Ereignisdaten muss zuerst verstanden werden, was genau ein Ereignis definiert und wie Informationssysteme zur Verarbeitung dieser geformt sind. Im Nachfolgenden werden Begriffe des *Event Processing* analysiert und bewertet.

2.1.1 Ereignis

Ereignisse (oder: Events) werden definiert als „Alles, was passiert oder als etwas Passierendes angenommen werden kann“ [19]. Demnach sind sie die kleinste messbare und modellierbare Einheit, die innerhalb eines Systems auftritt. Sie können verstanden werden als zeitlich punktuell auftretende Zustandsänderungen [15] eines Systems, die nebenläufig zu Prozessen entstehen (vgl. 2.1.2).

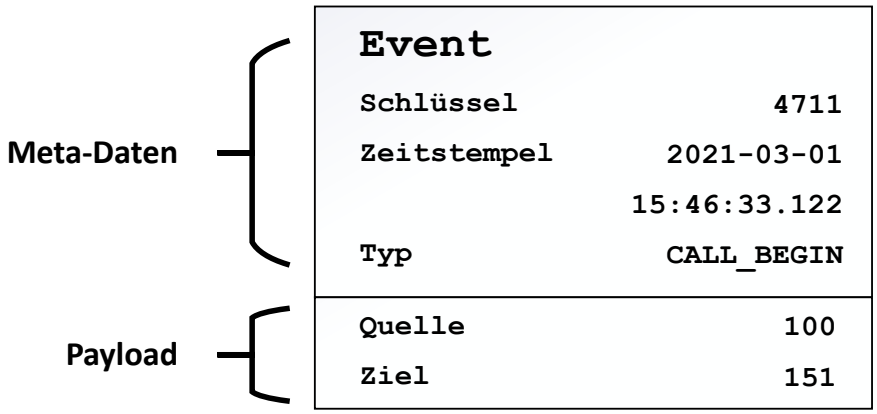


Abbildung 2.1: Anatomie eines typischen Ereignisses

Ereignisse bestehen aus einem Kopfteil (Meta-Daten) und – abhängig von dem Typ des Ereignisses – zugehörigen Nutzdaten (Payload). Ereignisse bestehen mindestens immer

aus einem eindeutigen Schlüssel, einem Zeitstempel und einem Ereignistypus (Abbildung 2.1). Diese Attribute ermöglichen eine Einordnung zeitlich- und kontextbasierter Abhängigkeiten zwischen auftretenden Ereignissen und bilden so das Fundament ereignisbasierter Auswertungen. Zusätzlicher Kontext kann durch den Anhang von Nutzdaten übertragen werden, um die Beziehungen zwischen Ereignissen zu vertiefen.

Physikalische Ereignisse lassen sich in einem digitalen System als einzelne Objekte abbilden. Ereignisse sind jedoch nicht immer einem physikalischen Prozess zuzuordnen, sondern können auch rein digital oder simuliert sein; solche Ereignisse werden als „virtuell“ bezeichnet [19].

2.1.2 Prozesse

Im Kontext ereignisbasierter Verarbeitung ist ein Prozess ein Ablauf, der zu Beginn, zwischen (bei der Beendigung von Einzelschritten) und während der Bearbeitung Ereignisse erzeugt. Prozesse können wohldefinierte Start- und Endpunkte besitzen, aber auch kontinuierlich fortgeführt werden. Die Verarbeitung von Ereignissen kann über Prozessgrenzen hinaus erfolgen, da Abhängigkeiten zwischen verschiedenen Prozessen nicht ausgeschlossen werden können.

2.1.3 Event Stream Processing

Anders als andere Anwendungen, stützen sich ereignisbasierte Algorithmen auf noch nicht vorhandenen Daten, die erst aus den Ereignissen anhand von bekannten Mustern geerntet werden müssen [15].

Die Verarbeitung wird generell als „Event Processing“ bezeichnet; werden Ereignisdaten als Strom verarbeitet, spricht man vom „Event Stream Processing“ (ESP) [15].

2.1.4 Complex Event Processing

Complex Event Processing (CEP) ist eine Erweiterung von ESP, bei dem komplexere Regeln auf den Ereignisstrom angewandt werden. CEP fokussiert sich vorwiegend auf die Echtzeit-Analyse von Prozessen, deren Zustandsraum sich schnell ändert und durch eine hohe Zahl an Variablen schwer überschaubar ist. Existierende Systeme werden in 2.2 beschrieben.

2.1.5 Ereignisprozessoren

Bietet ein System die Verarbeitung von Ereignissen in Form von ESP bzw. CEP an, handelt es sich um einen Ereignisprozessor.

2.2 Vergleichbare Softwarelösungen

2.2.1 Apache Foundation

Apache Spark

Apache Spark ist ein Framework, vorwiegend für die Batch-Verarbeitung von großen Datenmengen [28]. Ein entsprechendes Modul namens Spark Streaming erlaubt die Verarbeitung von Datenströmen, und könnte somit auch die Basis für Event Stream Processing bilden. Benutzer berichten hier jedoch von hohen Lernkurven für Erstnutzer und hohe Kapazitätsanforderungen durch die Verarbeitung im Arbeitsspeicher [29].

Apache Flink

Apache Flink ähnelt Apache Spark in seinen Anwendungen, Flink wird allerdings hauptsächlich für die Verarbeitung von Datenströmen genutzt [26]. Auch hier liegt der Fokus auf der Auswertung von Big Data, was für die Auswertung einfacherer Geschäftsprozesse eventuell ein zu großer Overhead sein könnte.

2.2.2 Cloud-Lösungen

Zusätzlich zu den zuvor erwähnten Lösungen, existieren auch einige SaaS-Lösungen[1], die für die abraxas Verlag GmbH nicht infrage kommen. Da alle intern entwickelten und verwendeten Anwendungen on-premise gehostet werden, würde eine externe Lösung eventuell hohe Bandbreite beim Senden der Ereignisse erfordern, und die Sicherheit der Daten kompromittieren.

[1] SaaS: Software as a Service [32]

3 Analyse

3.1 Kontext

Vor der Implementierung werden die Anforderungen an ein geeignetes System formuliert. Diese Anforderungen sollen in einem späteren Schritt für einen Entwurf einer passenden Architektur genutzt werden.

Der Entwurf eines Ereignisprozessors findet im Kontext der abraxas Verlag GmbH statt. Entsprechend muss darauf geachtet werden, dass die neue Applikation in vorhandene Systeme und Geschäftsprozesse eingebunden werden kann.

Im Folgenden wird der vorhandene Umgebungskontext beschrieben, der die Anforderungen bedingt.

3.1.1 Ist-Soll-Analyse

Um den Bedarf an einer neuen Lösung zu ermitteln, wird zunächst analysiert, wie der bisherige Geschäftsablauf funktioniert und welches Ziel erreicht werden soll.

Ist-Zustand

Als geschäftstragende Software nutzt die abraxas Verlag GmbH ein firmenintern entwickeltes ERP-System [4]. Ursprünglich als monolithische Applikation in Delphi [8] entwickelt, dient die Applikation der Speicherung relevanter Geschäftsdaten seit beinahe zwanzig Jahren. Inzwischen wird im parallelen Betrieb eine webbasierte, modulare Version mit Angular [13] und Laravel [18] betrieben, die als Docker-Container [6] ausgeliefert wird (gestrichelte Linie in Abbildung 3.1). Sowohl die Delphi-Applikation und die Web-Version greifen auf denselben Microsoft SQL Server [21] für die Persistierung der Daten zu.

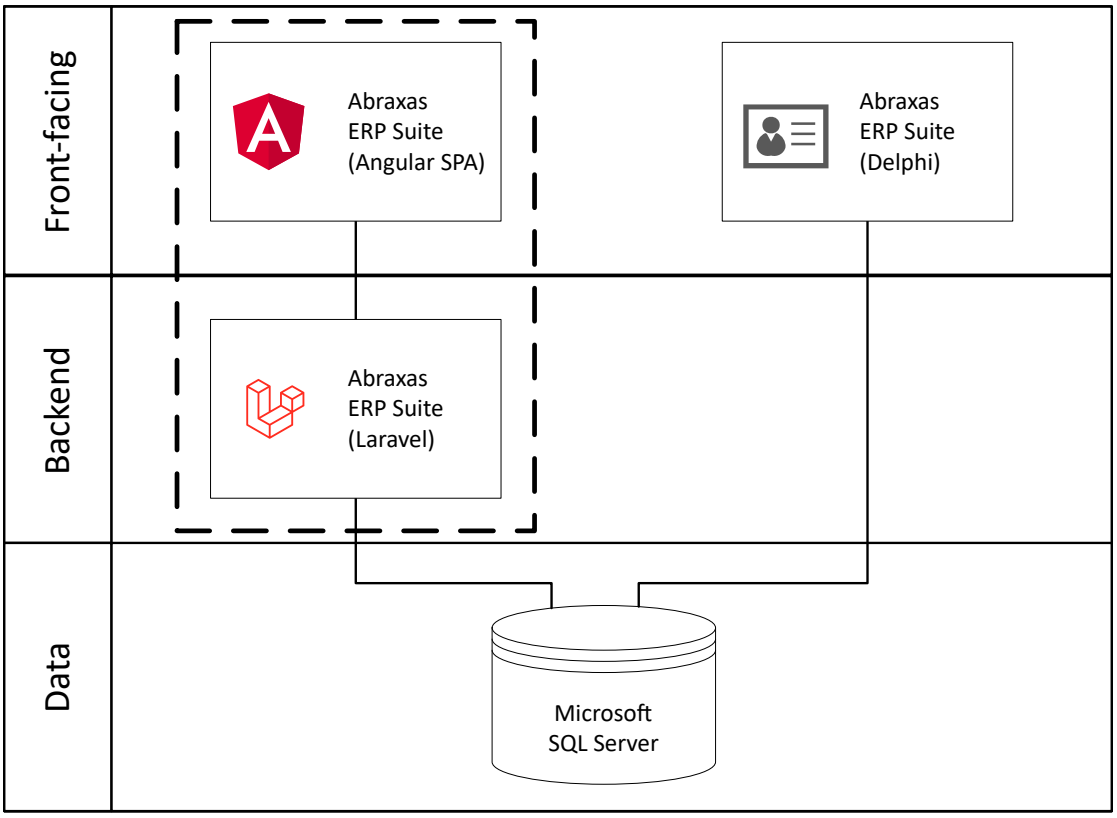


Abbildung 3.1: ERP-System der abraxas Verlag GmbH

Für die Mitarbeiter der abraxas Verlag GmbH ist die automatische Erfassung von Ereignissen kein unerforschtes Gebiet. Als Teil des ERP-Systems werden Telefonate und Fernwartungen bereits automatisch erfasst und können so von Mitarbeitern in Arbeitszeiten umgewandelt und dem Kunden in Rechnung gestellt werden. Die Daten für diese Aktivitäten werden bei direkt von der internen VoIP-Telefonanlage (3CX [1]) und dem externen Fernwartungsanbieter TeamViewer [25] regelmäßig über geplante Jobs abgeholt.

Die Kommunikation mit externen Partien wird zudem über einen intern entwickelten Helpdesk unterstützt. Tickets aus dem Helpdesk-System werden ebenfalls in Arbeitszeiten zusammengefasst. Durch die Bearbeitung der Helpdesk-Fälle werden automatisch Arbeitszeiten errechnet, welche nach manueller Prüfung durch die Mitarbeiter in die Rechnungsstellung übertragen werden.

Zudem erfassen die Mitarbeiter Aktivitäten (wie z.B. Einrichtung von Hardware, Installation von Software, etc.) selbst. Diese manuell erfassten Datensätze machen den Großteil

der erzeugten Aktivitäten aus. Im Jahre 2020 wurden[1] durchschnittlich pro Monat 100 Aktivitäten erfasst; im gesamten Jahr wurden 143 davon aus Telefonaten und Fernwartungen, und 24 aus dem Helpdesk generiert.

Soll-Zustand

Aus der Ist-Analyse beschrieben in 3.1.1, geht hervor, dass die Mitarbeiter überwiegend dazu tendieren, Arbeitszeiten manuell zu erfassen. Die Mitarbeiter sind durch die Komplexität der Arbeitszeiterfassung oft überfordert, oder verlieren getätigte Aktivitäten aus den Augen. Hier besteht die Hoffnung, dass die automatische Verarbeitung von Ereignissen die Übersicht über Aktivitäten verbessert oder Erfassungsprozesse automatisiert.

3.1.2 Business Activity Monitoring

Die abraxas Verlag GmbH profitiert von einem Ereignisprozessor auch als Werkzeug die automatische Aufzeichnung und Berechnung von Geschäftsprozessen. Ereignisse sollen genutzt werden, um Einsicht in den derzeitigen Zustand von Geschäftsprozessen zu erlangen. Zwar wird angestrebt, einen Ereignisprozessor zu entwerfen, der ein möglichst breites Spektrum an Anwendungsbereiche erlaubt; es muss jedoch sichergestellt sein, dass der Ereignisprozessor als Business Activity Monitor [16] geeignet ist.

3.2 Anforderungen

Die Anforderungen an einen Ereignisprozessor werden in Form eines Lasten- und Pflichtenheftes erfasst, welche sich aus zuvor skizzierten User Stories ergeben (Anhang A).

Die daraus resultierenden funktionalen und nichtfunktionalen Anforderungen werden hier aufgezeigt. Da es sich um ein firmeninternes Projekt handelt, sind keine externen Stakeholder in den Entwurfsprozess involviert.

[1] Überschlagsrechnung aus Datensätzen in Datenbank

3.2.1 Funktionale Anforderungen

Eingabe von Ereignissen

Der Prozessor soll in der Lage sein, auf externe Ereignisse zu reagieren und diese zu verarbeiten. Dafür soll der Aufbau der eingehenden Ereignisse vereinheitlicht werden.

Ausgabe von Ergebnissen

Der Prozessor soll, wenn zutreffend, auf eingegangene Ereignisse reagieren, indem das Ergebnis von Verarbeitungsregeln an zuvor festgelegte Endpunkte gesendet wird.

Laufzeit-Modifizierbarkeit

Durch Konfigurationen eines Benutzers sollen Regeln für die Verarbeitung von Ereignissen definiert werden können.

Die von dem Ereignisprozessoren angewandten Verarbeitungsregeln sollen zur Laufzeit des Prozessors von Benutzern dynamisch anpassbar und integrierbar sein. Dadurch können Administratoren den Prozessor flexibel an neue Gegebenheiten anpassen, und der Konfigurationsaufwand wird minimiert.

3.2.2 Nichtfunktionale Anforderungen

Containerisierung

Der Ereignisprozessor soll lauffähig in Docker-Umgebungen sein. Entsprechend muss von der Anwendung ein Docker-Image erstellbar sein. Dies erleichtert die Auslieferung nicht nur in der abraxas Verlag GmbH, sondern auch für potenzielle Kunden.

Datenbankanbindung zu Microsoft SQL Server

Als Datenbank für die geschäftskritischen Daten wird bisher ein Microsoft SQL Server genutzt. Verarbeitete Ereignisse sollen in diese Datenbank übertragen werden können, um firmeninterne Abläufe nachvollziehen zu können.

3.2.3 Qualitätsmerkmale

Um den Fokus der Bachelorarbeit auf die wesentlichen, notwendigen Eigenschaften des Ereignisprozessors zu legen, werden anhand ISO/IEC 9126-1 [3] die wichtigsten Qualitätsmerkmale für das Projekt festgelegt, die für den Entwurf und die Architektur des Ereignisprozessors berücksichtigt werden sollen.

Interoperabilität Um die Einbindung des Prozessors in vorhandene Umgebungen zu gewährleisten, muss der Prozessor Integrationen anbieten, die eine Vernetzung mit externen Applikationen ermöglichen. Dieses Merkmal wird als wichtigstes Ziel der Implementation eingeschätzt, da der Prozessor ohne Integrationen mit anderen Anwendungen nicht verwendet werden kann.

Änderbarkeit Die Anpassung von Verarbeitungsregeln soll entsprechend der funktionalen Anforderungen 3.2.1 während der Laufzeit modifizierbar sein.

Verständlichkeit Für die Definition der Verarbeitungsregeln soll eine leicht verständliche Syntax gefunden werden, die auch für Nutzer ohne Erfahrung eingängig ist.

Fehlertoleranz Fehlerzustände sollen den Ereignisprozessor nicht zum Absturz bringen, um die Verarbeitung weiterer Ereignisse nicht zu beeinträchtigen. Für die Implementierung des Prozessors werden vorerst keine Zusicherungen zur Stabilität und Wiederherstellbarkeit gestellt; das bedeutet auch, dass Ereignisse vor oder während der Verarbeitung verloren gehen könnten, wenn der Ereignisprozessor in einem instabilen Zustand läuft.

4 Entwurf

4.1 Black-Box-Verhalten

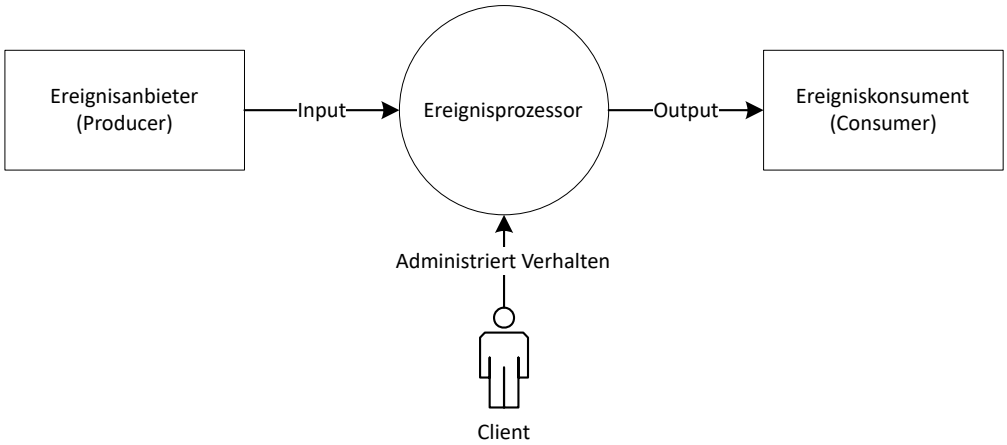


Abbildung 4.1: Kontext eines Ereignisprozessors

Um den vollen Nutzen aus Ereignisprozessoren zu ziehen, müssen sie Signale von externen Systemen erhalten, verarbeiten und weitergeben können. Um die Interoperabilität zu gewährleisten, werden zuerst die Integrationsbedingungen untersucht.

In Abbildung 4.1 wird der Kontext eines Ereignisprozessors aufgezeigt. Ereignisprozessoren agieren innerhalb verteilter Systeme als „Publisher-Subscriber“-Dienst; sie erlauben konsumierenden Systemen auf bestimmte Muster in dem Ereignisstrom der produzierenden Systeme zu lauschen. Damit agieren Ereignisprozessoren als Middleware für die Übersetzung von Ereignisströmen in konkret verwertbare Aktionen.

Benutzer, speziell Administratoren, können sich direkt mit einem Prozessor verbinden, um den aktuellen Zustand auszuwerten, oder das Verhalten des Prozessors zu manipulieren. Für eine Implementation müssen die Schnittstellen für den Ein- und Ausgang von Ereignissen abgebildet werden, um eine Einbindung in bestehende Umgebungen zu

ermöglichen. Ein administrativer Zugriff ist nicht zwingend erforderlich, erleichtert jedoch das Untersuchen von Fehlerzuständen erheblich und erlaubt die Anpassung von Verarbeitungsbedingungen ad hoc.

Im Folgenden werden Applikationen, die Ereignisse erzeugen, als *Produzenten*, empfangende Applikationen als *Konsumenten*, und das Ergebnis einer Verarbeitungsregel als *Aktion* bezeichnet.

Auch kann eine Applikation gleichzeitig sowohl Produzent als auch Konsument sein – solche Anwendungen werden als *Prosumer* bezeichnet.

4.2 Aktive / Passive Integration

Die Integration mit externen Systemen soll möglichst plattformagnostisch erfolgen, da der Ereignisprozessor möglicherweise mit einer sehr großen Anzahl an externen Systemen vernetzt wird. Um große Abhängigkeiten zu Produzenten zu vermeiden, werden Ereignisse passiv empfangen. Aktive Integrationen mit Produzenten werden für spezielle Anwendungen vorbehalten, für die keine passiven Integrationen eingerichtet werden können (zum Beispiel Drittanbieter-Applikationen).

Die Ausgabe an Konsumenten wird zunächst über aktive Integrationen in Form von Webhooks gelöst, sodass der Prozessor HTTP-Anfragen nach der Verarbeitung von Ereignissen und beim Auslösen von Aktionen versendet.

4.3 Schichtenarchitektur

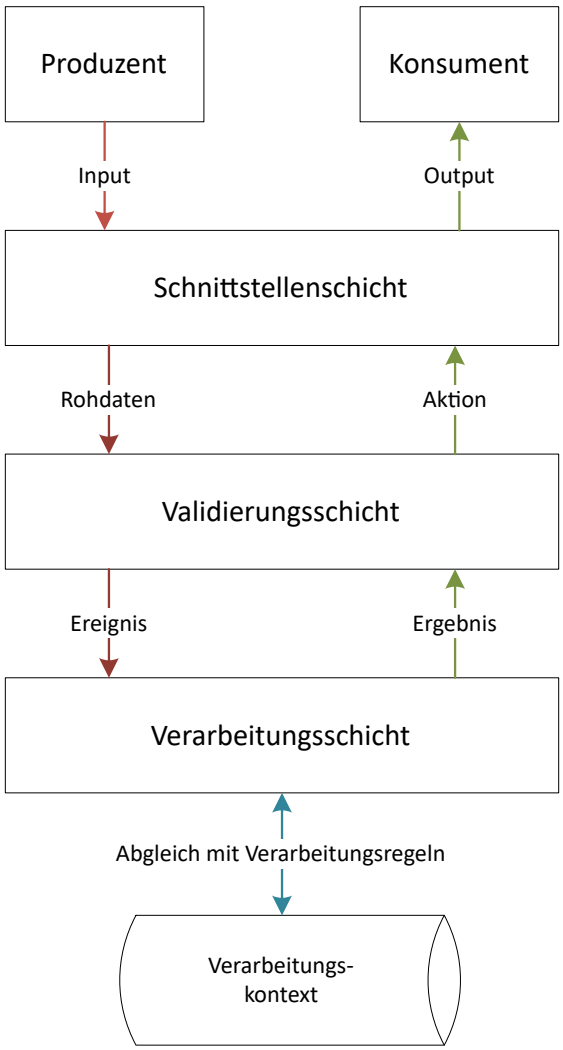


Abbildung 4.2: Schichten eines Ereignisprozessors

Damit der Prozessor möglichst modular implementiert wird, werden Komponenten in einzelne Schichten unterteilt. Hierdurch wird die zukünftige Erweiterbarkeit sichergestellt, und der Prozessor kann durch Austausch und Erweiterung einzelner Schichten leicht an neue Begebenheiten angepasst werden.

Für eine erste funktionierende Implementation wird angenommen, dass ein Ereignisprozessor mit drei Schichten implementiert werden kann (Abbildung 4.2):

4.3.1 Schnittstellenschicht

Die Schnittstellenschicht bietet Eingang- und Ausgangsschnittstellen für Produzenten und Konsumenten an. Außerdem ist sie für die Steuerung des Systems von außen, zum Beispiel durch einen Entwickler, zuständig. Es müssen also Schnittstellen für eingehende Ereignisse und administrative Befehle geschaffen werden. Ausgehende Signale werden aktiv an zuvor registrierte Konsumenten gesendet.

Um die Integration des Prozessors zu vereinfachen, wird die Eingangsschnittstelle als REST API umgesetzt, da viele Systeme bereits Webhooks für den Versand von Ereignissen unterstützen. Auch der administrative Zugriff kann hierüber erfolgen, sodass für diesen ein einfacher HTTP-Client genutzt werden kann (wie cURL [5] oder Postman [24]). Für ausgehende Signale können ebenfalls HTTP-Anfragen versandt werden, um ein gleichförmiges Format von Ein- und Ausgang zu schaffen.

In einem späteren Schritt kann die Schnittstellenschicht dann um weitere Protokolle erweitert werden. Möglich wäre hier zum Beispiel aktive Verbindungen über TCP-Sockets, die den Overhead von Ein- und Ausgang minimieren und Produzenten und Konsumenten zugleich auch passiv bei Verbindungsabbruch über eventuelle Ausfälle informiert.

4.3.2 Validierungsschicht

Die Validierungsschicht prüft eingehende und ausgehende Signale auf Korrektheit. Fehlerhafte Ereignisse oder Aktionen sollen hier abgefangen werden, um Fehlerzustände im Prozessor zu vermeiden.

Die Validierungsschicht ist außerdem für die Überprüfung von Fehlern in Verarbeitungsregeln zuständig. Syntaktische Fehler sollen hier vor dem Schreiben in den Speicher gefunden werden, damit ein Entwickler entsprechende Anpassungen vornehmen kann.

4.3.3 Verarbeitungsschicht

Als Kern des Ereignisprozessors nimmt die Verarbeitungsschicht Ereignisse entgegen, gleicht diese mit dem aktuellen Verarbeitungskontext ab, und löst entsprechende Aktionen aus Verarbeitungsregeln aus. Der Kontext wird hierbei auch von der Verarbeitungsschicht initialisiert, indem konfigurierte Verarbeitungsregeln in den Speicher – Datenbank

oder Arbeitsspeicher – geschrieben werden. Die Regeln werden dann auf eingehende Ereignisse angewandt, und der Kontext entsprechend erweitert.

Ist eine Regel entsprechend konfiguriert, wird durch die Erweiterung des Kontextes auch ein Ergebnis zurück in die Validierungsschicht getragen. Dies erlaubt die Ausführung von Aktionen nach dem Eintreffen eines Ereignisses.

4.4 Verarbeitung von Ereignissen

Die Verarbeitung der Ereignisse muss entsprechend der Anforderungen aus 3.2 durch beliebig definierbare Regeln erfolgen. Um Verarbeitungsregeln zu modellieren, wird zuerst die Verarbeitung von Ereignissen untersucht.

4.4.1 Lebenszeit einer Verarbeitungsroutine

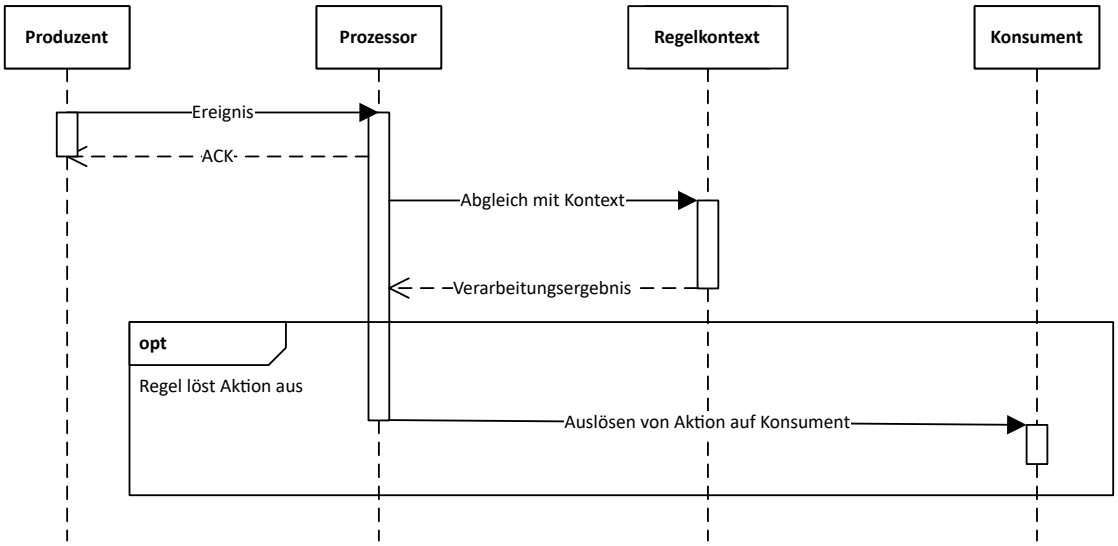


Abbildung 4.3: Lebenszeit einer Verarbeitungsroutine

Wie in Abbildung 4.3 zu sehen, wird bei Eingang eines Ereignisses von einem Produzenten der aktuelle Zustand der Verarbeitungsregeln ausgewertet und erweitert. Falls die Verarbeitungsregel dafür konfiguriert ist, wird durch das Ereignis eine Aktion ausgelöst. Daraus ergibt sich auch, dass Ereignisse kurzlebig sind: Werden sie nicht von Produzent

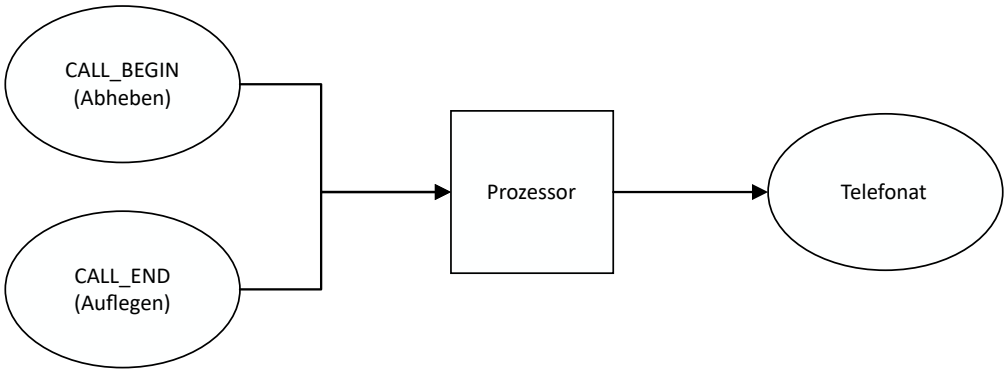


Abbildung 4.4: Beispiel: Korrelation von zwei Ereignissen eines Telefonats

oder Prozessor aufgezeichnet, existieren sie nur für die Dauer einer Abfrage an den Prozessoren. Für die Abbildung komplexer Regeln muss der Prozessor also zwingend die Daten eingehender Ereignisse aufbewahren, oder vergangene Ereignisse von den Produzenten abfragen können.

4.4.2 Korrelation von Ereignissen

Für die Verarbeitung komplexer Regeln wird betrachtet, wie Beziehungen zwischen den Ereignissen abgebildet, und entsprechend 4.4.1 persistiert werden müssen. Ereignisse sind, wie in 2.1.1 beschrieben, die kleinste modellierbare Einheit eines Systems. Dabei bilden mehrere Ereignisse durch ihre Beziehungen zueinander größere Entitäten ab. Sollen Verarbeitungsregeln definiert werden, müssen auch diese Beziehungen modelliert werden.

In Abbildung 4.4 werden die Ereignisse eines Telefonats beispielhaft aufgezeigt. Das Abheben (`CALL_BEGIN`) und das Auflegen (`CALL_END`) des Telefonhörers sollen dabei durch den Prozessor als ein zusammenhängendes Telefonat interpretiert werden. Dafür muss das Event vom Typ `CALL_BEGIN` durch eine Regel mit `CALL_END` in Beziehung gesetzt werden; außerdem müssen Payload-Daten durch Bedingungen abgeglichen werden, sodass sichergestellt ist, das beide Ereignisse dem gleichen Telefonat angehören (zum Beispiel müssen beide Telefonate der gleichen Nebenstelle zugeordnet sein).

Hierfür ist auch die zeitliche Reihenfolge zu beachten: Da Ereignisse nebenläufig zu Geschäftsprozessen entstehen (2.1.2), entspricht die zeitliche Abfolge eingegangener Ereignisse auch dem zeitlichen Ablauf der Einzelschritte des Prozesses. In dem obigen Beispiel bedeutet dies, das ein `CALL_END` immer auf einen `CALL_BEGIN` folgen muss.

Auch Verarbeitungsregeln, die eine dynamische Anzahl an Ereignissen erwarten, müssen betrachtet werden. Beispielsweise kann eine Anforderung sein, dass nicht nur einzelne Anrufe erfasst werden, sondern viele Telefonate eines Zeitraums. Der Kontext einer solchen Regel kann eine beliebige Größe annehmen, weswegen für entsprechenden Speicherplatz gesorgt werden muss.

4.4.3 Verarbeitung von Ereignissen mit Graphen

Vor der Implementierung einer konkreten Architektur wird das „Complex Event Processing“ genauer untersucht. Da die Erkennung komplexer Muster nach gegebenen Verarbeitungsregeln benötigt wird, muss eine entsprechende Datenstruktur für das Pattern Matching gefunden werden. Der Ansatz hierfür wird von einer Struktur abgeleitet, die in vielen Bereichen der Softwareentwicklung bereits für das Erkennen von Mustern verwandt wird: reguläre Ausdrücke. Dafür wird als Datenstruktur der deterministische endliche Automat (DEA) für die Verarbeitung der Ereignisse erwägt. Da reguläre Ausdrücke als DEA abgebildet werden können [17], sind sie ein geeignetes Modell für die Erkennung von Mustern in Ereignisströmen [14].

Angenommen, eine Telefonanlage erzeugt drei Ereignisse mit

- `CALL_BEGIN` beim Aufheben des Hörers einer Nebenstelle
- `CALL_DROPPED` bei Verbindungsabbruch
- `CALL_END` beim Auflegen des Hörers einer Nebenstelle

und alle Telefonate (auch abgebrochene) sollen von einer Verarbeitungsregel erkannt werden.
Sei L eine reguläre Sprache über dem Alphabet $\Sigma = \{ C_B, C_D, C_E \}$ mit

- Symbol C_B als Repräsentant für das Ereignis `CALL_BEGIN`
- Symbol C_D als Repräsentant für das Ereignis `CALL_DROPPED`
- Symbol C_E als Repräsentant für das Ereignis `CALL_END`

Dann definiert der reguläre Ausdruck $R = C_B(C_D|C_E)$ der Sprache L die Wörter, die gültige Telefonate erzeugen.

Abbildung 4.5: Beispiel: Regulärer Ausdruck für die Erkennung von Telefonaten

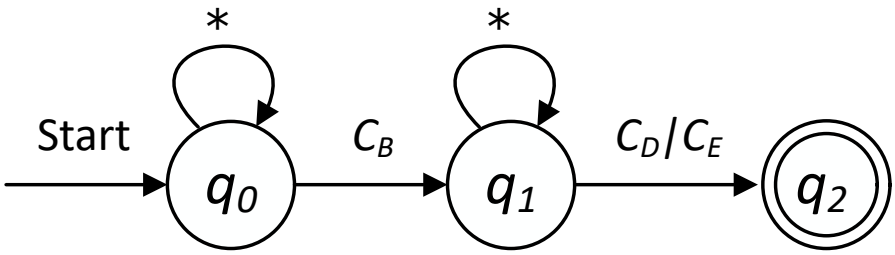


Abbildung 4.6: Beispiel: DEA für Abbildung 4.5

In Abbildung 4.5 wird eine reguläre Sprache über die Ereignisse einer Telefonanlage definiert. Die Wörter des regulären Ausdrucks R beginnen dabei immer mit dem Aufheben eines Telefonhörers, und enden entweder mit einem Abbruch des Telefonats oder dem Auflegen des Telefonhörers. Für die Verarbeitung eines oder mehrerer Ereignisströme eignen sich DEA also auch, da das sequentielle Einlesen von Zeichen eines Alphabets genau auch der Verarbeitung einzelner eintreffender Ereignisse entspricht [14]. Die Ereignisse werden dabei als einzelne Symbole einer formalen Sprache betrachtet.

Beispielhaft wird aus dem regulären Ausdruck R in Abbildung 4.5 ein deterministischer endlicher Automat konstruiert. Dafür werden die Zustände $S = \{\ q_0,\ q_1,\ q_2\ \}$ definiert, welche entsprechend R mittels Eingaben erreicht werden können. Dieser Automat kann die von einer Telefonanlage ausgegebenen Ereignisse in Relation bringen und in Aktionen – hier Telefonate – umwandeln.

Der in Abbildung 4.6 gezeigte Automat reicht jedoch noch nicht, um komplexe Regeln abzubilden. Bisher können nur zeitliche Abfolgen von Ereignissen als Muster erkannt werden. Für die Verarbeitung ist es allerdings notwendig, dass die Nutzdaten der korrelierenden Ereignisse abgeglichen werden können. Für eine generische Verarbeitungsregel mit den drei Ereignissen $\Sigma = \{\ a,\ b,\ c\ \}$ und dem Ereignisstrom mit $E_\Sigma = \{\ a_1, a_2, b_1, c_1, b_2, c_2\ \}$ muss der Prozessor nur diejenigen Ereignisse betrachten, die in ihren Nutzdaten nach bestimmten Bedingungen übereinstimmen [15]. Konkret muss der Prozessor zum Beispiel gleichzeitig auftretende Telefonate durch verschiedene Nebenstellen unterscheiden können. Dafür müssen über den Strom E_Σ die zwei Ergebnisse $\{\ a_1, b_1, c_1\ \}$ und $\{\ a_2, b_2, c_2\ \}$ gebildet werden. Hierdurch entsteht ein Problem für die Nebenläufigkeit: Da auch mehrere Pfade des Automaten zur gleichen Zeit traversiert werden können, müssen mehrere Kopien des Automaten erzeugt werden, um den Verarbeitungskontext der Regel zu speichern.

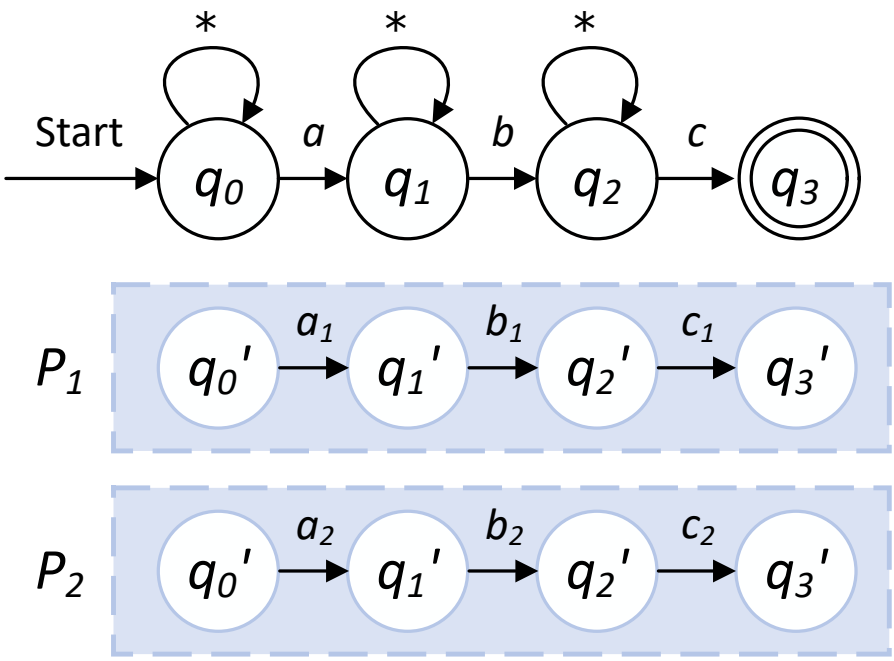


Abbildung 4.7: Automat für Ereignisse $\Sigma = \{ a, b, c \}$ mit Ergebnispfaden P_1, P_2

Um dieses Problem zu vermeiden, wird das bisherige Modell stattdessen erweitert, um die Speicherung der traversierten Pfade in eigene Teilgraphen auszulagern, die einen „aktivierten Pfad“ eines Automaten repräsentieren (Abbildung 4.7). Für jeden Zustand des Automaten wird die Aktivierung des Zustands nicht direkt am Automaten gespeichert, sondern an den Pfad der bereits traversierten Zustände angehangen. Die so entstehenden Ergebnispfade bilden die von dem Ereignisprozessor empfangenen, korrelierenden Ereignisse in der genauen zeitlichen Abfolge ab. Durch diese Modellierung soll nicht nur die Implementierung von Regeln, sondern auch das Auslesen des Zustands einer Regel für Anwender vereinfacht werden.

4.4.4 Nachteile deterministischer endlicher Automaten

Genau wie reguläre Ausdrücke können Verarbeitungsregeln abgebildet als deterministische endliche Automaten nicht genutzt werden, um Abhängigkeiten zwischen der Anzahl von Ereignistypen zu definieren; für zwei Ereignisse a, b kann keine Regel definiert werden, die genau dann auslöst, wenn genau so viele Ereignisse a als Eingabe erfolgt sind wie Ereignisse b [17]. Dieser Nachteil wird zugunsten der einfacheren Implementation

akzeptiert, mit der Aussicht, dass eine spätere Version des Ereignisprozessors durch die Funktionalitäten eines Kellerautomaten erweitert werden könnte.

4.4.5 Ausgabe von Aktionen

Die Anforderungen an das Ergebnis einer Verarbeitungsregel hängt stark von dem jeweiligen Konsumenten ab. Um die Integration von Konsumenten zu vereinfachen, wird beim Auslösen einer Aktion der gesamte Verarbeitungskontext an den Konsumenten gesendet. Das bedeutet, dass Konsumenten nach der Erkennung einer Regel nicht nur ein einfaches Signal erhalten, sondern auch den Pfad der Ereignisse, die den auslösenden Zustand erreicht haben.

5 Implementierung

5.1 Umgebung

Bevor die endgültige Architektur festgelegt wird, werden zunächst die Anforderungen aus der Analyse und dem Entwurf betrachtet, um die Umgebung für das zu entwickelnde System zu erarbeiten.

5.1.1 Wahl der Programmiersprache

Als Programmiersprache für die Implementierung werden drei Programmiersprachen in Betracht gezogen:

PHP

Die vorwiegend für webbasierte Projekte genutzte Programmiersprache wird bereits für andere Applikationen innerhalb der abraxas Verlag GmbH verwandt (siehe 3.1.1). Hier bestehen allerdings Bedenken zur Laufzeitperformanz – PHP ist eine dynamische, interpretierte Sprache. Auch wird Multithreading in PHP 8 weitestgehend nicht unterstützt. Da mit einer hohen Anzahl gleichzeitig zu verarbeitender Ereignisse zu rechnen ist, werden stattdessen kompilierte, performantere Sprachen in Betracht gezogen.

Rust

Die noch relativ junge Sprache eignet sich zwar durch die gute Performanz und Unterstützung von Nebenläufigkeit theoretisch sehr gut für einen Ereignisprozessor, bei der Entwicklung eines Prototyps scheitert es aber an der noch geringen Unterstützung von Integrationen von Drittanbietern. So fehlten hier bereits Bibliotheken für die Verbindung

mit Neo4j oder Microsoft SQL Server[1]. Zudem fehlen für Rust die Erfahrungen mit der Programmiersprache, wodurch zusätzliche Zeit zum Erlernen von Grundkenntnissen und Paradigma benötigt werden würde.

Java

Schlussendlich wird als Programmiersprache für die Entwicklung des Prozessors Java gewählt. Java eignet sich durch die hohe Verfügbarkeit von Standardbibliotheken und Plattformunabhängigkeit gut für die Integration in vorhandene Umgebungen. Als statisch typisierte, kompilierte Programmiersprache kann auch eine höhere Performanz als mit interpretierten Sprachen erzielt werden. Außerdem ist die Programmiersprache schon bekannt und muss nicht mehr erlernt werden.

Als Laufzeitumgebung wird die freie Implementierung der Java Platform[2] OpenJDK in der aktuellen[3] Version 16 verwendet.

5.1.2 Apache Maven & Spring

Projektspezifische Einstellungen und der automatische Download von Abhängigkeiten werden durch Apache Maven [27] gehandhabt. Hierdurch wird sichergestellt, dass die Kompilierung der Anwendung reproduzierbar und zukünftige Entwicklung und Wartung vereinfacht wird.

Das Grundgerüst für die Anwendung wird durch das Java Spring Framework [34] realisiert. Zudem bietet das Framework einige Module, die die Verwendung von externen Applikationen in Javakomponenten abstrahiert – für dieses Projekt von Relevanz ist dabei *Spring Data Neo4j* [23] und *Spring for Apache Kafka* [31].

5.1.3 Graphdatenbank Neo4j

Für die Speicherung des Verarbeitungskontexts beim Abarbeiten von Ereignissen wird – orientiert an den konzipierten Datenstrukturen aus 4.4.3 – die Graphdatenbank *Neo4j* [22] verwendet. Durch die Nähe der Datenstruktur an den deterministischen endlichen

[1] Stand April 2021
[2] Standard Edition
[3] Stand August 2021

Automaten für die Verarbeitung von Ereignissen aus 4.4.3 können diese direkt in der Datenbank abgebildet werden.

5.1.4 Apache Kafka

Die Skalierbarkeit soll durch die Spaltung des Ereignisprozessors in Koordinator- und Arbeiterprozesse sichergestellt werden. Durch die parallele Verarbeitung von Ereignissen können Blockaden durch lange Verarbeitungszeiten vermieden werden.

Die Kommunikation zwischen den Prozessen wird über Apache Kafka implementiert. Kafka ist eine Applikation für Event Streaming, welche eingehende Nachrichten als Publish-subscribe-Dienst an registrierte Anwendungen verteilt. Für den Ereignisprozessor dient Kafka als „Ereignisbus“, über den eingehende Ereignisse an Arbeiterprozesse verteilt und ausgelöste Aktionen an einen Koordinator zurückgegeben werden. Hierdurch können neue Koordinatoren und Arbeiter dynamisch zur Laufzeit des Prozessors hinzugefügt werden.

Die in dem Entwurf 4.4.3 beschriebene Verarbeitung von Ereignissen ist noch nicht für die parallele Verarbeitung ausgelegt, da bisher davon ausgegangen wurde, dass Ereignisse immer in zeitlicher Reihenfolge nacheinander verarbeitet werden. Für die weitere Implementierung wird zunächst von einem einzelnen Arbeiterprozess ausgegangen.

Kafka erlaubt die Einrichtung sogenannter „Topics“ (zu Deutsch: Themen), über welche Nachrichten an andere Applikationen versandt werden können, die auf die vorher definierten Topics lauschen. Dafür müssen sowohl der publizierende als auch erhaltende Dienst an dem gleichen Kafka-Server angemeldet sein.

5.1.5 Git

Für die Versionierung des Programmcodes wird Git mit der selbst-gehosteten Variante von GitLab [10] genutzt. Hierüber sollen später auch automatische Tests ausgeführt werden.

5.2 Datenstrukturen

5.2.1 Rule

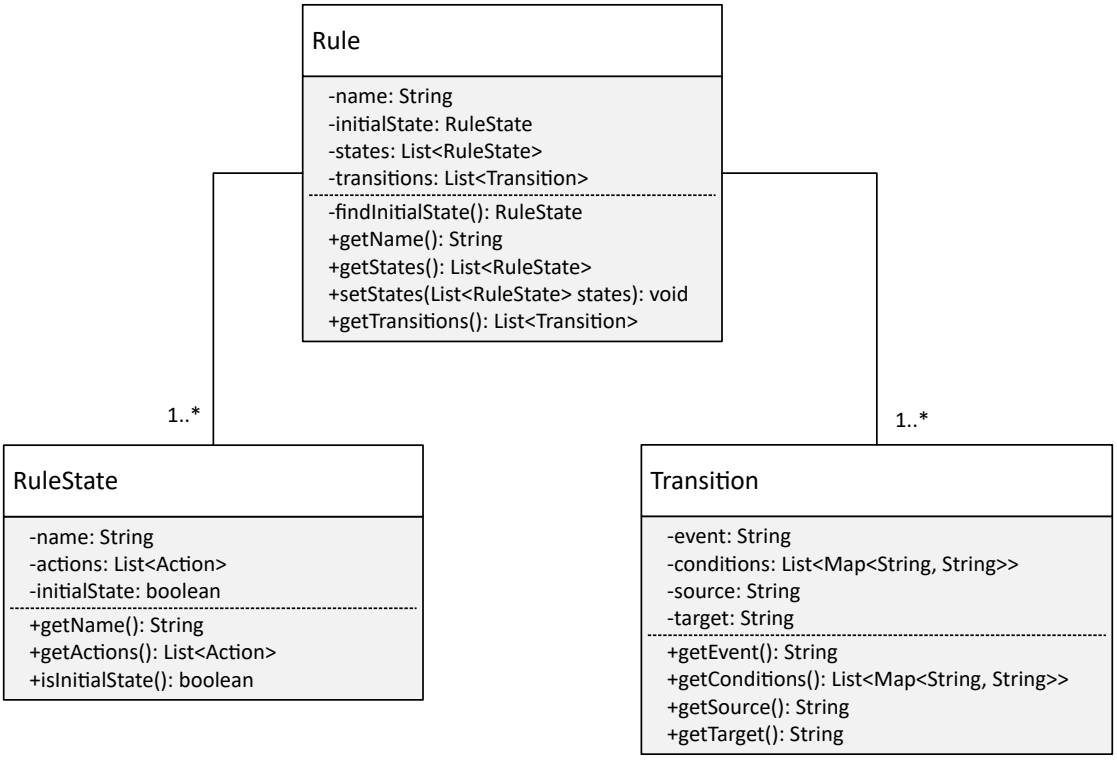
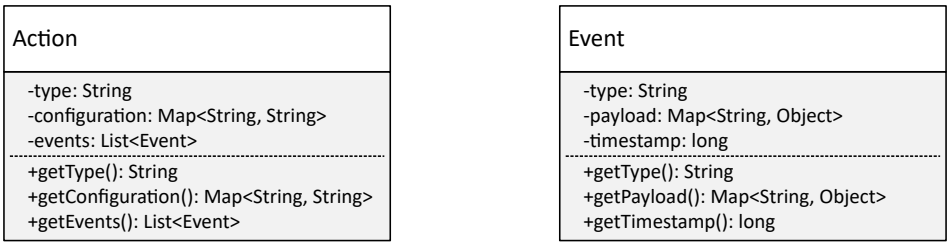


Abbildung 5.1: Klassendiagramm einer Verarbeitungsregel

Verarbeitungsregeln für Ereignisse werden als Klasse `Rule` abgebildet. Jede Regel besitzt einen eindeutigen Namen, eine Liste von Zuständen und eine Liste von Transitionen, die zwischen den Zuständen verlaufen. Ein Zustand `RuleState` besteht aus einem eindeutigen Namen, einer Liste von Aktionen, die bei Aktivierung ausgeführt werden, und einem Flag, das definiert, ob es sich um einen initialen Zustand handelt. Eine Transition `Transition` wiederum besteht aus den Eingabewerten, die für das traversieren benötigt werden: die Art des Ereignisses, eine Liste von zusätzlichen Einschränkungen des Payloads, die Quelle und die Senke. Dadurch stellen `Rule`-Instanzen deterministische endliche Automaten dar, die durch die Eingabe von Ereignissen gemäß Entwurf 4.4.3 traversiert werden.



(a) Action (b) Event

Abbildung 5.2: Klassen Action und Event

Konfiguration von Verarbeitungsregeln

Die Definition der Verarbeitungsregeln wird in Konfigurationsdateien gespeichert.

Als Dateiformat wird YAML gewählt, da dieses eine gute Leserlichkeit vorweist und so von Erstnutzern leicht verstanden werden kann. Die YAML-Dateien werden mittels Jackson [9] geparst. Ein Beispiel für eine Konfigurationsdatei wird in Anhang B gezeigt.

5.2.2 Action

Für jeden Zustand einer Verarbeitungsregel können Aktionen definiert werden, die bei Aktivierung ausgelöst werden. Eine solche Aktion ist zum Beispiel ein Webhook, der die gesammelten Ereignisse für die Aktivierung an ein definiertes Ziel sendet. Aktionen werden in der Klasse `Action` gekapselt und bestehen aus einem Typ, einer Map mit zusätzlichen Konfigurationswerten und der Liste von Ereignissen.

5.2.3 Event

Ereignisse werden durch die `Event`-Klasse abgebildet. Jedes Ereignis ist einem Typen zugeordnet, besitzt eine Map aus Nutzdaten und hat einen Zeitstempel. Der Zeitstempel wird als Unix time[4] gespeichert.

[4] Anzahl der Millisekunden, die seit dem 1. Januar 1970, 00:00:00 UTC vergangen sind

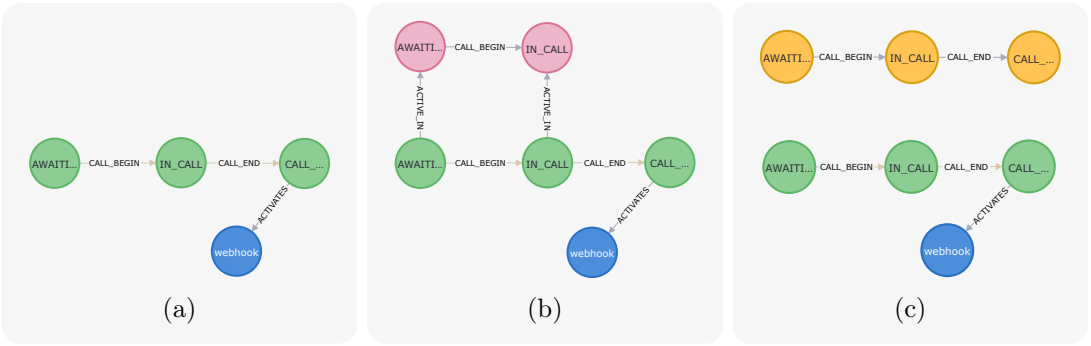


Abbildung 5.3: Aufbauender Ergebnispfad
(Grün: Zustand, Blau: Aktion, Rot: aktivierter Zustand, Gelb: Ergebnispfad)

5.3 Abbildung von deterministischen endlichen Automaten in Neo4j

5.3.1 Datenstrukturen in Neo4j

In Neo4j werden zur Modellierung von Daten zwei Datenstrukturen unterschieden: „Nodes“ (Knoten) und „Relationships“ (Kanten). Jeder Knoten und jede Kante kann dabei mit einem „Label“ versehen werden, durch das ähnliche Elemente gruppiert werden können. Zusätzlich zu dem Label können für jedes Element im Graphen auch „Properties“ (Eigenschaften) gespeichert werden.

Graphen in Neo4j sind immer gerichtet; für die deterministischen endlichen Automaten, die abgebildet werden sollen, ist dies jedoch auch explizit erforderlich [17].

5.3.2 Verarbeitungsregeln als Graphen

Verarbeitungsregeln werden in die Datenbank geschrieben, indem jeder Zustand als ein Knoten mit dem Label „`state`“ gespeichert wird. In die Eigenschaften der Knoten werden die Einstellungen der Zustände geschrieben: Jedem Knoten wird der eindeutige Zustandsname zugeordnet, und ob es sich um einen initialen Zustand handelt. Aktionen, die ein Knoten bei Aktivierung auslöst, werden in einem eigenen Knoten mit dem Label „`action`“ gespeichert. Dieser hält in seinen Eigenschaften alle für die Aktionen relevanten Einstellungen.

Die Zustandsübergänge werden als Kanten vom Typ „TRANSITIONS_TO“ zwischen den „state“-Knoten modelliert. Bedingungen für die Traversierung werden in die Eigenschaften der Kante geschrieben. Dazu zählen der benötigte Ereignistyp, das Zeitfenster für die Aktivierung, und Payload-Bedingungen, die von vorhergehenden Ereignissen abhängig sind.

Das Ergebnis der Modellierung für eine einfache Regel aus Anhang B ist in der Abbildung 5.3a zu sehen (Hinweis: Die Eigenschaften der einzelnen Elemente sind für eine vereinfachte Darstellung nicht sichtbar).

5.3.3 Pfadweise Aktivierung von Zuständen

Aktivierte Zustände werden durch das Kopieren des Zustands in einen neuen Knoten mit dem Label „active_state“ modelliert, der anschließend durch eine Kante vom Typ „ACTIVE_IN“ mit dem Ursprungszustand verbunden wird.

Beim Eintreffen eines Ereignisses werden folgende Schritte abgearbeitet:

1. Zuerst werden alle zuvor aktivierten Pfade gesucht, die durch das neue Ereignis erweitert werden könnten.

2. Wurden keine passenden Pfade gefunden, werden stattdessen initiale Zustände gesucht, die mit dem Ereignis aktiviert werden können. Die Priorisierung von Pfaderweiterungen verhindert Redundanzen, die durch doppelte Verarbeitung eines Ereignisses an derselben Verarbeitungsregel auftreten würden.

3. Für alle aktivierbaren Zustände aus Schritt 1 bzw. 2 werden die Bedingungen der zu traversierenden Kante auf das Ereignis angewandt. Schlägt eine Bedingung fehl, wird die Aktivierung abgebrochen, und das Ereignis ignoriert.

4. Wurden aktivierbare Zustände gefunden und sind alle Bedingungen erfüllt, werden neue aktivierte Zustände in der Datenbank erzeugt und an den Automaten gehangen. Handelt es sich bei dem aktivierten Zustand um einen Initialzustand, wird auch der Folgezustand aktiviert, um den Ergebnispfad zu erzeugen (Abbildung 5.3b).

5. Pfade, die einen Endzustand erreichen – d.h. einen Zustand ohne weitere ausgehende Kanten –, werden von den zugehörigen Zuständen der Verarbeitungsregel abgeschnitten (Abbildung 5.3c). Somit werden Graphen möglichst klein gehalten,

um die Performanz von Neo4j durch Traversierung großer Graphen nicht zu beeinträchtigen.

Dieses Vorgehen bildet die im Entwurf 4.4.3 betrachteten deterministischen endlichen Automaten mit mehreren Ergebnispfaden ab.

5.4 Schichtenarchitektur

Aus der Schichtenarchitektur beschrieben im Entwurf 4.3 werden die drei Schichten übernommen. Dafür werden die Verantwortlichkeiten der Schichten in einzelne Pakete separiert.

5.4.1 Bridge

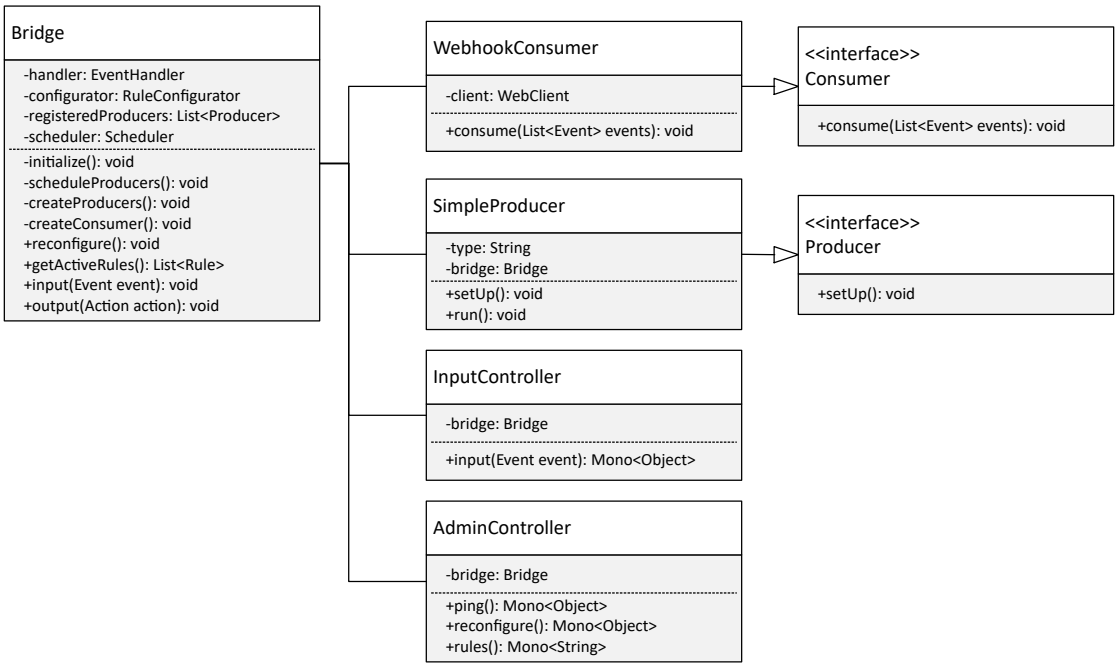


Abbildung 5.4: Klassendiagramm Bridge

Als Schnittstellenschicht dienen die Klassen des `Bridge`-Paketes der Kommunikation mit externen Diensten. Dieses besteht aus einem `Bridge`-Service, der für das Management des Ein- und Ausgangs verantwortlich ist, Klassen für die Abstrahierung von

Konsumenten und aktiven Produzenten, sowie verschiedenen Controllern für den Zugriff auf den Prozessor über HTTP.

Um die Kommunikation mit externen Systemen über HTTP zu realisieren wird der Spring Web Starter [30] über die Maven-Abhängigkeiten importiert.

Bridge-Service

Als Kernstück der Schnittstellenschicht ist die `Bridge` verantwortlich für die Weitergabe von Ereignissen in tiefere Schichten und dem Auslösen von Aktionen nach der Verarbeitung dieser.

Consumer

Aktionen werden von der Bridge zunächst in `Consumer` anhand ihres Typs umgewandelt. Eine Aktion vom Typ „Webhook“ wird so zum Beispiel zu einem `WebhookConsumer` umgewandelt. Diese bekommen die Liste der bisher gesammelten Ereignisse über die Methode `consume` übergeben um, entsprechend ihrer Konfiguration, eine Aktion auszulösen.

InputController

Für die Eingabe von Ereignissen wird ein `InputController` mit einer `/input`-Route implementiert. Hierüber können Ereignisse in den Prozessor für die Verarbeitung im JSON-Format gegeben werden. Aus dem JSON-String werden Instanzen der `Event`-Klasse über Jackson erzeugt.

Producer

Zusätzlich zu diesem „passiven“ Lauschen auf eingehende Ereignisse können über die Spring-Konfiguration noch weitere „aktive“ Ereignisproduzenten registriert werden. Zukünftig können so regelmäßig Ereignisse aus externen Quellen gelesen werden; vorerst wird jedoch ein `SimpleProducer` implementiert, der bei Aktivierung ein festes, definiertes Ereignis auslöst. Zusammen mit dem Spring `Scheduler` lassen sich hierdurch

Ereignisse für den Ablauf von Zeit definieren (bspw. „HOUR_PASSED“ oder „DAY_PASSED“).

AdminController

Der AdminController bietet Schnittstellen für die Verwaltung des Prozessors an. Administrative Benutzer können hierüber die Neukonfiguration von Verarbeitungsregeln anstoßen.

5.4.2 Ereignisbus

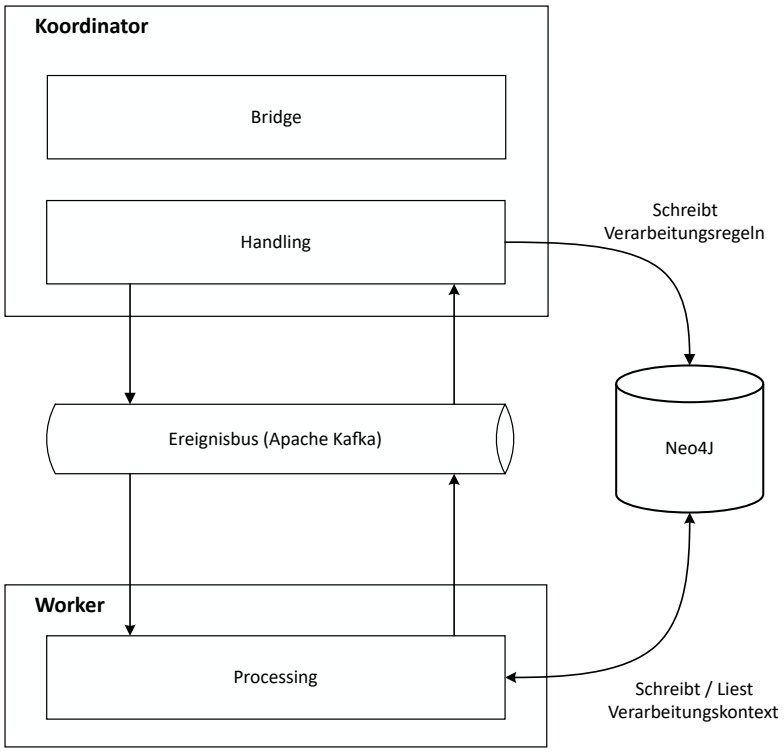


Abbildung 5.5: Aufteilung Koordinator- / Worker-Prozesse

Um die zukünftige Skalierbarkeit sicherzustellen wird die Verarbeitungsschicht in eine zusätzliche Applikation („Worker“) ausgelagert. So sollen viele zeitgleich auftretende Ereignisse parallel verarbeitet werden können. Dafür wird in der Haupt-Applikation („Koordinator“) ein Kafka Topic „EVENT_BUS“ angelegt, über welches eingehende Ereignisse an einen registrierten Worker verteilt werden.

Zusätzlich sollen Worker den Koordinator bei Aktivierung eines Zustands einer Verarbeitungsregel über den gleichen Kanal benachrichtigen. Auch hierfür wird ein Kafka Topic „RULE_BUS“ angelegt, das Koordinatoren bei oder nach der Verarbeitung von Regeln benachrichtigt. Hierdurch kann ein Prozessor auch aus mehreren Koordinator-Instanzen bestehen.

5.4.3 Handling

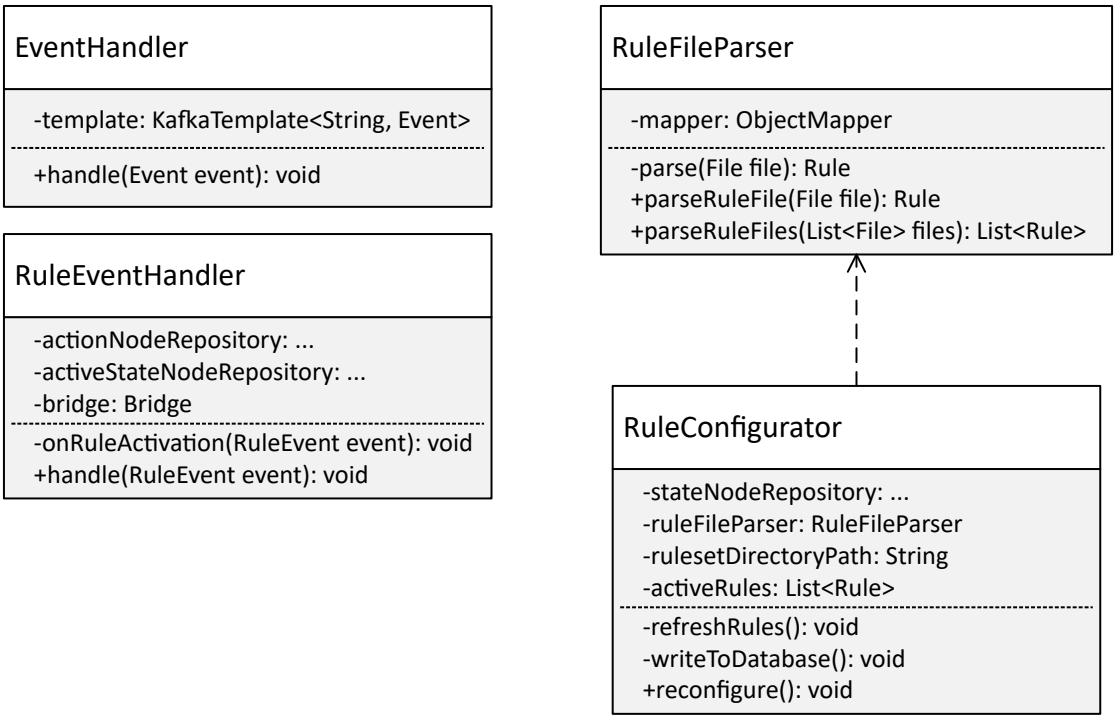


Abbildung 5.6: Klassendiagramm Handling

Im Handling werden eingehende Ereignisse für die Verarbeitung, und Aktionen für die Auslösung über die Bridge vorbereitet. Außerdem werden hier - angestoßen von der Bridge - die in Konfigurationsdateien definierten Verarbeitungsregeln in die Datenbank geschrieben.

Spring Data Neo4j

Für den Datenbankzugriff wird Spring Data Neo4j [23] genutzt. Die Datenstrukturen der Verarbeitungsregeln werden hierfür in Model-Klassen umgewandelt, dessen Struktur

entsprechend Neo4j angepasst ist. Beispielsweise werden die Auslösebedingungen einer `Transition` in JSON umgewandelt, da Neo4j für Eigenschaften von Knoten oder Kanten nur primitive Datentypen erlaubt.

EventHandler

Der `EventHandler` empfängt `Event`-Instanzen der Bridge und veröffentlicht diese auf dem Ereignisbus, um die Verarbeitung durch einen Worker zu starten. Eine zukünftige Version des Prozessors könnte hier die Einrichtung von Middleware erlauben, die Ereignisse vor der Verarbeitung abändert oder konvertiert.

RuleEventHandler

In dem `RuleEventHandler` werden Verarbeitungsereignisse vom Ereignisbus gelesen und entsprechend behandelt. Bei Aktivierung eines Regelzustands werden hier die zugehörigen Aktionen – falls vorhanden – aus der Datenbank gelesen, mit dem Ergebnispfad angereichert, und an die Bridge weitergegeben.

RuleConfigurator

Über den `RuleConfigurator` werden die aktiven Verarbeitungsregeln des Prozessors konfiguriert. Dafür wird in den Applikationseinstellungen der Pfad zum Verzeichnis mit allen Verarbeitungsregeln festgelegt. Beim Aufruf der `/reconfigure`-Route der Bridge werden alle im Verzeichnis gespeicherten YAML-Dateien an den `RuleFileParser` übergeben. Dadurch werden diese in `Rule`-Instanzen umgewandelt, welche anschließend in entsprechende Model umgewandelt und über das `StateNodeRepository` in der Datenbank gespeichert werden.

5.4.4 Processing

Die Verarbeitungsschicht wird im `Processing`-Paket des Workers umgesetzt. Hier werden eingehende Ereignisse mit dem Verarbeitungskontext gemäß Entwurf 4.3.3 verglichen.

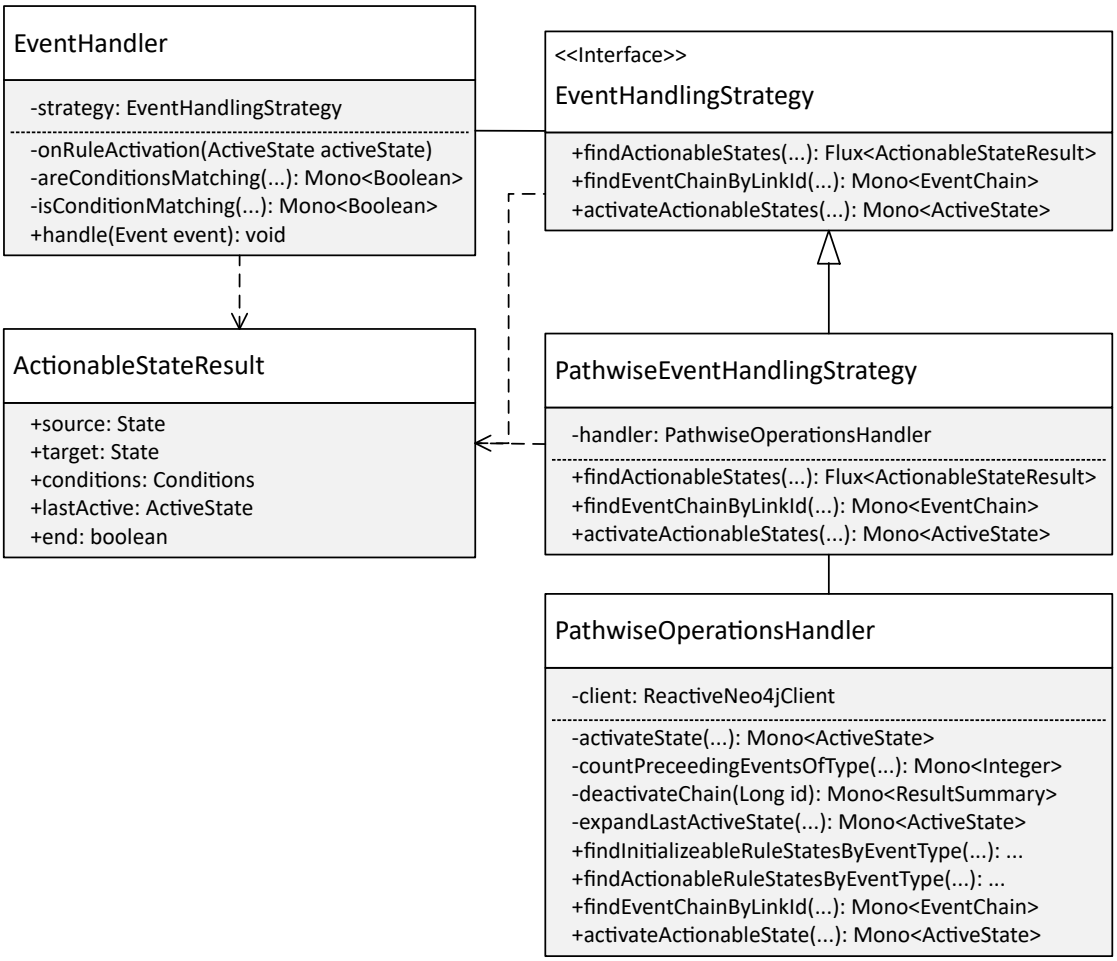


Abbildung 5.7: Klassendiagramm Processing

EventHandler

Der `EventHandler` lauscht auf eingehende Ereignisse über den Ereignisbus. Empfangene Ereignisse werden über eine `EventHandlingStrategy` verarbeitet. Dafür werden zuerst die Zustände von Verarbeitungsregeln gefunden, die aktivierbar sind. Ergebnisse werden im Data Transfer Object `ActionableStateResult` gespeichert, dessen Bedingungen anschließend durch den `EventHandler` überprüft werden. Alle geprüften, aktivierbaren Zustände werden zuletzt zurück an die `EventHandlingStrategy` für die Aktivierung übergeben.

PathwiseEventHandlingStrategy & PathwiseOperationsHandler

Entsprechend der beschriebenen Verarbeitung von Ereignissen aus 5.3 wird eine `PathwiseEventHandlingStrategy` implementiert, die aus eingehenden Ereignissen Ergebnispfade über den deterministischen endlichen Automaten in der Datenbank erzeugen. Abfragen an die Datenbank werden dafür in einen `PathwiseOperationsHandler` abstrahiert, der über den `ReactiveNeo4jClient` von Spring Data Neo4j Cypher-Anfragen[5] an die Datenbank sendet.

5.5 Deployment

5.5.1 Containererzeugung von Koordinator und Worker

Für Koordinator- und Worker-Prozess werden Docker-Images erzeugt. Prozesse des Ereignisprozessors werden mittels Supervisord [2] bei fatalen Fehlern automatisch neu gestartet, um die Resilienz erhöht.

Im Worker wird eine Startroutine implementiert, durch welche die Ausführung von Spring verzögert wird, bis der konfigurierte Koordinator-Prozess auf eine HTTP-Anfrage an eine `/ping`-Route antwortet. Dadurch wird gewährleistet, dass der Koordinator die Kafka Topics erzeugen kann, bevor sich der Worker verbindet, um Synchronisationsprobleme zu vermeiden.

5.5.2 Kafka & Neo4j-Container

Apache Kafka wird speziell für den Ereignisprozessor als Docker-Image verpackt, damit der Message Broker zusammen mit dem Prozessor ausgeliefert werden kann. Neo4j wird über das offizielle Docker-Image aus dem Dockerhub [7] ausgeliefert.

5.5.3 Verknüpfung mit ERP-System

Das vorhandene ERP-System aus 3.1.1 wird als Proxy für den Ereignisprozessor genutzt. Dafür werden in dem Backend des ERP-Systems neue HTTP-Routen implementiert, die

[5] Cypher: SQL-Pendant von Neo4j

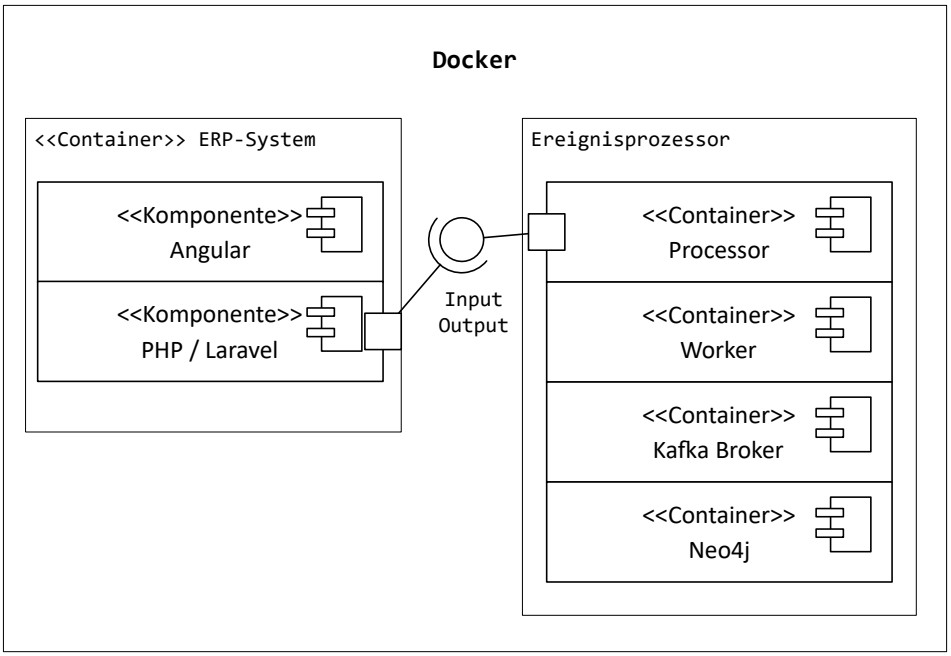


Abbildung 5.8: Deployment in Docker

den Empfang von Ereignissen von externen Systemen erlauben, und die Ausgabe des Prozessors empfangen. Konkret agiert das ERP-System damit als Prosumer des Ereignisprozessors. Hierdurch wird die Integration externer Dienste erleichtert, da die Umformung von Ereignissen, die nicht der vom Prozessor erwarteten Syntax entsprechen, ermöglicht wird. Zudem kann die bereits implementierte Authentifizierung und Autorisierung genutzt werden, um den Zugriff auf den Prozessor nur durch registrierte Applikationen zu erlauben.

6 Evaluierung

6.1 Testing

6.1.1 Unit-Tests & Mockito

Mit dem für Java gängigem Framework für Testing JUnit werden Unit-Tests sowohl in Koordinator- und Worker-Applikation implementiert. Die Testfälle werden so gewählt, dass die wichtigsten Komponenten der einzelnen Schichten getestet werden.

Abhängigkeiten zu externen Diensten wie zum Beispiel durch HTTP-Anfragen, Kafka und Neo4j werden durch das Mocking Framework Mockito abgefangen und getestet.

6.1.2 Manuelle und Explorative Tests

Unit-Tests werden durch exploratives Testen unterstützt. Mögliche (Falsch-)Eingaben eines Nutzers des Prozessors werden über Postman simuliert und auftretende Fehler behoben.

6.1.3 Continuous Integration / Continuous Delivery

Eine selbstgehostete GitLab-Instanz wird für die Versionierung des Programmcodes mittels Git genutzt. Darüber hinaus werden automatische Unit-Tests durch GitLab Runner [11] bei Erstellung von Git Tags ausgeführt. Für das Deployment werden Docker Images automatisch nach erfolgreichem Abschluss der Tests erzeugt, die anschließend in einer lokalen Docker Registry hochgeladen werden.

6.2 Evaluierung

Die implementierten Features des Ereignisprozessors werden anhand zwei realer Use-Cases getestet.

6.2.1 Automatisches Sammeln von Patchnotes in Git Commits

Die Anfertigung von Patchnotes, die für Endbenutzer als Changelog ausgeliefert werden, stellt für Entwickler meist einen hohen zusätzlichen Aufwand dar. Aus Commit-Nachrichten sollen über den Ereignisprozessor automatisch Einträge für einen Changelog gesammelt werden, der dann bei Deployment einer neuen Version automatisch veröffentlicht werden kann.

GitLab Webhooks

Die Ereignisse der GitLab-Instanz werden über Webhooks geerntet. Über die Admin-Konsole wird ein „System Hook“ angelegt, der bei Push-Events in beliebigen Repositories Ereignisse versendet. Für die Eingabe in den Prozessor müssen diese noch umgeformt werden, da sie noch nicht der erwarteten Syntax entsprechen. Hierfür wird der Webhook dazu konfiguriert, die Ereignisse zuerst an das Backend der ERP-Suite zu senden, wo sie angepasst und schließlich in den Prozessor als `GITLAB_PUSH` und `GITLAB_TAG_PUSH` gegeben werden. Als Payload wird die jeweilige Commit-Nachricht und -Hash, der Git Branch sowie der Name des zugehörigen Repositories angehangen.

Definieren der Verarbeitungsregel

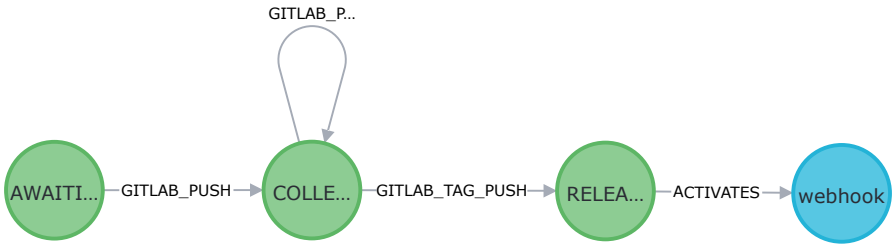


Abbildung 6.1: Verarbeitungsregel zum Sammeln von Patchnotes in Git Commits

Die zugehörige Verarbeitungsregel wird in drei Zustände geteilt. Bei einem eingehenden `GITLAB_PUSH`-Ereignis geht der Startzustand `AWAITING_PUSH` über in `COLLECTING_COMMITS`. Hier werden über eine zyklische Kante weiter `GITLAB_PUSH`-Ereignisse gesammelt. Wird eine neue Release-Version erzeugt, indem ein `GITLAB_TAG_PUSH` ausgelöst wird, erreicht die Regel den `RELEASE`-Zustand und sendet alle bisher gesammelten Commits eines Projektes per Webhook zurück an das ERP-Backend, wo sie entsprechend weiterverarbeitet werden können.

Patternmatching im Payload

```yaml
- on: GITLAB_PUSH
  with:
    - property: message
  source: AWAITING_PUSH
  target: COLLECTING_COMMITS
```

(a) ohne Mustererkennung

```yaml
- on: GITLAB_PUSH
  with:
    - property: message
      matching:
        ↪ ^.*\npatchnote:.*$
  source: AWAITING_PUSH
  target: COLLECTING_COMMITS
```

(b) mit Mustererkennung

Abbildung 6.2: Payload-Bedingung ohne / mit regulärem Ausdruck

Da nicht jeder Commit relevant ist, muss eine zusätzliche Bedingung für die Filterung von Ereignissen implementiert werden. Bisher konnte nur eine einfache Bedingung für die Prüfung auf die Existenz eines Payload-Wertes genutzt werden (Abbildung 6.2a). Diese Bedingung wird durch eine zusätzliche Einschränkung erweitert, die bei Bedarf angegeben werden kann und die Nutzdaten eines Ereignisses mit einem regulären Ausdruck abgleicht (Abbildung 6.2b).

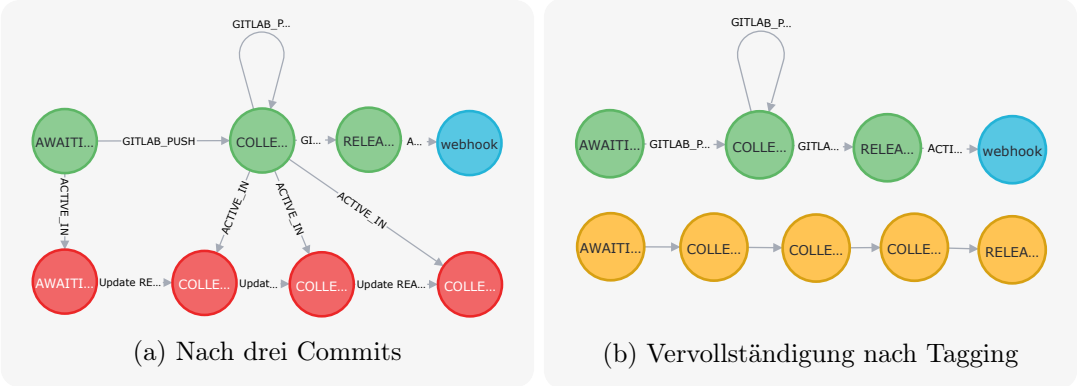
Aggregation über Ergebnispfaden

```
output:
  as: event
  named:
  ↪ RELEASE_NOTES_GATHERED
  with:
    - property: notes
      by: collection
      of: message
      from:
        - GITLAB_PUSH
    - property: project
      by: first
      from:
        - GITLAB_PUSH
```

Abbildung 6.3: Output-Format in Konfiguration

Wie im Entwurf 4.4.5 beschrieben, werden bisher gesamte Ereignisketten an Konsumenten gesendet. Die Konfiguration wird erweitert, um Aggregationen von Ereignisnutzdaten zu erlauben. Dafür kann ein „Output“-Format an einer Aktion definiert werden. Für den Aufbau des Changelogs wird nur der jeweilige Projektname und die Commit-Nachrichten benötigt. In Abbildung 6.3 wird die Konfiguration für die Ausgabe eines neuen Ereignisses `RELEASE_NOTES_GATHERED` mit zwei Eigenschaften `notes` als Sammlung aller Werte aus dem Feld `message` und `project` als erster Wert des gleichnamigen Feldes aller eingegangener `GITLAB_PUSH` Ereignisse gezeigt.

Ergebnis



(a) Nach drei Commits

(b) Vervollständigung nach Tagging

Abbildung 6.4: Sammlung von Commits in Verarbeitungsregel
(Grün: Zustand, Blau: Aktion, Rot: aktivierter Zustand, Gelb: Ergebnispfad)

In einem neuen Repository werden drei Test-Commits erzeugt, die das zuvor definierte Muster in der Commit-Nachricht enthalten. Zuletzt wird die Webhook-Aktion ausgelöst, indem ein Release-Tag erstellt wird. Zwischen einzelnen Schritten wird die Verarbeitungsregel in Neo4j betrachtet. In den Abbildungen 6.4a und 6.4b sind die Ergebnisse der Einzelschritte zu sehen. Der Log des ERP-Systems zeigt zudem, dass der Webhook mit dem nach der Output-Konfiguration geformten Ergebnis erfolgreich versandt wurde.

Die Konsolenausgabe des Worker-Prozesses zeigt, dass einkommende Ereignisse vom ersten Empfang bis zur Benachrichtigung des Koordinators ungefähr in 150 Millisekunden verarbeitet werden.

6.2.2 Erkennung von inaktiven Tickets in Helpdesk

Für die Kommunikation mit Kunden wird in der abraxas Verlag GmbH ein intern entwickelter Helpdesk verwendet. Eingereichte Tickets können jedoch leicht „verloren gehen“, da weder Kunden noch Mitarbeiter über inaktive Tickets informiert werden. Der Ereignisprozessor soll genutzt werden, um die Partie zu benachrichtigen, die „am Zug ist“, zu antworten. Hier soll das Complex Event Processing des Prozessors an einer komplexeren Kette von Ereignissen erprobt werden.

Definieren der Verarbeitungsregel

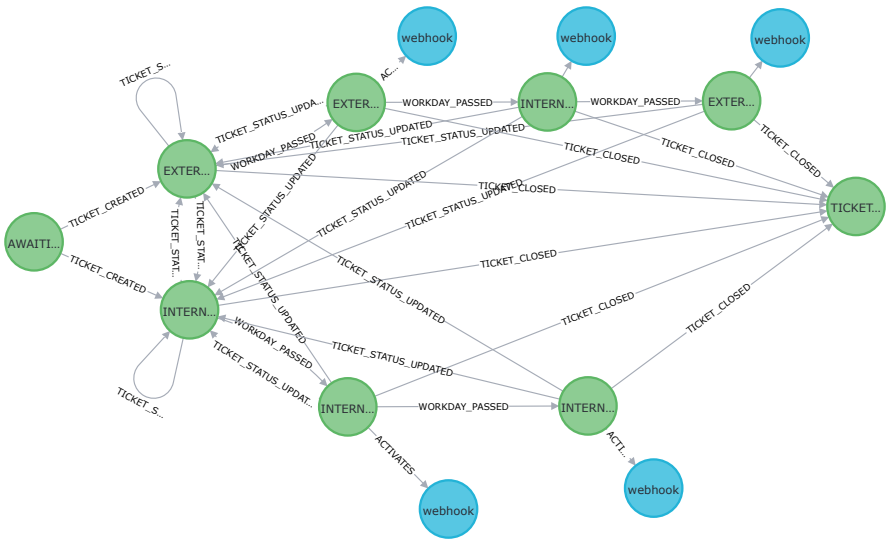


Abbildung 6.5: Verarbeitungsregel für Benachrichtigung von Nutzern des Helpdesks

Jedes Ticket im Helpdesk-System besitzt einen Status, der angibt, ob eine Antwort vom jeweiligen Kunden oder von einem Mitarbeiter erwartet wird. Hierfür werden zwei Zustände `EXTERNAL_PARTY_PENDING` und `INTERNAL_PARTY_PENDING` modelliert. Diese Zustände werden bei Erstellung des Tickets erreicht, oder wenn der Status des Tickets geändert wird.

Verweilt ein Ticket in einem Status, müssen in verschiedenen Zeitintervallen Benachrichtigungen versandt werden. Dafür werden die Zustände erweitert um `EXTERNAL_PARTY_NOTIFIED`, `INTERNAL_PARTY_NOTIFIED` et cetera. Benachrichtigungen werden über Webhooks an das ERP-System abgewickelt. Wird das Ticket geschlossen oder der Status

des Tickets verändert, werden alle Benachrichtigungen abgebrochen bzw. Zeitintervalle zurückgesetzt.

Produzenten für Ereignisse in Zeitintervallen

```yaml
processor:
  producers:
    - schedule: "0 0 4 * * 1-5"
      name: "Daily"
      type: simple
      event: WORKDAY_PASSED
```

Abbildung 6.6: SimpleProducer für Ablauf eines Tages innerhalb der Woche

Um Zeitintervalle zu erkennen, werden in der Konfiguration des Prozessors `SimpleProducer` wie beschrieben in 5.4.1 angelegt. Da weder Mitarbeiter noch Kunden aufgrund von Geschäftszeiten an Wochenenden erinnert werden sollen, wird ein Produzent für `WORKDAY_PASSED` angelegt, der nur an Arbeitstagen der Woche (montags bis freitags) auslöst.

Vereinfachung von Schreibweisen in Verarbeitungsregeln

```yaml
- on: WORKDAY_PASSED
  source: PENDING_00
  target: PENDING_01
# ...
- on: WORKDAY_PASSED
  source: PENDING_06
  target: NOTIFIED
```

(a) alte Syntax

```yaml
- on: WORKDAY_PASSED
  times: 7
  source: PENDING
  target: NOTIFIED
```

(b) neue Syntax

Abbildung 6.7: Implementierung von Zählbedingung an Kante

Für Zeitintervalle mussten bisher „Hilfszustände“ definiert werden, die die Übersicht in der Konfigurationsdatei deutlich reduziert haben (Abbildung 6.8a). Es wird eine neue vereinfachte Schreibweise implementiert, die das Zählen von Ereignissen erlaubt (Abbildung 6.8b).

```yaml
- on: STATUS_UPDATED
  source: PENDING
  target: PENDING

- on: STATUS_UPDATED
  source: NOTIFIED
  target: PENDING

- on: STATUS_UPDATED
  source: REMINDED
  target: PENDING
```

(a) alte Syntax

```yaml
- on: STATUS_UPDATED
  source:
    - PENDING
    - NOTIFIED
    - REMINDED
  target: PENDING
```

(b) neue Syntax

Abbildung 6.8: Implementierung für Angabe mehrerer Quellen für Kanten

Viele gleichförmige Kanten mit dem selben Ziel haben in der Konfigurationsdatei viele Redundanzen erzeugt. Kantendefinitionen werden erweitert, sodass mehrere Quellen definiert werden können.

Ergebnis

Auch komplexere Graphen werden durch den Prozessor korrekt traversiert. Durch die Komplexität wird jedoch das Auslesen des Prozessors erschwert, da der Kontext schwerer nachzuvollziehen ist; mittelfristig muss hier ein besseres Interface für die Einsicht auf Verarbeitungsregeln und dem aktuellen Verarbeitungskontext geschaffen werden.

7 Fazit

Nach Implementierungen und Evaluierung werden Projekt und Bachelorarbeit rückblickend betrachtet.

7.1 Retrospektive

Es konnte ein Ereignisprozessor basierend auf Graphen implementiert werden, der durch verhältnismäßig leichter Konfiguration komplexe Zusammenhänge zwischen Ereignissen aggregieren und auswerten kann. Abbildungen von Regeln als deterministische endliche Automaten halfen, Ereignisse als reguläre Sprache zu betrachten und Beziehungen zwischen Ereignissen für die Verarbeitung zu nutzen.

Über Apache Kafka konnte ein Ereignisbus mit Arbeiterprozessen implementiert werden, durch welcher die Verarbeitung zukunftssicher und skalierbar umgesetzt werden konnte. Ausstehend bleiben Analysen zum Durchsatz und Performanz des Prozessors, welche abhängig von der Anzahl und Komplexität der definierten Verarbeitungsregeln stark schwanken könnten.

Ziel der Arbeit war die Unterstützung der Mitarbeiter der abraxas Verlag GmbH bei der täglichen Arbeit. Während der Entwicklung des Prozessors ist allerdings klar geworden, dass für ein solches Tooling bisherige Arbeitsabläufe zuerst genauer untersucht und verstanden werden müssen, um die Ereignisse täglicher Geschäftsprozesse in Muster zu „gießen“: zwar ist das Ziel bekannt – welche Verarbeitungsregeln und welche Muster an Ereignissen jedoch wirklich für den täglichen Gebrauch relevant sind und Mitarbeiter aktiv unterstützen, muss zunächst noch näher erforscht werden. So konnte auch – neben zeitlicher Begrenzung – keine Client-Anwendung für Mitarbeiter implementiert werden.

7.2 Ausblick

Der Ereignisprozessor zeigt das Potenzial, Geschäftsprozesse aller Art durch Automatisierungen und Auswertungen zu unterstützen. Zukünftig könnte eine Client-Anwendung erlauben, dass Mitarbeiter Verarbeitungsregeln direkt selbst definieren können. Der relativ leichte Aufbau der Konfiguration würde es ermöglichen, eigens entdeckte Muster in produzierten Ereignissen bei erneutem Auftreten in verwertbare Datensätze zu wandeln. So könnten Formulierungen wie: „Immer dann, wenn ich ein Telefonat starte, und dann ein Helpdesk-Ticket öffne, verknüpfe das Telefonat mit diesem Ticket“ ohne Programmieraufwand als eine Regeldefinition dienen.

Zudem sollen Mitarbeiter durch eine Smartphone-App bei der Erfassung von Arbeitszeiten bei Kundenbesuchen unterstützt werden. Durch einen Knopfdruck – oder sogar automatisch durch Geofencing bei Nähe zu einem Kunden – könnte eine Zeiterfassung gestartet werden, die alle Ereignisse des Mitarbeiters während des Besuches aufzeichnet und zu konkreten, berechenbaren Arbeitszeiten umwandelt; zum Beispiel das Erfassen neuer Hardware nach Auslieferung, oder die Aktualisierung von Helpdesk-Tickets.

Literaturverzeichnis

[1] 3CX: *Business Communication Solutions & Software*. Accessed August 2021. – URL https://www.3cx.com

[2] Agendaless Consulting: *Supervisor: A Process Control System*. Accessed August 2021. – URL http://supervisord.org/

[3] Balzert, Helmut: *Anforderungen und Anforderungsarten*. S. 455–474. In: *Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering*. Heidelberg : Spektrum Akademischer Verlag, 2009. – URL https://doi.org/10.1007/978-3-8274-2247-7_16. – ISBN 978-3-8274-2247-7

[4] Bidgoli, Hossein ; Vehovar, V.: *Enterprise Resource Planning (ERP)*. S. 707. In: *The Internet Encyclopedia*, John Wiley & Sons, 2004. – URL https://books.google.com/books?id=ACfBmYiNaTcC. – ISBN 978-0-471-22202-6

[5] cURL: *command line tool and library*. Accessed August 2021. – URL https://curl.se/

[6] Docker, Inc.: *Docker: Empowering App Development for Developers*. Accessed August 2021. – URL https://www.docker.com/

[7] Docker Inc.: *Dockerhub: neo4j*. Accessed August 2021. – URL https://hub.docker.com/_/neo4j

[8] embarcadero Inc.: *Delphi 10.4.2: Build Native Apps 5x Faster For Windows, Android, iOS, macOS, and Linux*. Accessed August 2021. – URL https://www.embarcadero.com/products/delphi

[9] FasterXML, LLC.: *The Jackson Project*. Accessed August 2021. – URL https://github.com/FasterXML/jackson

[10] GitLab Inc.: *GitLab*. Accessed August 2021. – URL https://gitlab.com/

[11] GitLab Inc.: *GitLab Runner*. Accessed August 2021. – URL https://docs.gitlab.com/runner/

[12] Google LLC: *Angular Full Color Logo*. Accessed August 2021. – URL https://angular.io/presskit. – Provided under CC BY 4.0

[13] Google LLC: *Angular: The Modern Web Developer's Platform*. Accessed August 2021. – URL https://angular.io/

[14] Hedtstück, Ulrich: *Complex Event Processing Engines*. S. 85–97. In: *Complex Event Processing: Verarbeitung von Ereignismustern in Datenströmen*, Springer Vieweg, Berlin, Heidelberg, 2020. – URL https://www.springer.com/de/book/9783662615751. – ISBN 978-3-662-61575-1

[15] Hedtstück, Ulrich: *Datenanalyse mit Complex Event Processing*. S. 12. In: *Complex Event Processing: Verarbeitung von Ereignismustern in Datenströmen*, Springer Vieweg, Berlin, Heidelberg, 2020. – URL https://www.springer.com/de/book/9783662615751. – ISBN 978-3-662-61575-1

[16] Heinz, Christoph ; Greiner, Torsten: Business Activity Monitoring mit Stream Mining am Fallbeispiel TeamBank AG. In: *HMD Praxis der Wirtschaftsinformatik* 46 (2009), Nr. 4, S. 82–89. – URL https://doi.org/10.1007/BF03340383

[17] Hopcroft, John E. ; Motwani, Rajeev ; Ullman, Jeffrey D.: *Endliche Automaten und reguläre Ausdrücke*. S. 122. In: *Einführung in Automatentheorie, Formale Sprachen und Berechenbarkeit* , Pearson Deutschland, 2011. – URL https://elibrary.pearson.de/book/99.150005/9783863265090. – ISBN 9783868940824

[18] Laravel LLC: *Laravel: The PHP Framework For Web Artisans*. Accessed August 2021. – URL https://laravel.com/

[19] Luckham, David: *Glossary of Terminology: The Event Processing Technical Society: (EPTS) Glossary of Terms—Version 2.0*. S. 237–258. In: *Event Processing for Business: Organizing the Real-Time Enterprise*, John Wiley & Sons, Ltd, 2012. – URL https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119198697.app1. – ISBN 978-1-119-19869-7

[20] Luckham, David C.: *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing

Co., Inc., 2001. – URL https://dl.acm.org/doi/book/10.5555/515781. – ISBN 0201727897

[21] Microsoft Corporation: *SQL Server 2019*. Accessed August 2021. – URL https://www.microsoft.com/en-us/sql-server/sql-server-2019

[22] Neo4j, Inc.: *Neo4j Graph Database*. Accessed August 2021. – URL https://neo4j.com/product/neo4j-graph-database/

[23] Neo4j, Inc.: *Spring Data Neo4j*. Accessed August 2021. – URL https://neo4j.com/developer/spring-data-neo4j/

[24] Postman, Inc.: *Postman API Platform*. Accessed August 2021. – URL https://www.postman.com/

[25] TeamViewer AG: *TeamViewer: The Remote Desktop Software*. Accessed August 2021. – URL https://www.teamviewer.com

[26] The Apache Software Foundation: *Apache Flink*. Accessed August 2021. – URL https://flink.apache.org/

[27] The Apache Software Foundation: *Apache Maven*. Accessed August 2021. – URL https://maven.apache.org/

[28] The Apache Software Foundation: *Apache Spark*. Accessed August 2021. – URL http://spark.apache.org/

[29] TrustRadius: *Apache Spark Reviews*. Accessed August 2021. – URL https://www.trustradius.com/products/apache-spark/reviews

[30] VMware, Inc.: *Spring Boot Reference Documentation: Starters*. Accessed August 2021. – URL https://docs.spring.io/spring-boot/docs/2.5.0/reference/htmlsingle/#using.build-systems.starters

[31] VMware, Inc.: *Spring for Apache Kafka*. Accessed August 2021. – URL https://spring.io/projects/spring-kafka

[32] VMware, Inc.: *Spring: Spring makes Java simple*. Accessed August 2021. – URL https://spring.io/

[33] Wikimedia Commons: *Laravel.svg - Laravel logo*. Accessed August 2021. – URL https://commons.wikimedia.org/wiki/File:Laravel.svg. – Logo in the public domain

[34] Wikipedia: *Software as a Service*. Accessed August 2021. – URL https://de.wikipedia.org/wiki/Software_as_a_Service

A User Stories

Für die Entwicklung eines Lasten- und Pflichtenheftes wurden zu Beginn der Entwurfsphase User Stories erfasst, die Anforderungen an die Funktionalität des Ereignisprozessors beschreiben. Hier sind die gesammelten User Stories in keiner spezifischen Reihenfolge abgebildet.

- Der Benutzer kann Verarbeitungsregeln anlegen
- Der Benutzer kann den aktuellen Zustand von Regeln aus dem Prozessor auslesen
- Das System sendet aktiv Ergebnisse an andere Systeme
- Das System veröffentlicht passiv Ergebnisse für externen Zugriff

B Beispiel für eine Verarbeitungsregel in YAML

```yaml
# Definiere den eindeutigen Namen einer Verarbeitungsregel
name: Record calls

# Definiere die Zustände des Automaten.
# Jeder Zustand hat einen eindeutigen Namen,
# und spezifiziert Aktionen, die beim aktivieren
# des Zustands ausgelöst werden.
states:
  - name: AWAITING_CALL
    initial: true
  - name: IN_CALL
  - name: CALL_ENDED
      actions:
        - type: webhook
          url: "<External system URI>"
          headers:
            - Authorization: "Bearer <Token>"
```

```yaml
# Definiere die Kanten zwischen den Zuständen.
# Kanten werden beim Verarbeiten von Ereignissen
# traversiert, wenn sie den angegebenen Bedingungen
# entsprechen.
transitions:
  # "IN_CALL" wird genau dann aktiviert, wenn
  # ein Ereignis vom Typ "CALL_BEGIN" mit den
  # Payload-Werten "source" und "target"
  # verarbeitet wird.
  - on: CALL_BEGIN
    with:
      - property: source
      - property: target
    source: AWAITING_CALL
    target: IN_CALL

  # "CALL_ENDED" kann nur innerhalb eines 24h-Zeitfenser
  # aktiviert werden, und die Payload-Daten müssen mit
  # dem zuvor verarbeiteten Eigenschaften des "CALL_BEGIN"-
  # Ereignisses übereinstimmen.
  - on: CALL_END
    with:
      - property: source
      - property: target
      - same: source
        as: CALL_BEGIN
      - same: target
        as: CALL_BEGIN
    window: PT24H # ISO-8601
    source: IN_CALL
    target: CALL_ENDED
```

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original