

Bachelor-Thesis

zur Erlangung des akademischen Grades *Bachelor of Science* (B.Sc.) im
Studiengang Angewandte Informatik

Konzeptionierung und Implementierung eines Alarmierungssystems mit einer Message Oriented Middleware

Erstprüfer:	Prof.Dr.-Ing. Silvia Keller Hochschule Ravensburg-Weingarten
Zweitprüfer:	Dipl.-Inf. (FH) Daniel Bonneval Engelbart Software GmbH
Autor:	Philipp Bobek MatNr. 23765
Abgabetermin:	30. November 2016

Ehrenwörtliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit mit dem Titel **Konzeptionierung und Implementierung eines Alarmierungssystems mit einer Message Oriented Middleware** selbständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt, keine anderen als die angegebenen Hilfsmittel benutzt und wörtliche sowie sinngemäße Zitate als solche gekennzeichnet habe.

Ort, Datum :

Unterschrift :

Danksagung

An dieser Stelle möchte ich mich bei der Engelbart Software GmbH bedanken, in deren Auftrag ich diese Arbeit geschrieben habe. Ich bedanke mich für die Zeit in der Firma, seitdem ich dort 2014 mein Praxissemester absolvieren durfte. Mein besonderer Dank gilt meinem Betreuer Dipl.-Inf. (FH) Daniel Bonneval, mit dem ich zusammen das Thema dieser Arbeit ausgearbeitet habe und der mich während der Umsetzung unterstützt hat.

Des Weiteren gilt mein Dank meiner Betreuerin an der Hochschule, Frau Prof.Dr.-Ing. Silvia Keller.

Nicht zuletzt möchte ich mich bei meinen Eltern bedanken, die mich während meines Studiums finanziell unterstützt haben und auf deren Unterstützung ich mich in jeder Lebenslage verlassen kann.

Abstract

Diese Arbeit beschäftigt sich mit der Entwicklung eines prototypischen Alarmierungssystems. Die einzelnen Komponenten des Systems basieren auf dem Spring Framework.

Es wurde untersucht, wie eine Message Oriented Middleware zur Verteilung von Alar-men an zuständige Personen genutzt werden kann.

In der Analyse wurden Schnittstellen und Protokolle, Broker Implementierungen und Spring Projekte untersucht und verglichen. Daraufhin wurde eine Auswahl an Techno-logien getroffen, die für den Prototypen verwendet werden sollen.

Mit dieser Auswahl wurden mögliche Konzepte ausgearbeitet, die eine Verteilung der Alarme ermöglichen. Der letzte Schritt bestand darin, mit den gewonnenen Erkennt-nissen einen Prototypen zu implementieren.

Inhaltsverzeichnis

1	Einleitung	1
1.1	Motivation	1
1.2	Zielsetzung	1
1.3	Aufbau der Arbeit	2
2	Grundlagen	3
2.1	Spring Framework	3
2.2	Message Oriented Middleware	4
2.3	Schnittstellen und Protokolle	4
2.3.1	Java Message Service	5
2.3.2	Advanced Message Queuing Protocol	6
2.3.3	Apache Kafka	8
2.3.4	Streaming Text Orientated Messaging Protocol	10
2.3.5	Message Queue Telemetry Transport	10
3	Analyse	11
3.1	Schnittstellen und Protokolle	11
3.1.1	Vergleich der Schnittstellen und Protokolle	11
3.1.2	Begründung der Wahl von AMQP und STOMP	12
3.2	Broker	13
3.2.1	Vergleich der Broker Implementierungen	13
3.2.2	Begründung der Wahl von RabbitMQ	16
3.3	Spring Projekte	16
3.3.1	Spring AMQP	17
3.3.2	Spring Integration	17
3.3.3	Spring Cloud Stream	18
3.3.4	Spring WebSocket	19
3.3.5	Einsatzgebiete der Spring Projekte	21
4	Konzepte	22
4.1	Dezentrale Alarmierung	22
4.1.1	Exchange to Exchange Bindings	22
4.1.2	Headers Exchange	23
4.2	Zentrale Alarmierung	24
4.2.1	Sender-selected Distribution	24
4.2.2	Spring Websocket	25
5	Prototyp	26
5.1	Architektur	26
5.2	Projektstruktur	28
5.3	Datenmodell	28
5.4	REST Gateway	30
5.5	Alarmierungsapplikation	32
5.6	User Applikation	35
5.7	Authentifizierungs Applikation	36
5.8	Fehlende Funktionalitäten	37

6 Zusammenfassung und Ausblick	38
Abbildungsverzeichnis	39
Tabellenverzeichnis	40
Listingverzeichnis	41
Abkürzungsverzeichnis	42
Literaturverzeichnis	43
Anhang	45

1 Einleitung

1.1 Motivation

Elektronische Systeme überwachen heutzutage immer mehr Bereiche unseres Lebens. Ob einen Patienten im Krankenhaus, dessen Gesundheitszustand überwacht wird, oder eine Werkzeugmaschine in einer Werkhalle, deren Funktionsfähigkeit überwacht wird. Diese elektronischen Systeme können beim Auftreten bestimmter Ereignisse einen Alarm erzeugen. Dieser wird von einem Alarmierungssystem verarbeitet. Es wird erwartet, dass der Alarm entgegen genommen wird, dass zuständige Personen alarmiert werden und dass der Fortschritt der Bearbeitung des Alarms verfolgt wird.

Alarme werden je nach System über unterschiedliche Protokolle und Formate übertragen. Sie könnten beispielsweise über einen REST Call an ein Alarmierungssystem übertragen werden. Es kommen jedoch auch diverse andere Schnittstellen zur Übertragung in Frage, wie vorhandene Alarmierungssysteme zeigen [Der16].

Für das Beispiel des Patienten in einem Krankenhaus könnte bei der Alarmübertragung der Gesundheitszustand und die Patientenummer übertragen werden, bei einer Werkzeugmaschine die Fehlernummer und die Kennung der betroffenen Maschine. Die Alarmverarbeitung muss daher an die verschiedenen Systeme angepasst werden können.

Essentiell für die Behebung der Alarmursache ist, dass die zuständigen Personen alarmiert werden und alle benötigten Informationen erhalten. Dabei stellt sich insbesondere die Frage, wie die Alarme an zuständige Personen verteilt werden sollen. An dieser Stelle könnte eine MOM ansetzen. Sie bietet Funktionalitäten, um Nachrichten in verteilten Systemen auszutauschen und damit auch an User weiterzuleiten.

1.2 Zielsetzung

In dieser Arbeit soll ein System zur Alarmierung konzipiert und implementiert werden. Die Weiterleitung von Alarmen an zuständige Personen oder weitere Systeme wird über eine MOM abgewickelt. Der Fokus der Arbeit liegt auf der Analyse, wie die MOM in dem System verwendet werden kann. Folgende Funktionalitäten werden gefordert:

- Die verwendete Software muss Open Source sein.
- Die einzelnen Komponenten des Systems werden mit dem Spring Framework implementiert.
- Über einen REST Call sollen neue Alarme erstellt werden können. Dabei muss das System leicht erweiterbar bleiben, sodass sich einfach weitere Quellen für Alarme hinzufügen lassen.

- Alarme werden über eine MOM an User oder weitere Systeme verteilt.
- User können über eine Webanwendung alarmiert werden. Anschließend können die User das Bearbeiten des Alarms annehmen oder ablehnen. Weitere User Anwendungen, wie eine mobile Applikation, können hinzugefügt werden.
- Über eine Autorisierung wird gesteuert, welche Alarme ein User empfängt.

Diese Arbeit beschäftigt sich nicht mit der Implementierung einer spezifischen Geschäftslogik für Eskalation oder Quittierung der Alarme.

1.3 Aufbau der Arbeit

In dem Kapitel Grundlagen wird das für den Prototyp verwendete Spring Framework vorgestellt. Anschließend wird geklärt, worum es bei einer MOM und Message Brokern handelt, auf denen der Schwerpunkt der Arbeit beruht. Im Zuge dessen werden die hier wichtigen Schnittstellen und Protokolle vorgestellt, über die mit einer MOM kommuniziert werden kann.

In der Analyse werden Schnittstellen und Protokolle, Broker Implementierungen und Projekte des Spring Frameworks verglichen. Es werden die Technologien ausgewählt, die für den Prototypen verwendet werden sollen.

Aus den dort gewonnenen Erkenntnissen werden mögliche Konzepte für das Alarmieren von Usern entwickelt. Letztendlich wird der entwickelte Prototyp vorgestellt, der auf den entwickelten Konzepten basiert.

2 Grundlagen

2.1 Spring Framework

Das Spring Framework bietet eine ganzheitliche Lösung für das Erstellen von Java und insbesondere Java EE Applikationen. Das Herzstück des Frameworks bildet ein Core Modul. Dieses Modul stellt zentrale Funktionalitäten des Frameworks wie einen *Inversion of Control* Container bereit, ohne die das Framework nicht funktionsfähig wäre. Das Framework kann durch weitere optionale Projekte ergänzt werden. Erwähnenswert für diese Arbeit sind das Spring Boot, Spring Integration, Spring Cloud und das Spring Security Projekt.

Das Spring Boot Projekt verringert den Aufwand beim Programmieren auf ein Minimum. Dies wird vor allem durch die Umsetzung des Softwaredesign-Paradigmas *Convention over Configuration* [Che06] ermöglicht. Spring wird demnach an allen möglichen Stellen vorkonfiguriert und eine manuelle Konfiguration wird lediglich benötigt, wenn von der Konvention abgewichen wird. „[Spring Boot takes] an opinionated view of building production-ready Spring applications. Spring Boot favors convention over configuration and is designed to get you up and running as quickly as possible.“, so die eigene Beschreibung des Projekts [Piv16c].

Das Spring Integration Projekt erweitert das Framework um Implementierungen von *Enterprise Integration Patterns* wie sie in [HW03] definiert sind. Diese beschreiben Entwurfsmuster für die Architektur von verteilten Systemen. Die Kommunikation zwischen den Systemen wird somit vereinheitlicht. Das Projekt ermöglicht die Verwendung einer Vielzahl von Schnittstellen und Protokollen.

Das Spring Cloud Projekt stellt Funktionalitäten für das Erstellen von verteilten System zur Verfügung. „Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems (e.g. configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus, one-time tokens, global locks, leadership election, distributed sessions, cluster state).“, so eine Aufzählung der Funktionalitäten [Piv16d].

Spring Security bietet Lösungen für die Authentifizierung und Autorisierung der Benutzer. „Spring Security is a powerful and highly customizable authentication and access-control framework. It is the de-facto standard for securing Spring-based applications.“, wie es kurz beschrieben wird [Piv16f].

2.2 Message Oriented Middleware

Eine MOM ist eine Komponente, über die verteilte Systeme Nachrichten austauschen können. „MOM is one of the cornerstone foundations that distributed enterprise systems are built upon.“ schreibt [Cur04, S. 26], was ihre Bedeutung veranschaulicht. Der Austausch läuft dabei über einen sogenannten Broker ab. Der Einsatz einer MOM bietet eine Vielzahl an Vorteilen gegenüber der direkten Kommunikation zwischen verteilten Systemen. Für diese Arbeit sind folgende Vorteile die wichtigsten:

- Nachrichten können asynchron ausgetauscht werden [Cur04, S. 4].
- Die MOM kann als Puffer zwischen Sender und Empfänger agieren. Wenn der Empfänger nicht verfügbar ist bleiben die Nachrichten in einer Queue zwischengespeichert und können später zugestellt werden [Cur04, S. 12].
- Viele Broker bieten ein Routing an, sodass Nachrichten anhand einer vordefinierten Logik verteilt werden [Cur04, S. 11].
- Durch den Einsatz einer MOM entsteht eine lose Kopplung zwischen den Systemen. Das macht die Systeme flexibler und änderungsfreudiger [Cur04, S. 5].

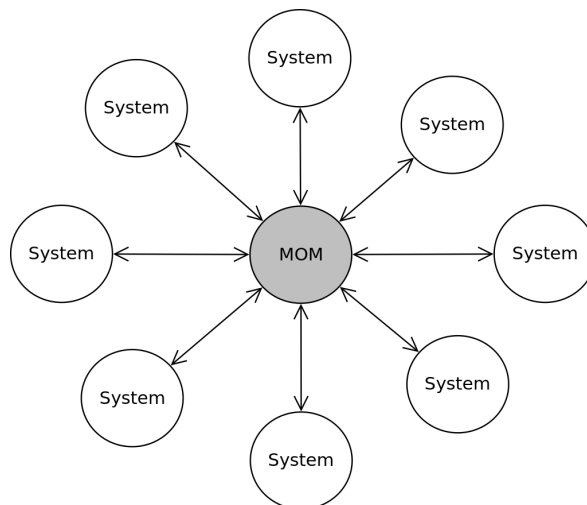


Abbildung 1: Message Oriented Middleware

2.3 Schnittstellen und Protokolle

Es gibt eine Vielzahl von Schnittstellen und Protokollen um mit den Brokern zu kommunizieren. Im Folgenden werden die Wichtigsten davon vorgestellt.

2.3.1 Java Message Service

JMS ist eine API um Nachrichten zwischen Systemen auszutauschen. Es ist Teil von Java EE. Es gibt zwei verschiedene Modelle wie Nachrichten gesendet und empfangen werden können. Diese sind in [JCNE⁺13] beschrieben. Zum einen gibt es das PTP Modell. Dabei werden Nachrichten von Publishern in eine Queue geschrieben. Daraufhin kann die Nachricht von genau einem Consumer empfangen werden. Deshalb ist das Modell auch unter „*Once and only once messaging*“ bekannt [Cur04, S. 12].

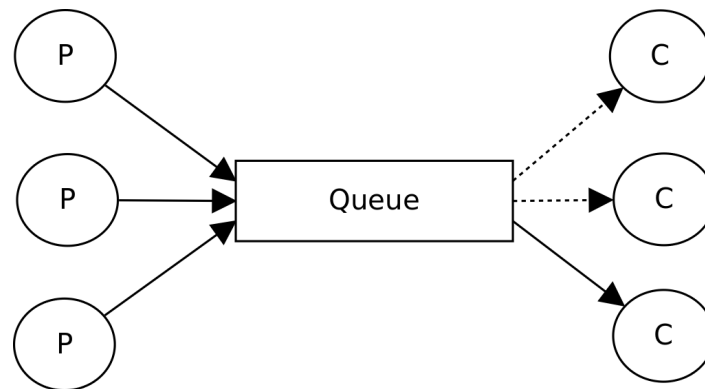


Abbildung 2: JMS Queue

Eine weitere Möglichkeit ist das pub/sub Modell. Dabei werden Nachrichten von Publishern an sogenannte *Topics* gesendet. Consumer können sich an diesem Topic anmelden und erhalten eine Kopie jeder Nachricht.

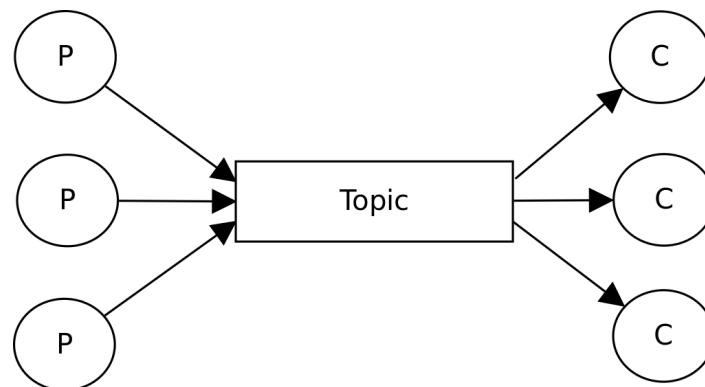


Abbildung 3: JMS Topic

Da es sich bei JMS um eine API und nicht um ein Protokoll handelt, muss beachtet werden, dass es viele verschiedene Implementierungen gibt. Um JMS in Verbindung mit einem bestimmten Broker zu verwenden, wird daher eine spezifische Implementierung benötigt. Momentan ist JMS vor allem in den Versionen 1.1 und 2.0 in Verwendung. In beiden Versionen werden die oben beschriebenen Modelle verwendet.

2.3.2 Advanced Message Queuing Protocol

AMQP ist im Gegensatz zu JMS keine API, sondern ein Protokoll. Es wird in einem offenen Standard beschrieben. Alle Funktionen von JMS wurden in diesem aufgenommen. Da es sich um ein offenes Protokoll handelt, gibt es viele verschiedene Broker und Client Implementierungen. Allerdings muss beachtet werden, welche Version des Protokolls verwendet wird. Momentan wird vor allem die Version 0-9-1 und 1.0 verwendet. Diese unterscheiden sich stark voneinander und sind untereinander nicht immer kompatibel. Die Version 1.0 ist im Gegensatz zu den anderen Versionen ein ISO und OASIS Standard. Der Unterschied wird wie folgt beschrieben: „Despite the name, AMQP 1.0 is a radically different protocol from AMQP 0-9-1 / 0-9 / 0-8, sharing essentially nothing at the wire level. AMQP 1.0 imposes far fewer semantic requirements; it is therefore easier to add support for AMQP 1.0 to existing brokers. The protocol is substantially more complex than AMQP 0-9-1, and there are fewer client implementations.“[Piv07b]. Bei der Version 1.0 wird versucht, den Brokern kein Modell vorzugeben, das sie unterstützen müssen. Es wird lediglich ein Protokoll beschrieben, um Daten mit einem Broker auszutauschen. AMQP 0-9-1 dagegen gibt Brokern ein gewisses Modell vor. Diese Grundlagen beziehen sich daher lediglich auf AMQP 0-9-1.

Beim AMQP Protokoll schreiben Publisher die Nachrichten nicht direkt in eine Queue wie bei der JMS API. Stattdessen gibt es sogenannte *Exchanges*. Diese werden über *Bindings* mit Queues verbunden. An Queues melden sich die Consumer an, um Nachrichten zu empfangen. Zusammengefasst wird eine Nachricht von einem Publisher in ein Exchange geschrieben, das Exchange verteilt die Nachricht über die Bindings an Queues und dort angemeldete Clients erhalten die Nachricht. Somit wird ein pub/sub Modell erzeugt, zu sehen in Abbildung 4.

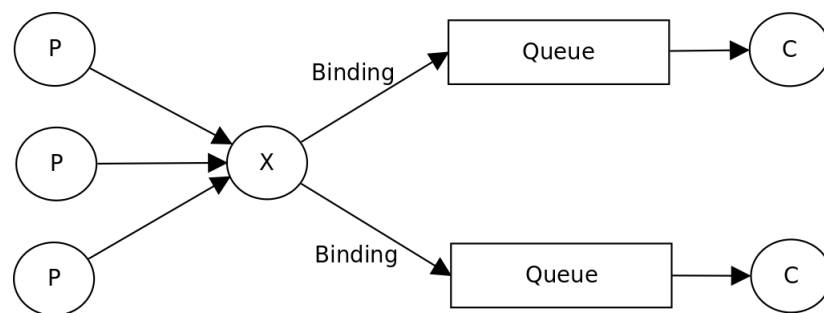


Abbildung 4: AMQP pub/sub

Das bekannte PTP Modell kann ebenfalls erzeugt werden. Denn wenn mehr als ein Consumer an einer Queue angemeldet ist, werden die Nachrichten in einem Round-Robin Verfahren an die Consumer verteilt, zu sehen in Abbildung 5. Wie man sieht verhält sich eine Queue, wenn mehrere Consumer angemeldet sind, gleich wie eine JMS Queue. Das JMS Modell von Queues und Topics, lässt sich demzufolge mit den AMQP Exchanges, Bindings und Queues emulieren.

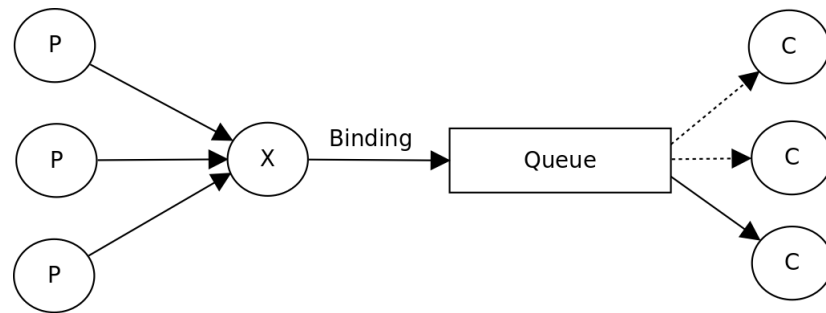


Abbildung 5: AMQP PTP

Was das AMQP Protokoll auszeichnet, sind die verschiedenen Möglichkeiten ein Routing für die Nachrichten festzulegen. Einer Nachricht kann ein sogenannter *Routing Key* und mehrere Argumente zugewiesen werden. Bei einem Binding können ebenfalls ein Routing Key und Argumente angegeben werden. Anhand dieser Werte kann ein Exchange bestimmen, über welche Bindings die Nachricht ausgeliefert werden muss. Es gibt vier Typen von Exchanges, die verschiedene Routing Möglichkeiten bereitstellen:

- Fanout
- Direct
- Topic
- Headers

Beim Fanout Exchange wird eine Nachricht ohne Rücksicht auf Routing Key oder Header Werte, über alle Bindings ausgeliefert. Dieses Verhalten ist in Abbildung 6 dargestellt. Es gibt immer ein vordefiniertes Exchange dieses Typs, mit dem Namen `amqp.fanout`.

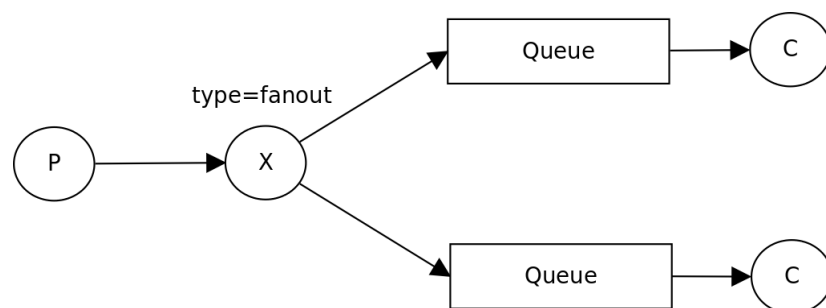


Abbildung 6: AMQP Fanout Exchange

Beim Direct Exchange wird die Nachricht ausgeliefert, wenn der Routing Key der Nachricht und der Routing Key des Bindings gleich sind. Andere Argumente werden ignoriert, wie in Abbildung 7 zu sehen ist. Besonders ist, dass es zwei vordefinierte Exchanges gibt. Eins mit dem Namen `amqp.direct`. Ein anderes hat einen **leeren String** als Namen. Jede erstellte Queue wird automatisch über ein Binding mit diesem verbunden. Dabei wird der Name der Queue als Routing Key verwendet. Wird beim

Versenden einer Nachricht kein Exchange angegeben, daher ein leerer String, und wird als Routing Key der Name einer Queue gewählt, wird die Nachricht automatisch in diese Queue geroutet. Daher wird dieses Exchange auch *Default Exchange* genannt. Dieser Mechanismus lässt es erscheinen, als ob ein Publisher Nachrichten direkt in eine Queue schreiben kann.

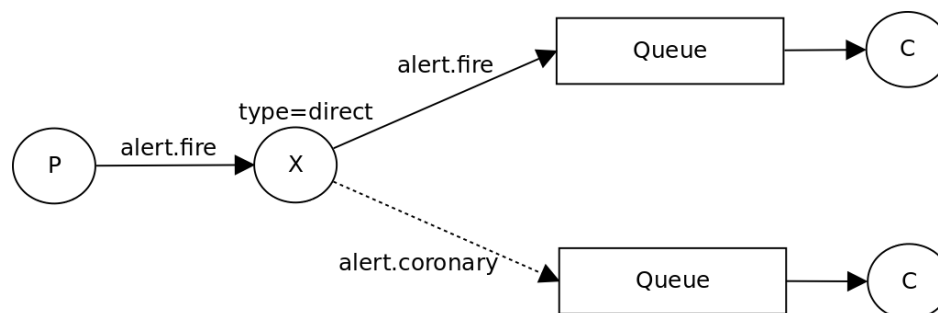


Abbildung 7: AMQP Direct Exchange

Das Topic Exchange interpretiert den Routing Key eines Bindings als *Pattern*. Ein '*' Zeichen im Pattern kann ein Wort ersetzen und ein '#' kann mehrere Worte ersetzen. Wenn der Routing Key der Nachricht zum Pattern passt, wird die Nachricht ausgeliefert. Beispielsweise passt der Routing Key 'event.alert.fire.high' zum Pattern '*.alert.#', wie in Abbildung 8 zu sehen. Die Header Werte werden hier ebenfalls ignoriert. `amqp.topic` ist der Name eines vordefinierten Exchange dieses Typs.

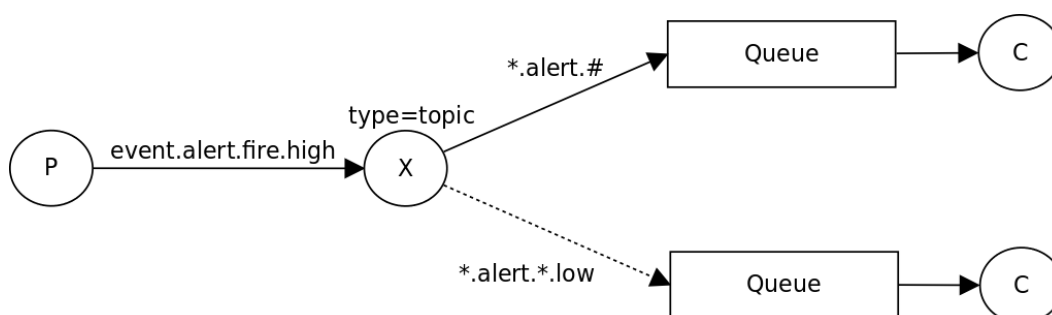


Abbildung 8: AMQP Topic Exchange

Beim Headers Exchange wird der Routing Key ignoriert. Dafür werden dort die Werte aus dem Header Argument verglichen. Außerdem wird definiert, ob für das Ausliefern der Nachricht lediglich einer der Werte oder alle Werte identisch sein müssen. Dafür kann das Argument `x-match` auf 'any' oder 'all' gesetzt werden. Zu sehen in Abbildung 9. Auch hier gibt es ein vordefiniertes Exchange, mit dem Namen `amq.match`.

2.3.3 Apache Kafka

Apache Kafka ist ein Message Broker mit den Schwerpunkten auf einen hohen Durchsatz, guter Skalierbarkeit und hoher Fehlertoleranz. Er verwendet ein eigenes Protokoll

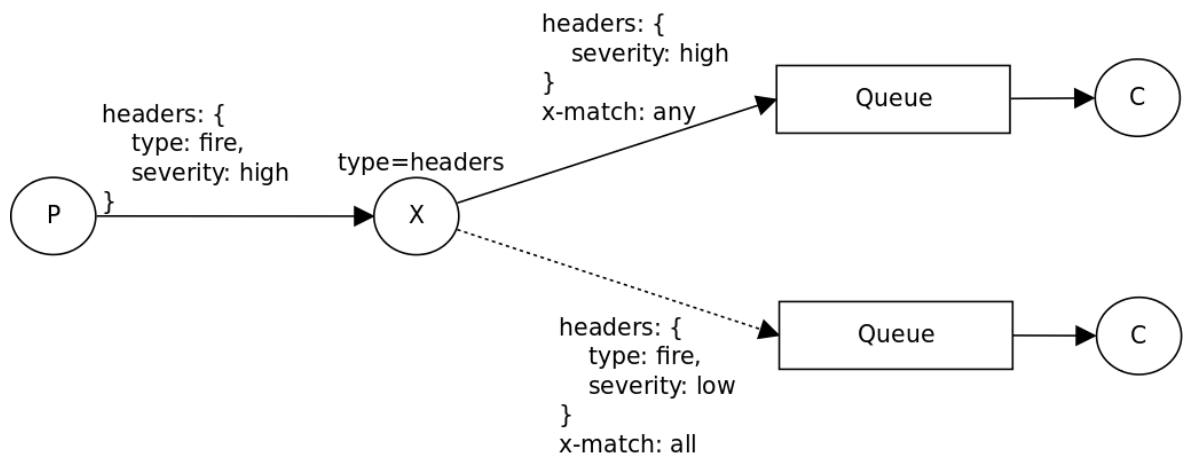


Abbildung 9: AMQP Headers Exchange

zur Kommunikation mit den Publishern und Consumern. Es wird lediglich ein einfaches pub/sub und PTP Modell, ohne Routing Möglichkeiten, angeboten. Publisher schicken ihre Nachrichten an ein Topic, von dort werden sie an Consumer Gruppen verteilt. Bei einer Gruppe von Consumern wird eine Nachricht immer nur an einen Consumer der Gruppe verteilt [Apa16a]. Dieses Modell ist in Abbildung 10 dargestellt. Das Besondere ist ein Transaktionsprotokoll, mit dem eine hohe Fehlertoleranz erreicht wird.

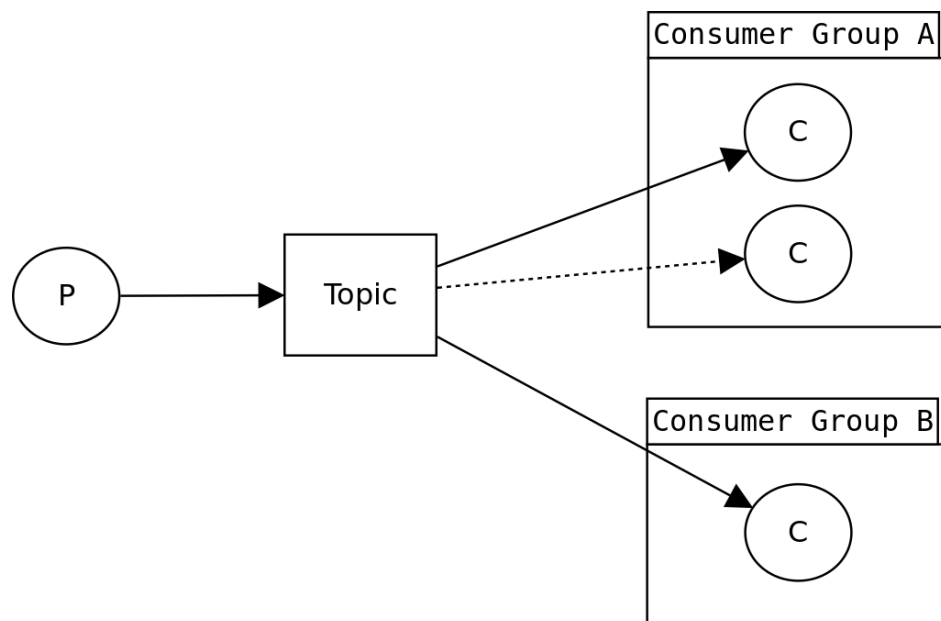


Abbildung 10: Apache Kafka Modell

Es gibt zahlreiche Client Implementierungen in verschiedenen Sprachen. Nicht alle sind ausgereift und decken somit ein kleineres Spektrum ab, als z.B die Anzahl der AMQP Client Implementierungen. Alle Implementierungen können in [Rao12] eingesehen werden.

2.3.4 Streaming Text Orientated Messaging Protocol

Wie der Name schon sagt, handelt es sich bei STOMP um ein textbasiertes Protokoll, um mit Brokern zu kommunizieren. Ursprünglich wurde es für den Einsatz in Skriptsprachen wie Ruby oder Python entwickelt. In den meisten Fällen bieten Broker STOMP als Alternative zu anderen Protokollen an. Besonders erwähnenswert ist, dass sich STOMP über eine WebSocket Verbindung betreiben lässt. Webanwendungen können daher, über die bidirektionale WebSocket Verbindung und STOMP, mit einer MOM kommunizieren. Das Protokoll bietet ein pub/sub Modell an. Die Verwendung des Protokolls ist immer Broker spezifisch. „The STOMP specification does not prescribe what kinds of destinations a broker must support, instead the value of the destination header in SEND and MESSAGE frames is broker-specific.“ [Piv07a], daher muss beim Wechsel des Brokers unter Umständen die *destination* bei der Verwendung von STOMP angepasst werden. Es ist beispielsweise möglich, dass ein AMQP Broker auch STOMP anbietet und die *destinations* auf das interne AMQP Modell abbildet. Die Routing Möglichkeiten eines AMQP 0-9-1 Brokers können dadurch auch über das STOMP Protokoll verwendet werden.

2.3.5 Message Queue Telemetry Transport

MQTT ist ein leichtgewichtiges Protokoll um mit Brokern zu kommunizieren. Es wird als „machine-to-machine (M2M)/"Internet of Things" connectivity protocol“ [MQT16] beschrieben. Wie STOMP wird es häufig als Alternative zu anderen Protokollen angeboten. Es gibt jedoch auch reine MQTT Broker. Es bietet ein pub/sub Modell an, dass ähnliche Funktionalitäten wie AMQP Topic Exchanges bietet. Das Besondere ist die Einfachheit des Protokolls. Es besitzt gerade einmal fünf API Methoden und kommt auch mit geringer Bandweite und hohen Latenzen aus. Zusätzlich lässt es sich auch über WebSockets betreiben. Ein typisches Einsatzgebiet ist zum Beispiel, wenn *mobile Applications* oder *Embedded Systems* mit einem Broker kommunizieren müssen [Pip13].

3 Analyse

3.1 Schnittstellen und Protokolle

3.1.1 Vergleich der Schnittstellen und Protokolle

In den Grundlagen dieser Arbeit wurden die wichtigsten Schnittstellen und Protokolle vorgestellt, mit denen verteilte Systeme über eine MOM kommunizieren können. Wichtig für die Auswahl der Schnittstellen und Protokolle, für ein Alarmierungssystem, sind folgende Kriterien:

- Funktionalität
- Verfügbarkeit unter Brokern
- Verfügbarkeit unter Clients

Alle Protokolle bieten ein pub/sub Modell an. Eine Besonderheit ist, dass das Kafka Protokoll eine hohe Fehlertoleranz und hohen Durchsatz bietet. Dafür muss bei der Verwendung auf ein Routing verzichtet werden. In [Qua16] wird der Durchsatz des Kafka Brokers mit einem AMQP Broker verglichen. Als Fazit wird empfohlen, Kafka zu verwenden, wenn 100 000 Nachrichten oder mehr pro Sekunde verarbeitet werden.

Beim STOMP Protokoll geht dagegen allein bei der Übertragung Geschwindigkeit verloren, da es im Gegensatz zu den anderen Protokollen textorientiert arbeitet. Dafür kann man es auch über eine WebSocket Verbindung verwenden. Außerdem bezieht sich das Protokoll auf kein bestimmtes Modell, das von einem Broker unterstützt werden muss.

MQTT ist für Einsätze mit niedriger Bandbreite und hohen Latenzen ausgelegt. Im Austausch muss auf komplexe Routing Möglichkeiten verzichtet werden.

Wenn AMQP in der Version 0-9-1 verwendet wird, bietet es ein sehr komplexes Modell, mit großen Routing Möglichkeiten. Bei allen Protokollen gibt es, je nach Broker Implementierung, Möglichkeiten eine Verschlüsselung und Authentifizierung durchzuführen.

In Tabelle 1 werden die von Brokern angebotenen Schnittstellen und Protokolle verglichen. Dafür werden fünf der am häufigsten verwendeten Open Source Broker verwendet. Dabei handelt es sich um ActiveMQ, ActiveMQ Artemis, RabbitMQ, Qpid und Apache Kafka.

Man erkennt, dass AMQP und JMS weit verbreitet sind. Da das AMQP Protokoll die JMS Funktionalitäten emulieren kann, ist es auch nicht verwunderlich, dass Broker AMQP und JMS oft parallel anbieten. STOMP wird von drei der fünf verglichenen

	ActiveMQ	Artemis	RabbitMQ	Qpid	Kafka
AMQP	1.0	1.0	0-9-1 und 1.0	1.0	×
JMS	1.1	1.1 und 2.0	1.1	1.1	×
STOMP	✓	✓	✓	×	×
MQTT	✓	✓	✓	×	×
Kafka	×	×	×	×	✓

Tabelle 1: Von Brokern angebotene Schnittstellen und Protokolle

Broker angeboten, genau wie MQTT. Kafka ist ein Spezialfall. Der Kafka Broker ist sehr spezialisiert und bietet lediglich das eigene Kafka Protokoll an.

Doch nicht nur die Verbreitung unter Brokern ist bei der Auswahl von Bedeutung, sondern auch welche Client Implementierungen möglich sind. Aus den Anforderungen geht hervor, dass es sowohl Clients geben muss, die in Java geschrieben sind, sowie Browser Applikationen mit JavaScript. Für diese Anwendungsfälle wurden die Implementierungen in Tabelle 2 verglichen.

	Java Applikation	Browser Applikation
AMQP	✓	*
JMS	✓	×
STOMP	✓	✓
MQTT	✓	✓
Kafka	✓	×

Tabelle 2: Client Implementierungen von Schnittstellen und Protokolle

Es wird klar, dass in Java Applikationen ohne Einschränkungen auf alle Protokolle und Schnittstellen zurückgegriffen werden kann. Sollen Browser Anwendungen direkt als Client agieren, kann nur das STOMP oder das MQTT Protokoll über WebSockets verwendet werden. Die Verwendung von AMQP über WebSockets ist grundsätzlich möglich, es fehlen jedoch noch Implementierungen.

3.1.2 Begründung der Wahl von AMQP und STOMP

AMQP kann durch seine weite Verbreitung überzeugen. Es ist zum Standard Protokoll für Message Queuing geworden. Das zeigt sich in den vielen Broker und Client Implementierungen. Außerdem bietet es in Version 0-9-1, von den verglichenen Protokollen, die besten Routing Möglichkeiten. Das Kafka Protokoll besitzt zwar eine hohe Fehler-toleranz und Geschwindigkeit, ist dagegen jedoch weniger verbreitet und erfordert die Verwendung des Kafka Brokers. Dieser wird empfohlen, wenn 100 000 Nachrichten oder mehr pro Sekunde verarbeitet werden. Da von dem Prototypen jedoch nicht erwartet wird diese große Anzahl zu verarbeiten verliert Kafka an Bedeutung.

Möchte eine Browser Anwendung mit einer MOM kommunizieren, sollte eine bidirektionale WebSocket Verbindung verwendet werden. Sie ermöglicht eine asynchrone Kommunikation mit einem Server für Webanwendungen. Mann kann sich daher zwischen STOMP und MQTT entscheiden, für die es Implementierungen über WebSockets gibt. Da es keine Anforderungen gibt, niedrige Bandweiten und hohe Latenzen zu unterstützen, wird das STOMP Protokoll gewählt. Dieses kann einfach in Kombination mit AMQP verwendet werden, da sich die STOMP *destinations* auf das AMQP Modell abbilden lassen. In Zukunft werden sich vielleicht ausgereifte Implementierungen für AMQP über WebSockets etablieren. Dann entfällt die Notwendigkeit dafür STOMP zu verwenden und man braucht nur noch das AMQP Protokoll zur Kommunikation.

3.2 Broker

3.2.1 Vergleich der Broker Implementierungen

Gesucht ist ein Broker, der AMQP und STOMP über WebSockets unterstützt. Diese Voraussetzungen werden von ActiveMQ, ActiveMQ Artemis und RabbitMQ erfüllt. Jedoch unterstützt lediglich RabbitMQ AMQP 0-9-1 und verwendet daher auch das, in den Grundlagen vorgestellte, AMQP Modell.

ActiveMQ wird als der populärste Broker beschrieben [Apa16b]. Abbildung 11 deutet jedoch an, dass ActiveMQ an Beliebtheit verliert und dass der neuere Broker RabbitMQ mittlerweile beliebter ist. Dort wird das Suchinteresse der Begriffe *activemq* und *rabbitmq* auf Google, im zeitlichen Verlauf, dargestellt.

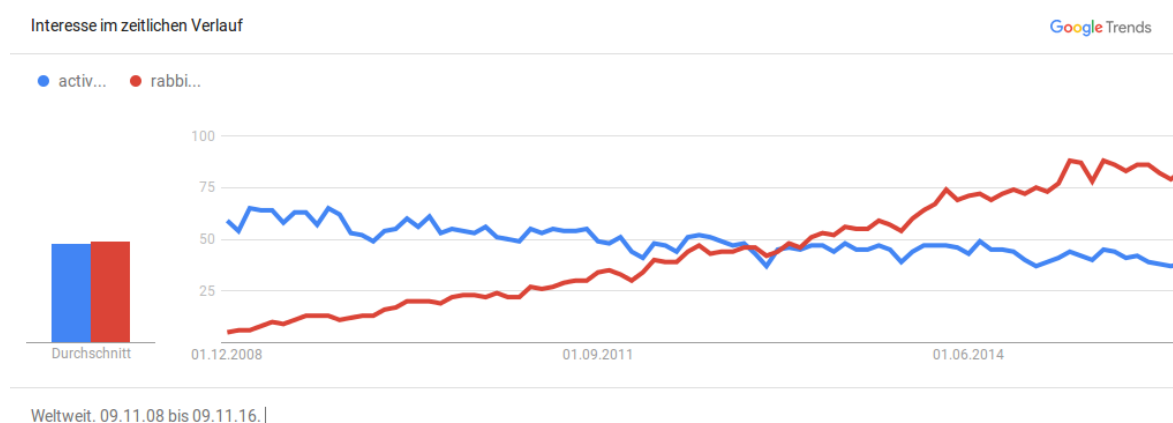


Abbildung 11: Trend ActiveMQ und RabbitMQ, Bildquelle: [Goo16]

ActiveMQ ist seit vielen Jahren im Einsatz und bietet eine Vielzahl von Funktionen. Es unterstützt AMQP 1.0, JMS 1.1 und STOMP. Das in den Grundlagen vorgestellte AMQP 0-9-1, wird nicht angeboten. Wenn die JMS API eingesetzt werden soll, lässt sich ActiveMQ einfach in Spring Applikationen integrieren [Apa16c]. Es gibt, ähnlich

wie bei einem AMQP Topic Exchange, die Möglichkeit über *Wildcards* ein Routing der Nachrichten bereitzustellen. Browser Applikationen können auch über eine Ajax Schnittstelle mit dem Broker kommunizieren. Dafür wird keine WebSocket Verbindung benötigt. Über einen *Polling* Mechanismus können dort Nachrichten empfangen werden.

Neben ActiveMQ, gibt es auch noch ActiveMQ Artemis. Das ist der Codename für den HornetQ Code, der der Apache Foundation gespendet wurde. HornetQ wurde ehemals von JBoss entwickelt. Artemis unterscheidet sich vor allem darin von ActiveMQ, dass es neben JMS 1.1, auch den neuen Standard JMS 2.0 unterstützt. Konkrete Vorteile entstehen dadurch jedoch noch nicht. Artemis lässt sich ebenfalls mit JMS einfach in Spring Applikationen integrieren. Eventuell wird der Artemis Code, den derzeitigen ActiveMQ Code, in dem nächsten Major Release von ActiveMQ ablösen. In [Apa15] heißt es dazu: „It is possible that Artemis will eventually become the successor to ActiveMQ 5.x (and that it might eventually be branded as ActiveMQ 6.x) [...]“.

Als letztes bleibt noch RabbitMQ. Im Gegensatz zu den anderen zwei Brokern, ist RabbitMQ in der Programmiersprache Erlang und nicht in Java geschrieben. RabbitMQs Fokus liegt auf der AMQP 0-9-1 Implementierung. Es kann daher das komplette Routing Potential von AMQP 0-9-1 verwendet werden. Der Broker lässt sich über AMQP in das Spring Framework integrieren. Die Referenzimplementierung für den AMQP Client im Spring Framework ist sogar speziell für den RabbitMQ Broker geschrieben [Piv16b]. Es unterstützt jedoch auch AMQP 1.0, STOMP und MQTT. RabbitMQ erweitert zudem den AMQP 0-9-1 Standard um weitere Funktionen. Alle Protokoll Erweiterungen werden in [Piv16a] beschrieben. Die für diese Arbeit interessanten Erweiterungen werden kurz vorgestellt.

Eine dieser Erweiterungen sind die *Exchange to Exchange Bindings*. Sie ermöglichen es, zwei Exchanges durch ein Binding zu verbinden. Normalerweise lassen sich nur ein Exchange und eine Queue verbinden. Eine Nachricht geht üblicherweise von einem Producer zu einem Exchange, weiter in eine Queue und letztendlich zu einem Consumer. Mit dieser Erweiterung lässt sich die Nachricht jedoch durch beliebig viele Exchanges routen, siehe Abbildung 12. Dadurch lassen sich komplexere Routings erstellen als mit dem Standard AMQP Modell.

Die Erweiterung *Sender-selected Distribution*, soll es einem Publisher ermöglichen selbst die Consumer zu bestimmen, die seine Nachricht erhalten. Beim AMQP Protokoll kann ein Publisher normalerweise keinen Einfluss darauf nehmen, an welche Consumer seine Nachrichten verteilt werden. Um diese Funktionalität trotzdem anbieten zu können kann ein Publisher die Argumente *CC* und *BCC* setzen. Diese Argumente entstammen zwei Header Feldern von Emails und sollen den gleichen Zweck erfüllen wie ihre Pendants aus den Emails. Sie dienen bei RabbitMQ als zusätzliche Routing Keys. Eine

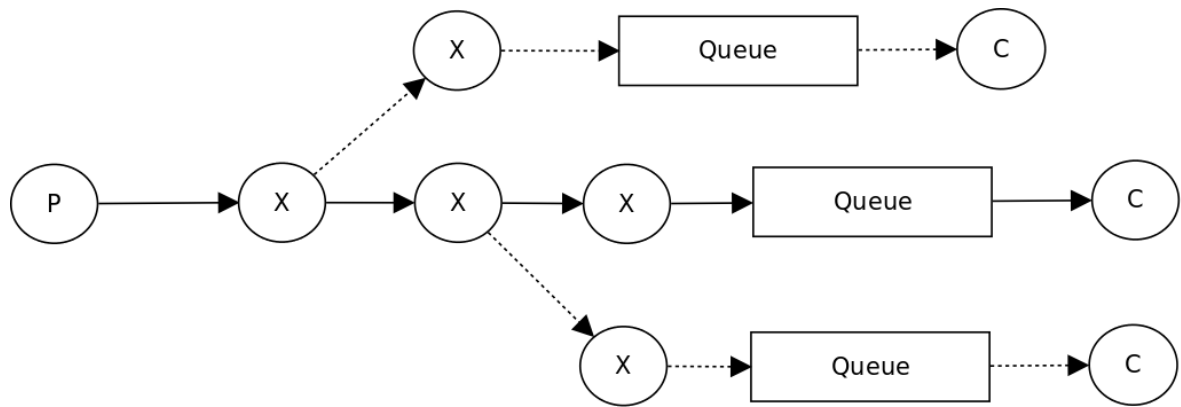


Abbildung 12: RabbitMQ Exchange to Exchange Bindings

Nachricht wird über ein Binding ausgeliefert, wenn der Routing Key oder ein Wert aus den CC oder BCC Argumenten mit dem Routing Key der Binding übereinstimmt. Die Werte des BCC Arguments werden vor dem Ausliefern der Nachricht über ein Binding entfernt. Wie dieser Mechanismus verwendet werden kann ist in Abbildung 13 zu sehen.

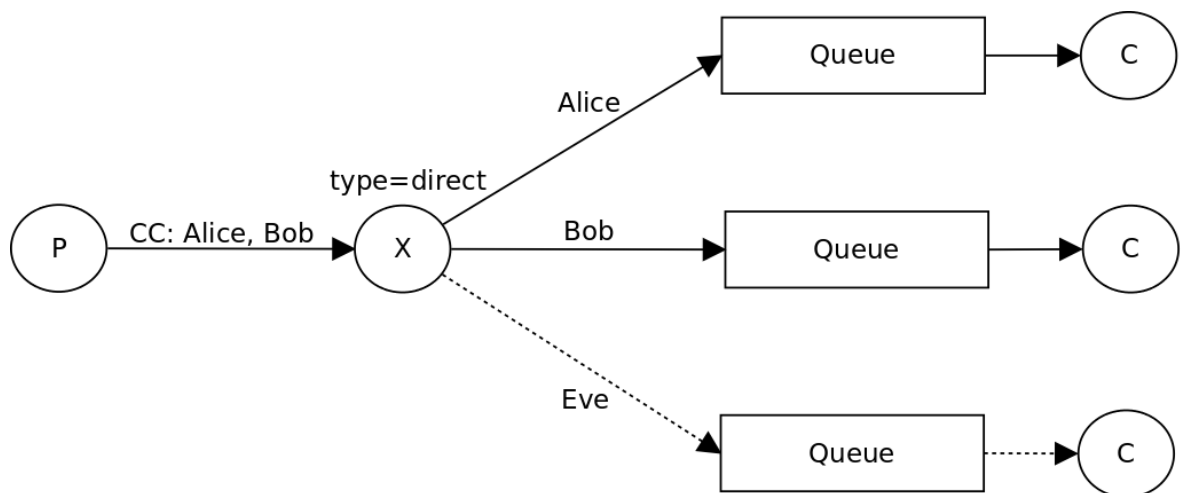


Abbildung 13: RabbitMQ Sender-selected Distribution

Als letzte erwähnenswerte Erweiterung wird noch kurz die Möglichkeit vorgestellt, einen RPC über den RabbitMQ Broker durchzuführen. Bei einem RPC möchte ein Client synchron oder auch asynchron eine Funktion auf einem anderen System aufrufen. Das Ergebnis dieses Aufrufs soll an den Client zurückgegeben werden. Für ein solches *Request-response* Szenario kann man die Attribute `reply_to` und `correlation_id` bei einer Nachricht setzen. Der Client setzt in dem `reply_to` Argument seiner Nachricht, den Namen einer Queue und in `correlation_id` eine eindeutige Kennung für die Anfrage. Der Server empfängt diese Nachricht und führt seine Funktion aus. Anschließend schreibt er eine Nachricht, mit dem Ergebnis der Funktion als Inhalt, in die Queue, die im `reply_to` Argument angegeben war. Zusätzlich setzt er in der Nachricht das `correlation_id` Argument mit dem selben Wert, wie dem aus der Anfrage. Der Client empfängt

die Antwortnachricht und kann sie seiner vorherigen Anfrage zuordnen. In Abbildung 14 ist ein einfacher Aufbau für das Durchführen eines RPCs über RabbitMQ zu sehen.

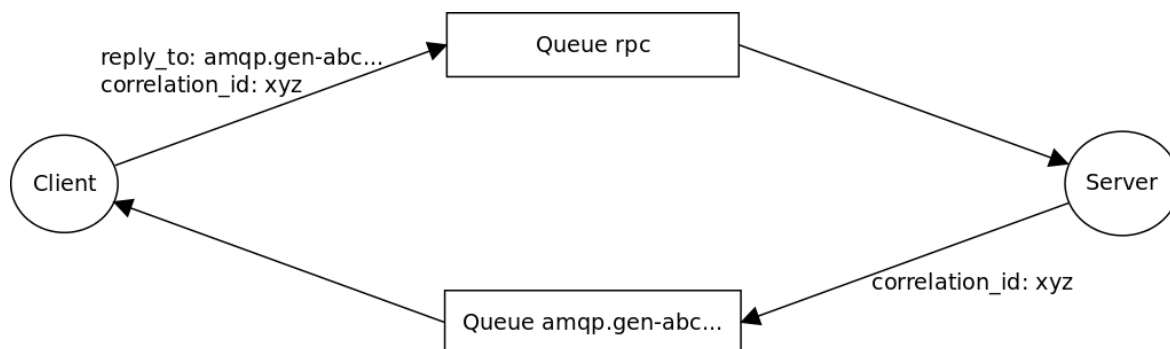


Abbildung 14: RabbitMQ RPC

Der RabbitMQ Broker ist durch Plugins erweiterbar. Dadurch lassen sich fehlende Funktionalitäten hinzufügen. Allerdings müssen die Plugins in Erlang geschrieben werden. Schlecht programmierte Plugins können zu einem Absturz des Systems führen. Daher wird in dieser Arbeit nicht weiter darauf eingegangen, eigene Plugins zu schreiben, um speziell für das Alarmierungssystem weitere Funktionalitäten hinzuzufügen. Es wird nur auf die Plugins eingegangen, mit denen RabbitMQ bereits ausgeliefert wird. Eines davon ermöglicht beispielsweise die Verwendung des STOMP Protokolls.

3.2.2 Begründung der Wahl von RabbitMQ

Alle drei verglichenen Broker haben einen großen Funktionsumfang. Doch der von RabbitMQ ist der Überzeugendste. Das bereits umfangreiche AMQP 0-9-1 Modell wird um sinnvolle und nützliche Funktionen erweitert. Da die Referenzimplementierung für AMQP im Spring Framework speziell für RabbitMQ geschrieben wurde, ist davon auszugehen, dass die Verwendung dieses Brokers lange Zeit unterstützt wird. Außerdem lässt sich ein Trend erkennen, dass RabbitMQ mittlerweile häufiger zum Einsatz kommt als seine Konkurrenten. Aus diesen Gründen wird RabbitMQ als MOM im Prototyp des Alarmierungssystems eingesetzt. Damit wird auch das erweiterte AMQP 0-9-1 Modell die Grundlage der Kommunikation sein.

3.3 Spring Projekte

Für das Spring Framework gibt es eine Vielzahl an Projekten, die zusätzliche Funktionalitäten bereitstellen. Einige dieser Projekte können verwendet werden, um mit dem RabbitMQ Broker zu kommunizieren. Ausgewählt wurden solche, die das Protokoll AMQP oder STOMP verwenden. Sie können für den Prototypen des Alarmierungssystems verwendet werden. Teilweise bauen die Projekte aufeinander auf. Im Folgenden

wird die Funktionalität und das mögliche Einsatzgebiet der verschiedenen Projekte analysiert.

3.3.1 Spring AMQP

Zum einen gibt es das Spring AMQP Projekt. Dieses Projekt stellt einen AMQP Client für die AMQP 0-9-1 Spezifikation bereit. Es handelt sich dabei jedoch lediglich um eine Abstraktionsschicht für AMQP Implementierungen. Die konkrete Referenzimplementierung ist speziell für den RabbitMQ Broker geschrieben, wie schon in [Piv16b] erwähnt. Mit Spring AMQP kann theoretisch auf jede Funktion des AMQP 0-9-1 Protokolls zugegriffen werden. Es sollte daher verwendet werden, wenn man auf spezielle Funktionalitäten von AMQP und RabbitMQ zurückgreifen will. Das Projekt beschränkt sich jedoch nicht darauf, lediglich eine *low-level* Abstraktion für AMQP basierte Kommunikation bereitzustellen. Stattdessen wird auch eine *high-level* Abstraktion zum Versenden und Empfangen von Nachrichten, sowie zum automatisierten Erstellen von Exchanges, Queues und Bindings bereitgestellt.

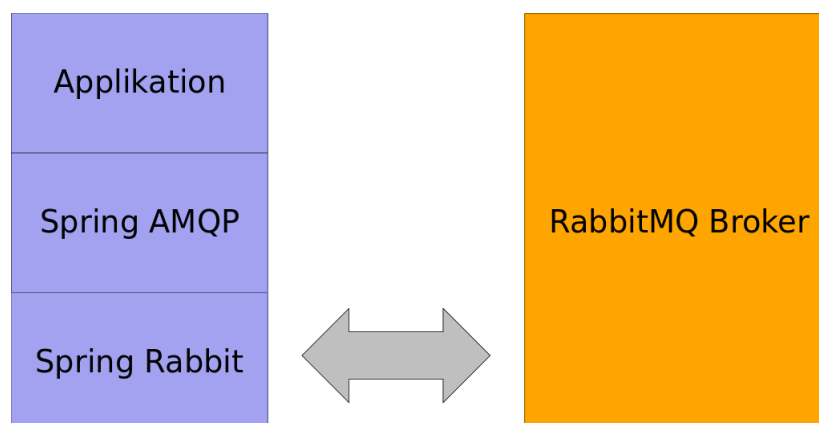


Abbildung 15: Spring AMQP

3.3.2 Spring Integration

Spring Integration baut auf dem Spring AMQP Projekt und weiteren Modulen zum Nachrichtenaustausch mit anderen Schnittstellen auf. Spring Integration vereinheitlicht die Integration der Applikation mit anderen Systemen. Dies wird über Implementierungen von *Enterprise Integration Patterns* erreicht, siehe dafür [HW03]. Spring Integration unterstützt eine Vielzahl von Schnittstellen und Protokollen. So kann man über ein einheitliches Framework die Applikation über AMQP, Kafka oder JMS mit einem Message Broker verbinden, Emails verarbeiten oder Nachrichten in ein *Social Network* wie Twitter posten. Die Integration erfolgt über einzelne Bausteine, die miteinander verbunden werden können. Beispielsweise können Nachrichten über einen *AMQP Inbound Channel Adapter* empfangen werden, von einem *Transformer* in ein Modell

der Applikation übersetzt werden und anschließend von einem *Handler* verarbeitet werden. Diese drei beispielhaft genannten Komponenten können einfach ersetzt oder wiederverwendet werden. Sollen jedoch neue Exchanges oder Queues in dem Broker erstellt werden, muss trotzdem auf Spring AMQP zurückgegriffen werden. Ob man Nachrichten eines Message Brokers nun über Spring AMQP oder Spring Integration verarbeitet, muss in jedem Fall abgewägt werden. Die Stärken von Spring Integration liegt darin, dass sich die verwendeten Konzepte prinzipiell für jede Schnittstelle verwenden lassen.

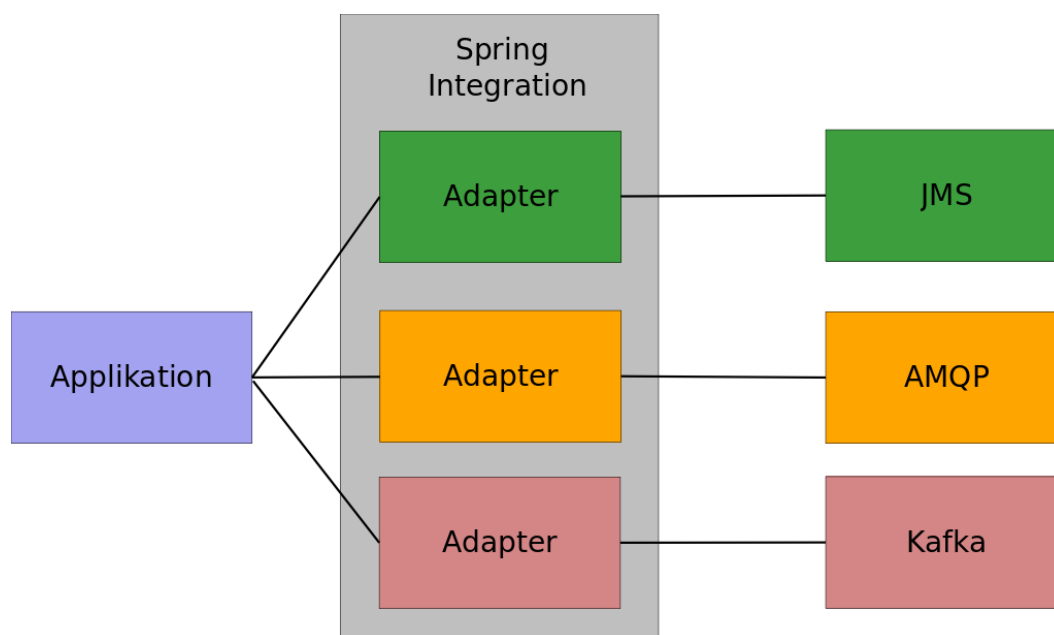


Abbildung 16: Spring Integration, Bildquelle: [Pad15]

3.3.3 Spring Cloud Stream

Spring Cloud Stream ist ein Framework um Microservices in einer ereignisgesteuerten Architektur zu entwickeln. In der eigenen Beschreibung des Frameworks heißt es: „Spring Cloud Stream builds upon [...] Spring Integration to provide connectivity to message brokers.“ [Piv16e]. Man erkennt, dass die Funktionalitäten von Spring Integration genutzt werden um die Microservices über einen Broker zu verbinden. Sie sollen dabei über ein allgemein gültiges und brokerübergreifendes pub/sub Modell kommunizieren. Unterstützt werden die Broker RabbitMQ und Kafka. RabbitMQ spezifische Funktionalitäten, wie das Routing von Nachrichten, können mit Spring Cloud Stream nicht mehr genutzt werden. Es können nur noch die Routing Funktionalitäten von Topic Exchanges genutzt werden, die von Spring Cloud Stream verwendet werden. Daher sollte dieses Framework nur verwendet werden, wenn man wie vorgesehen, Microservices entwickeln möchte, die über ein pub/sub Modell kommunizieren. Wie die einzelnen Projekte aufeinander aufbauen ist in Abbildung 17 dargestellt.

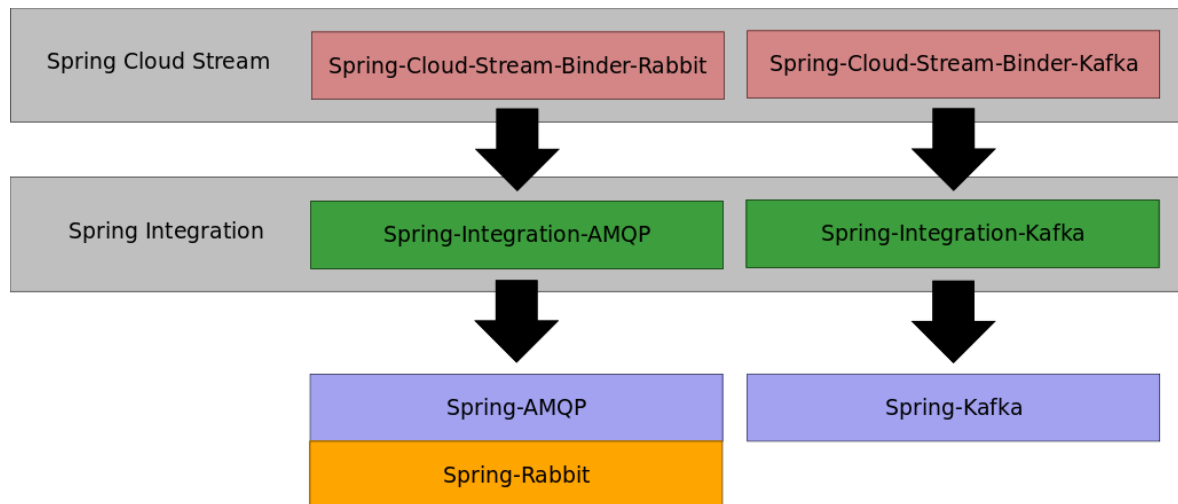


Abbildung 17: Spring Cloud Stream

3.3.4 Spring WebSocket

Das letzte hier vorgestellte Projekt ist Spring WebSocket. Mit dem Spring Framework lassen sich neben klassischen Webanwendungen auch solche erstellen, in denen zwischen Client und Server bidirektional und asynchron über Websockets kommuniziert wird. Dabei werden die Daten über das STOMP Protokoll ausgetauscht. Die Serverapplikation kann dabei als Vermittler zwischen Clients und einem STOMP Broker wie RabbitMQ agieren. Dadurch wird es möglich über den Broker asynchron Nachrichten an einen einzelnen Client oder an eine Gruppe von Clients zu schicken. Authentifizierung und Autorisierung der User erfolgt dabei in der Serveranwendung mit dem Spring Security Framework und nicht über den Broker. Die beim STOMP Protokoll verwendeten Adressen werden von der Serveranwendung, je nach Konfiguration, in drei verschiedene Funktionalitäten übersetzt. Diese werden kurz beschrieben und sind zudem in Abbildung 18 dargestellt.

- Adressen die mit `/app` beginnen werden hier von einem *AnnotationMethodMessageHandler* in der Webanwendung selbst verarbeitet. Eine Nachricht eines Clients an die Adresse `/app/alerts.open` wird an einen internen Handler für `alerts.open` Nachrichten übergeben. Dieser Handler kann auf die Nachricht reagieren. Die Nachricht wird nicht an den STOMP Broker weitergeleitet.
- Adressen die mit `/user` beginnen werden hier von einem *UserDestinationMessageHandler* verarbeitet. Wenn ein Client mit dem User `michael.mueller` und der Session Id `123`, die Adresse `/user/queue/alerts` abonniert, wird diese Adresse zu `/queue/alerts-user123` übersetzt. Wird nun eine Nachricht an die Adresse `/user/michael.mueller/queue/alerts` geschickt, wird die Nachricht an alle `/queue/alerts-user{SessionId}` Adressen des Users `michael.mueller` geschickt.

- Für die letzte Funktionalität werden hier Adressen die mit `/topic` beginnen von einem *StompBrokerRelayMessageHandler* verarbeitet. Dieser ist ein Vermittler zwischen der Serveranwendung und dem STOMP Broker. Nachrichten an solche Adressen werden an den Broker weitergeleitet. Bei so abonnierten Adressen, wird dafür gesorgt, dass Nachrichten vom Broker an den Client weitergegeben werden. Für RabbitMQ bietet sich hier eine andere Konfiguration an. Dort werden Adressen, die mit `/queue` beginnen, auf eine Queue gemappt und Adressen, die mit `/exchange` beginnen, auf ein Exchange. Das Abbilden von STOMP Adressen auf das intern verwendete Modell ist immer anwendungsspezifisch.

STOMP over WebSocket messages

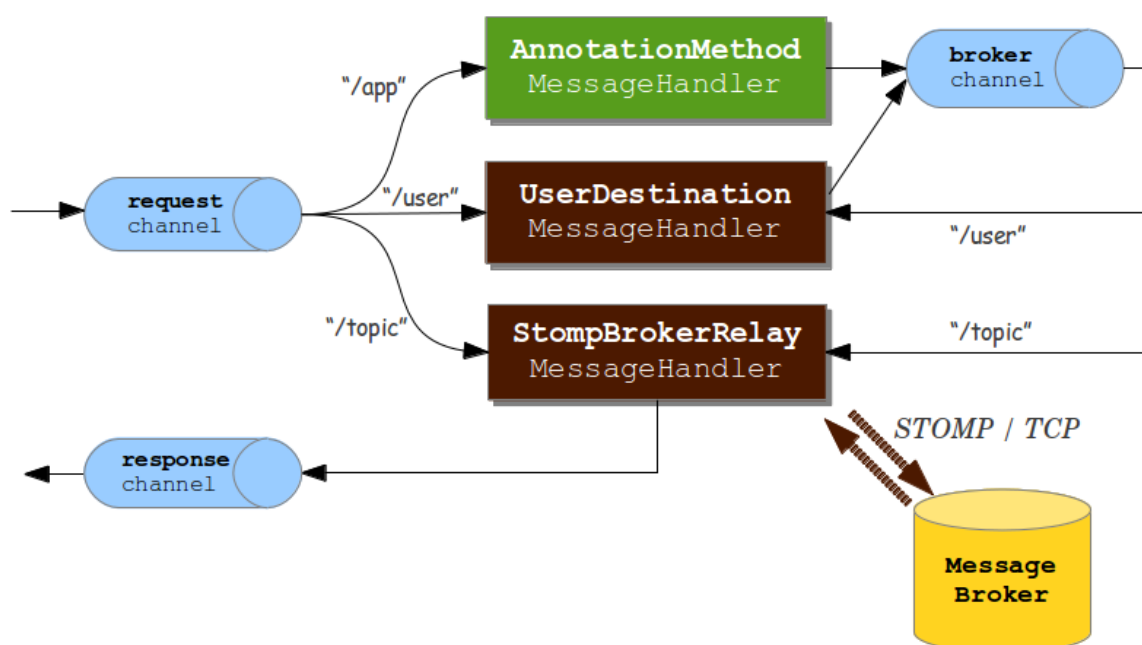


Abbildung 18: Spring WebSocket, Bildquelle: [Sto13]

Spring WebSocket ermöglicht es asynchron mit Clients zu kommunizieren. Über ein pub/sub Modell können Nachrichten an Clients verteilt werden. Ebenfalls ist es möglich über ein PTP Modell speziell einen Client anzusprechen. Eine Alternative wäre, dass Clients direkt über STOMP mit dem Broker kommunizieren. Dann muss er jedoch für die Clients erreichbar sein. Authentifizierung und Authorisierung müssen ebenfalls über den Broker laufen und können nicht mehr über eine eigene Applikation gesteuert werden. Eine PTP Kommunikation mit den Clients lässt sich direkt über den Broker nur schwer umsetzen.

3.3.5 Einsatzgebiete der Spring Projekte

Der Vorteil von Spring Cloud Stream ist, dass die Applikation unabhängig von der Broker Implementierung wird. Dafür kann man nur noch auf ein allgemeines pub/sub Modell und nicht mehr auf die Broker spezifischen Funktionalitäten zurückgreifen. Bei der Analyse der Protokolle ist man zum Ergebnis gekommen, dass das Kafka Protokoll mit dem Kafka Broker nicht zu den Anforderungen passt. Denn dort liegt der Fokus auf einer hohen Leistung und dafür gibt es wenig Routing Möglichkeiten. Mit dem Spring Cloud Stream Projekt kann man nachträglich den RabbitMQ Broker durch den Kafka Broker ersetzen, wenn das allgemeine pub/sub Modell ausreichend ist. Man arbeitet lediglich mit einer unabhängigen Abstraktion und braucht keine genauen Kenntnisse über den eingesetzten Broker. Welcher Broker in der produktiven Umgebung zum Einsatz kommt, ist bei der Entwicklung nicht mehr von Bedeutung.

Ähnlich verhält es sich mit Spring Integration, bei dem über Adapter auf die Schnittstellen zugegriffen wird. In dem sogenannten *Integration Flow*, den man mit dem Framework aufbaut, kann man den AMQP Adapter einfach austauschen, um mit einem anderen Broker zu kommunizieren. Zusätzlich arbeitet man mit einem Modell, dass sich nicht nur auf die Kommunikation mit einem Broker beschränkt, sondern für jegliche Art der Kommunikation genutzt werden kann.

Ansonsten wird der Client von Spring AMQP verwendet, der es erlaubt auf alle AMQP Funktionalitäten zurückzugreifen. In jedem Fall muss erst analysiert werden, auf welche Funktionalitäten zurückgegriffen werden muss. Anschließend kann die Wahl getroffen werden welches Spring Projekt verwendet werden muss.

Das Spring WebSocket Projekt sollte auf jeden Fall verwendet werden, wenn Clients über WebSocket Verbindungen an das System angebunden werden. Zwar könnten Clients auch direkt mit einem Message Broker kommunizieren, allerdings kann so vermieden werden, dass der Broker für Authentifizierung und Autorisierung der Clients verwendet werden muss. Zusätzlich deckt man mit Spring WebSocket nicht nur einfache pub/sub Modelle ab, sondern kann auch über ein PTP Modell in beide Richtungen mit den Clients kommunizieren.

4 Konzepte

4.1 Dezentrale Alarmierung

Bei einem dezentralen Alarmierungssystem werden über den Broker Alarme verteilt. Dabei weiß das Alarmierungssystem beim Verteilen nicht, ob es Clients gibt, die den Alarm empfangen. Die Clients melden sich, sobald sie Alarme empfangen wollen über eine selbst erstellte Queue bei einem Exchange an, aus dem sie die Alarmnachrichten empfangen. Das Alarmierungssystem erfährt über Rückmeldung der Clients, wer den Alarm empfangen hat und bearbeitet. Die Schwierigkeit besteht bei diesem Ansatz darin, dass ein Client nur die Alarme empfängt, die für ihn relevant sind. Wenn möglich, soll ein Client über eine private Queue, aus einem einzigen Exchange, alle Alarme empfangen können.

4.1.1 Exchange to Exchange Bindings

Die *Exchange to Exchange Bindings* Erweiterung des RabbitMQ Brokers ermöglicht es, wie schon erwähnt, zwei Exchanges direkt mit einem Binding zu verbinden. Dies kann genutzt werden, um über ein Routing Alarm Nachrichten zu verteilen. Ausgangspunkt soll ein einziges Exchange sein, in das alle Alarm Nachrichten geschrieben werden. Der Routing Key der Nachricht setzt sich aus dem Typ und dem Schweregrad des Alarms zusammen und hat das Format `{typ}.{severity}`. Ziel ist es, die Alarmnachricht in verschiedene Exchanges zu routen. Es kann beispielsweise ein Exchange für Alarme eines bestimmte Typs geben. Ein Client, der nur diesen einen Typ empfangen möchte, braucht nur aus diesem einen Exchange die Nachrichten zu empfangen. Eine andere Möglichkeit wäre ein Exchange für eine bestimmte Gruppe von Personen. In dieses Exchange werden alle Typen von Alarmen geroutet, die für diese Gruppe von Bedeutung sind. Oder jeder Client hat ein persönliches Exchange, in das alle, für speziell diese Person, relevanten Nachrichten geroutet werden.

In Abbildung 19 ist ein beispielhafter Aufbau gezeigt. Dort gibt es vier Consumer, die alle unterschiedliche Nachrichten empfangen wollen:

- Ein Consumer möchte alle Alarme erhalten. Er empfängt seine Nachrichten daher aus dem `alerts` Exchange. Dieser Consumer könnte beispielsweise eine Applikation sein, die alle Alarme zur Analyse aufzeichnet.
- Ein weiterer Consumer möchte alle medizinisch relevanten Alarme erhalten. Diese empfängt er aus dem Exchange `alerts.medical`. Das könnte z.B eine Krankenschwester sein.

- Der nächste Consumer möchte zwei sehr unterschiedliche Alarme empfangen. Er hat dafür sein privates `user.michael-mueller` Exchange.
- Der letzte Consumer möchte lediglich informiert werden, wenn ein Feueralarm auftritt. Das wäre beispielsweise die Leitstelle der Feuerwehr.

Die Consumer empfangen ihre Alarme an unterschiedlichen Exchanges. Über die Bindings zwischen den Exchanges werden die Nachrichten dorthin geroutet. Über die Autorisierung des Brokers könnte der Zugriff an den Exchanges geregelt werden. Die Leitstelle der Feuerwehr wäre beispielsweise nur autorisiert Nachrichten aus dem `alert.fire` Exchange zu empfangen.

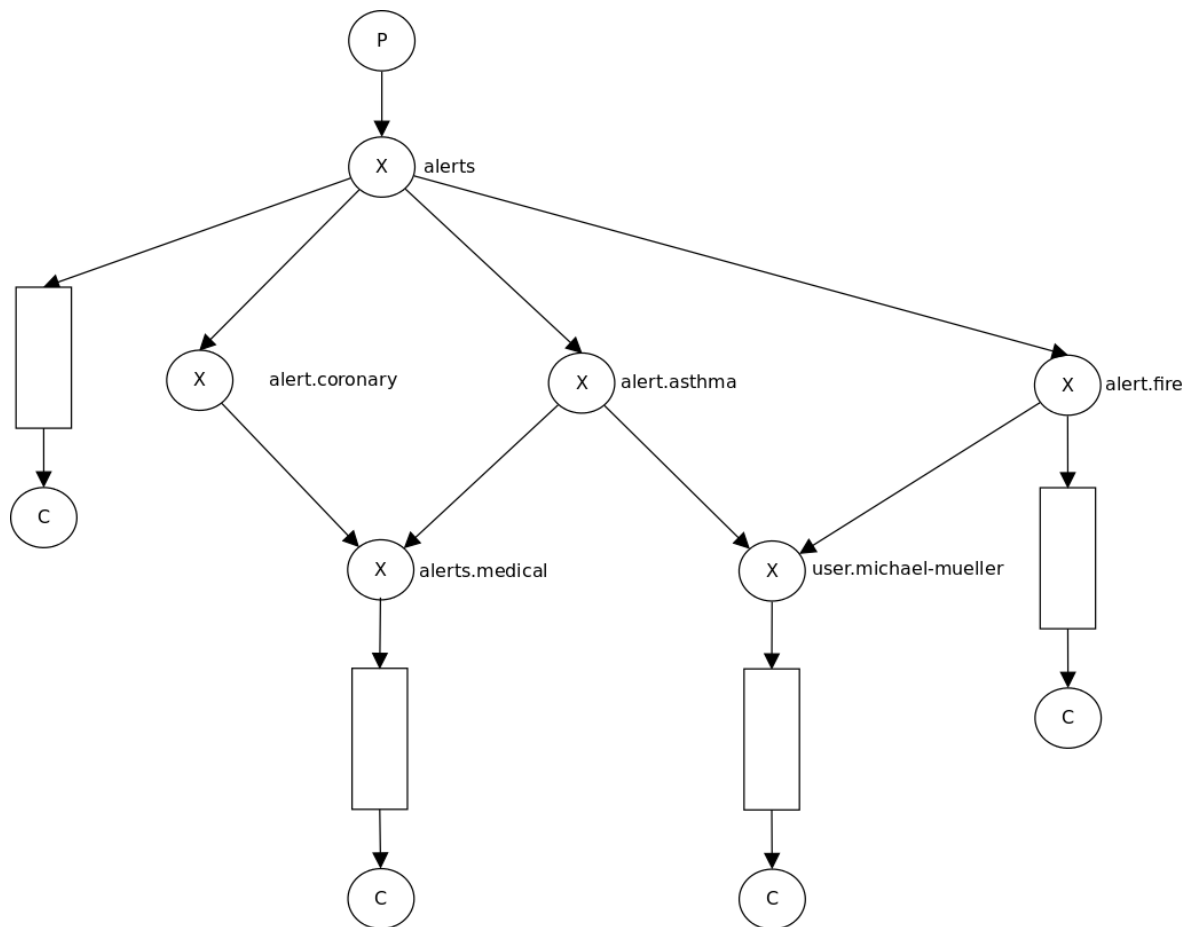


Abbildung 19: Konzept Exchange to Exchange Bindings

4.1.2 Headers Exchange

Das Headers Exchange ist das Exchange, mit den größten Routing Möglichkeiten. Es soll daher geprüft werden, ob es ausreicht alle Alarm Nachrichten in ein einziges Headers Exchange zu schreiben. Ein Client kann über Bindings bestimmen, welche Alarme aus dem Exchange an seine Queue geroutet werden. Es wäre wünschenswert, wenn ein Client lediglich ein Binding bräuchte, um seine Nachrichten zu erhalten. Zu diesem

Zweck werden alle Werte, die als Routing Key einer Nachricht verwendet werden können, als Header Werte angegeben. Eine Alarm Nachricht hat hier einen Header Wert für den Typ und einen für den Schweregrad. Beim Binding eines Users wird bestimmt, welche Kombination dieser Werte vorliegen muss, damit die Nachricht geroutet wird. Dieser Ansatz stößt jedoch an seine Grenzen, wenn man für einen Routing Key mehrere Werte angeben möchte. Eine Krankenschwester möchte beispielsweise Alarmer des Typs `coronary` und `asthma` empfangen. Dafür muss sie zwei Bindings erstellen. Eins für jeden der Werte. Dieses Problem ist in Abbildung 20 dargestellt.

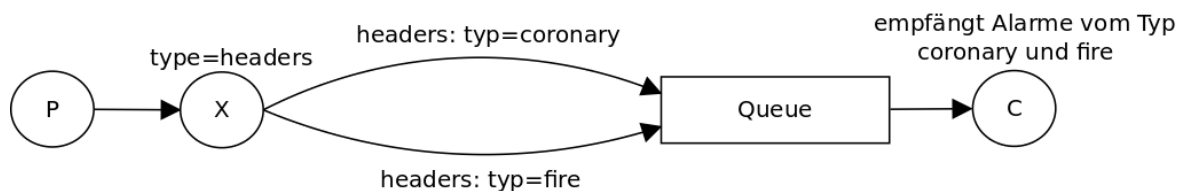


Abbildung 20: Konzept Headers Exchange

Ein weiteres Problem ist die Autorisierung. Es kann nicht überprüft werden, welche Header Werte bei einem Binding angegeben werden. Entweder ein Client darf ein Binding zu dem Exchange erstellen, oder nicht. Daraus folgt, dass die Clients alle Nachrichten empfangen können, die über dieses eine Exchange ausgeliefert werden. Es ist nicht möglich zu überprüfen, ob eine Krankenschwester über ihre Bindings auch wirklich nur medizinische Alarmer abrufen oder nicht.

4.2 Zentrale Alarmierung

Bei einem zentralen Alarmierungssystem bestimmt das System, welcher User oder welche Gruppe von Usern eine Alarm Nachricht erhalten soll. Der Broker soll nicht anhand von Metadaten des Alarms entscheiden, wohin die Nachricht geroutet werden muss. Er soll die Nachricht lediglich an einen vorgegebenen User oder Gruppe weiterleiten. Wie bei der dezentralen Alarmierung erfährt das System über eine Rückmeldung der Clients, ob der Alarm tatsächlich empfangen wurde.

4.2.1 Sender-selected Distribution

Die *Sender-selected Distribution* Erweiterung des RabbitMQ Brokers ermöglicht es, dass der Publisher einer Nachricht Einfluss darauf nehmen kann, wohin der Alarm geroutet wird. Dafür wird in diesem Konzept das `BCC` Attribut verwendet. Im ersten Schritt sollen Gruppen alarmiert werden. Das System könnte bestimmen, dass ein Alarm für alle Doktoren und Krankenschwestern relevant ist. Daraufhin werden die Werte `doctor` und `nurse` als `BCC` Werte beim Versenden der Alarm Nachricht gesetzt. Ähnlich wie im Konzept, das im Kapitel 4.1.1 vorgestellt wurde, gibt es für jede

Gruppe ein Exchange, in das Nachrichten geroutet werden. Daher kann auch hier eine Autorisierung an diesen Exchanges erfolgen. Zu sehen ist dieses Konzept in Abbildung 21.

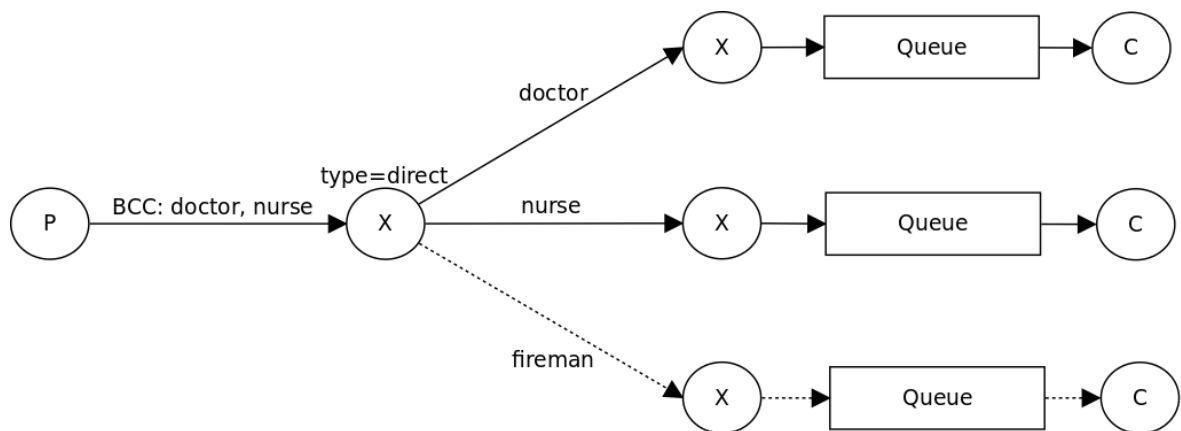


Abbildung 21: Konzept Sender-selected Distribution

Dieses Konzept könnte auch verwendet werden, um jeden User einzeln alarmieren zu können. Dafür müsste allerdings jeder User sein eigenes Exchange haben. Als Routing Key der Binding zum Alarm Exchange, wird die ID des entsprechenden Users gewählt. Beim Versenden eines Alarms werden als BCC Werte, die IDs der User gewählt, die alarmiert werden sollen.

4.2.2 Spring Websocket

Das Spring Websocket Projekt kann verwendet werden, um Clients über STOMP und Websockets an das Alarmierungssystem anzubinden. Eine Spring Websocket Applikation kann als Vermittler zwischen Broker und Clients agieren. Dann kann auf das PTP und pub/sub Modell von Spring Websocket zurückgegriffen werden. Entweder die User werden einzeln alarmiert über den *UserDestinationMessageHandler* oder die User können über den *StompBrokerRelayMessageHandler* Alarme aus verschiedenen Exchanges abonnieren. Dadurch würde sich auch ein dezentrales Alarmierungssystem aufbauen lassen.

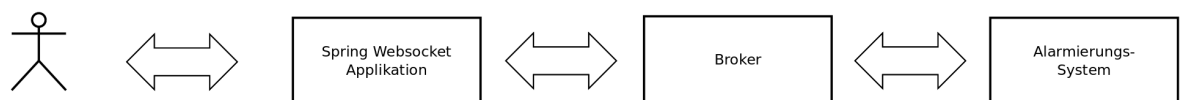


Abbildung 22: Konzept Spring Websocket

5 Prototyp

5.1 Architektur

Die aus dem Kapitel 4 erstellten Konzepte sollen genutzt werden, um eine Architektur für den Prototypen des Alarmierungssystem zu erstellen. Für eine dezentrale Alarmierung werden die Exchange to Exchange Bindings von RabbitMQ genutzt. User werden über Spring Websockets an das System angebunden. Diese Architektur ist in Abbildung 23 dargestellt. Sie besteht aus folgenden Komponenten:

- Ein oder mehrere **Gateways**. Sie sind dafür zuständig, einen Alarm von einem externen System entgegenzunehmen und zum Alarmierungssystem weiterzuleiten. Sie fungieren dabei als Adapter. Der Prototyp wird ein solches Gateway besitzen. Es nimmt Alarmer über eine REST Schnittstelle entgegen und schreibt sie in das `event.alert.new` Exchange.
- Eine **Alarmierungsapplikation**. Sie ist das Herzstück des Systems. Sie verwaltet alle Alarmer im System. Neue Alarmer werden über das `event.alert.new` Exchange empfangen. Anschließend werden User über das `event.alert` Exchange alarmiert und es wird auf deren Rückmeldung gewartet. Über diese Rückmeldungen werden den Alarmen ihre Bearbeiter zugeordnet, oder die Alarmer werden geschlossen. Über RPCs können Clients bereits vorhandenen Alarmer abrufen.
- In den **Alarm Exchanges** können Alarmer von Usern empfangen werden. Alle Alarmer werden in das `event.alert` Exchange geschrieben. Von dort aus werden sie in weitere Exchanges geroutet, aus denen Clients Alarmer empfangen können. Bei jedem der Exchanges soll eine Autorisierung stattfinden.
- Die Queue `event.response.alert` kann genutzt werden, um **Rückmeldungen** der Clients an die Alarmierungs Applikation weiterzugeben.
- Die Queue `event.command.alert` wird für **RPCs** genutzt, um beispielsweise alle noch offenen Alarmer abzufragen. Diese abgerufenen Daten werden über selbst erstellte, temporäre Queues zurückgeliefert.
- Eine **User Applikation**. Spring WebSocket wird hier in einer Serveranwendung genutzt, um es Clients zu ermöglichen mit dem System zu interagieren. Die User verwenden eine Webanwendung, um mit der Serveranwendung zu kommunizieren. Authentifikation und Autorisierung der Clients erfolgt mit Spring Security. Die Alarmierungsapplikation und die User Applikation sind ausschließlich über RabbitMQ verbunden.

- Eine **Autorisierungsapplikation**. Diese stellt mit dem Spring Security Framework einen OAuth2 Provider bereit. Er wird zur Authentifizierung und Autorisierung von Usern verwendet.

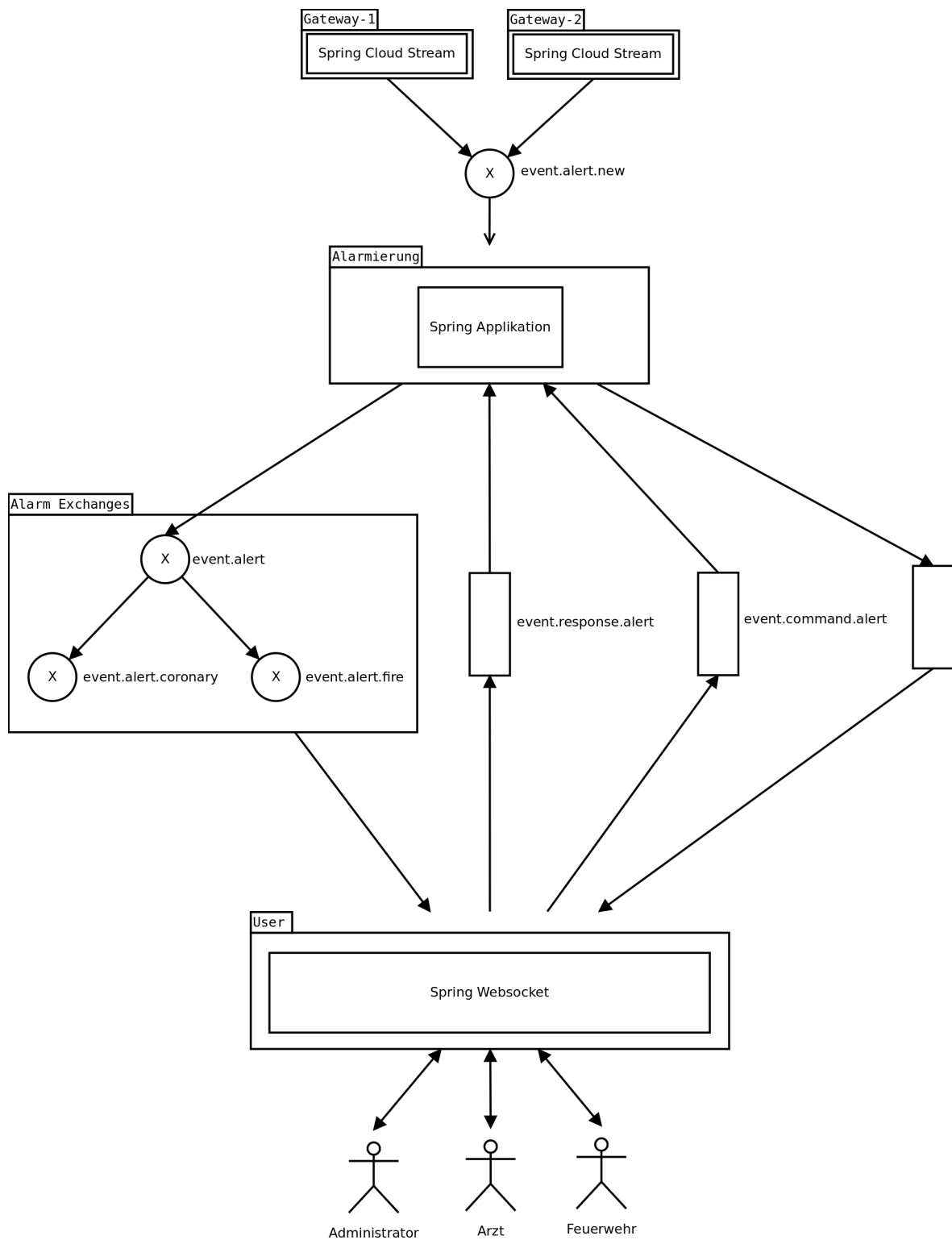


Abbildung 23: Prototyp Architektur

5.2 Projektstruktur

Das Projekt wird mit dem Build-Management Tool Maven verwaltet und erstellt. Der Grund ist, dass das System aus mehreren Java Applikationen besteht. Diese können einfach als separate Module in dem Maven Projekt verwaltet werden. Die Projektstruktur, mit den einzelnen Modulen, ist in Abbildung 24 dargestellt. Mit dem Tool lassen sich die Abhängigkeiten der Module verwalten. Die Abhängigkeiten zum Spring Framework werden dabei über das Framework eigene **Milestone Repository** aufgelöst, das unter <http://repo.spring.io/milestone> erreichbar ist. Für die Arbeit mit Spring Boot Applikationen gibt es ein spezielles Maven Plugin, mit dem beispielsweise ein ausführbares **jar** oder **war** Archiv der Applikation erstellt werden kann.

Das **alert** Modul erstellt die Alarmierungsapplikation. **authentication** ist eine Applikation zur Verwaltung und Authentifizierung von Benutzern. Das **common** Modul wird von anderen Modulen verwendet und stellt das gemeinsame Datenmodell zur Verfügung. **gateway** verwaltet die REST Schnittstelle, um neue Alarmer zu erstellen. Das **user** Modul enthält die User Applikation.

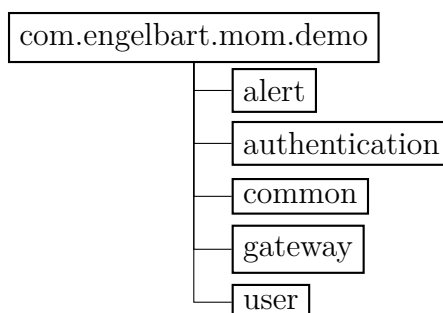


Abbildung 24: Projektstruktur

5.3 Datenmodell

Die verschiedenen Systeme kommunizieren über ein gemeinsames Datenmodell. Dieses kann in Abbildung 25 eingesehen werden. Das Modell wurde im Hinblick auf die Erweiterung durch weitere Alarmtypen konzipiert. Es steht im **common** Modul allen Applikationen zur Verfügung.

Im Paket **event** sind Klassen definiert, die unabhängig vom Anwendungsfall eine Basis für die Kommunikation bilden. Zentral ist hier die abstrakte **Event** Klasse. Sie besitzt eine eindeutige **id** und einen **type**. Der **type** bestimmt, um welche konkrete Implementierung der Klasse es sich handelt. **Event** ist die Basisklasse für alle anderen Klassen des Datenmodells. Die abstrakte **Command** Klasse wird verwendet, um RPCs durchzuführen. Dafür muss sie keine weiteren Methoden implementieren. Eine weitere abstrakte Klasse ist **EventResponse**. Sie wird verwendet, wenn auf ein vorhergehendes Event

geantwortet werden soll. Dafür lässt sich dieses im `request` Attribut referenzieren. In `principalName` wird die Kennung des Users festgehalten, der die Antwort verschicken möchte.

Im Paket `event.alert` dagegen sind die Klassen definiert, die speziell für den Einsatz in diesem Prototyp konzipiert wurden. Die abstrakte Klasse `Alert` dient als Basisklasse für alle Alarmer, die im System vorkommen können. Jeder Alarm hat ein `severity` Attribut, in dem der Schweregrad des Alarms angegeben ist. Im `acknowledgment` Attribut wird zu einer User Kennungen angegeben, ob der Alarm angenommen oder abgelehnt wurde. Das `closed` Attribut gibt an, ob der Alarm geschlossen wurde. Es gibt zwei beispielhafte Implementierungen dieser Klasse. Zum einen die `Fire` Klasse, die keine weiteren Attribute definiert und die `Coronary` Klasse, die zwei weitere Attribute definiert, die nur für diesen speziellen Alarmtyp relevant sind. Das ist ein `timeToAnswer` Attribut, welches eine Zeitspanne angibt, in der User auf den Alarm Rückmeldung geben können und das `location` Attribut, das einen Standort angibt. Die abstrakte Klasse `AlertCommand` ist die Basisklasse für alle RPCs in diesem Paket. Die einzige Implementierung ist die Klasse `RpcOpenAlerts`. Damit können alle offenen Alarmer abgefragt werden. Da User auf einen empfangenen Alarm Rückmeldung geben können, gibt es die abstrakte Klasse `AlertResponse`. Anstatt eines `Event` als `request` Attribut, kann hier nur ein `Alert` angegeben werden. Sie ist die Basisklasse für `Acknowledgment` und `Close`. Die Attribute `accepted` und `closed` geben an, ob der Alarm akzeptiert, bzw. geschlossen wird.

Objekte dieser Klassen werden zur Übertragung in das JSON Format serialisiert. Bei der Deserialisierung dieser Daten stößt man auf ein Problem, das als **Polymorphic Deserialization** bezeichnet wird [Fas11]. Möchte man einen JSON String der abstrakten `Alert` Klasse deserialisieren, muss bekannt sein, um welche Implementierung der abstrakten Klasse es sich konkret handelt. Dafür wird dem verwendeten JSON Prozessor **Jackson** mitgeteilt, dass er die Information über die Implementierung aus dem `type` Attribut entnehmen kann. Ist dieses auf 'CORONARY' gesetzt, wird versucht den JSON String zu einem Objekt der Klasse `Coronary` zu deserialisieren. Dieses Verhalten des JSON Prozessors wird mit Java Annotations definiert, wie in Listing 1 zu sehen ist.

```
1 @JsonTypeInfo(use = JsonTypeInfo.Id.NAME, include = JsonTypeInfo.As.  
2   PROPERTY, property = "type", visible = true)  
3 @JsonSubTypes({  
4     @JsonSubTypes.Type(value = Coronary.class, name = "CORONARY"),  
5     @JsonSubTypes.Type(value = Fire.class, name = "FIRE")})  
6 public abstract class Alert extends Event {  
7     // ...  
8 }
```

Listing 1: Jackson Polymorphic Deserialization

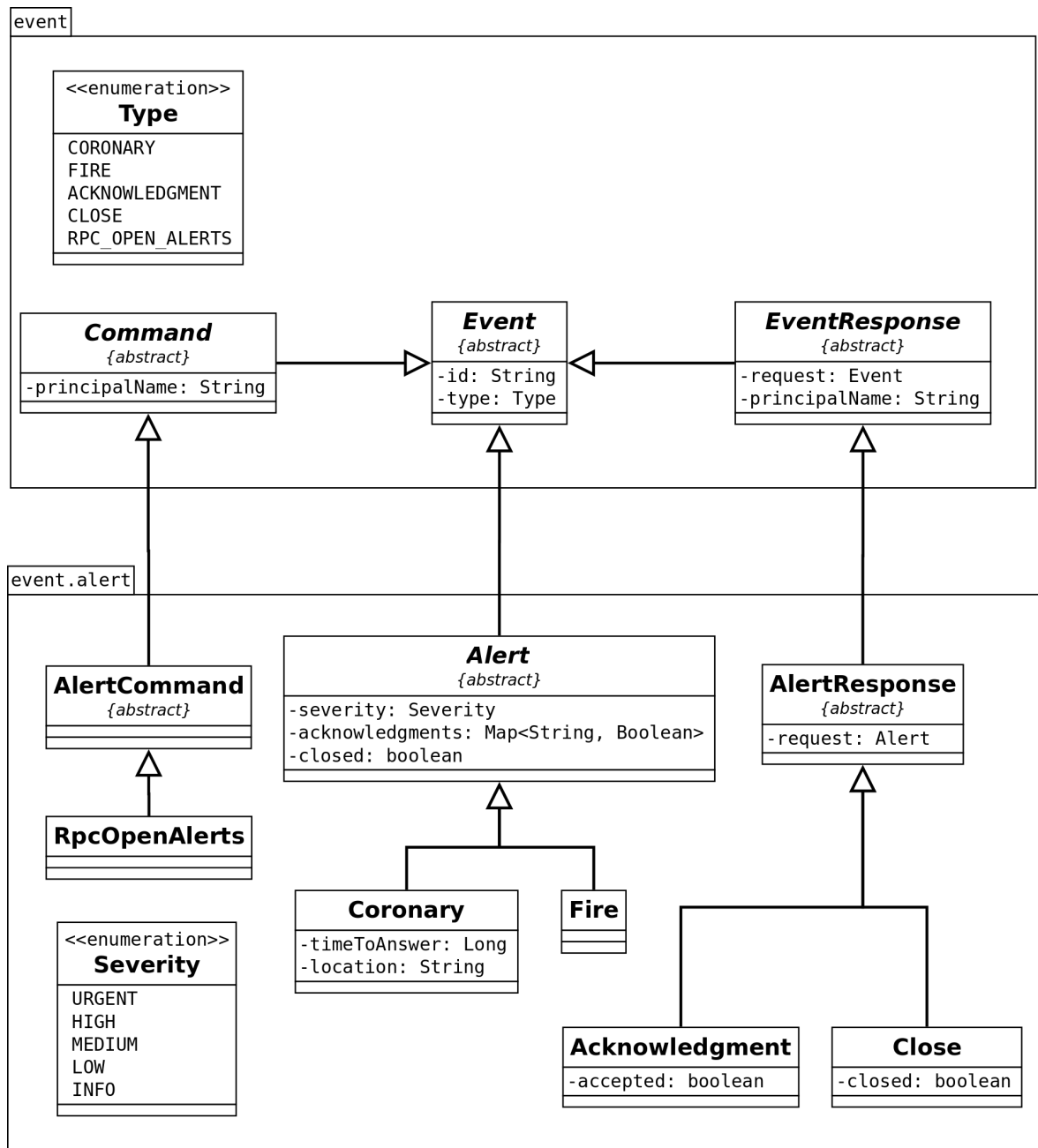


Abbildung 25: Prototyp Datenmodell

5.4 REST Gateway

Über das REST Gateway können neue Alarmer erstellt werden. Dafür wird ein REST Endpoint zur Verfügung gestellt. Dieser wird mit dem Spring Cloud Stream Framework implementiert und ist über den Port 8081 und den Pfad **v1/alert** erreichbar. Der Zugriff wird mit einer HTTP Basic Authentication geschützt. Die übertragenen Daten müssen einen, in das JSON Format serialisierter, **Alert** sein. Der Quellcode des REST Endpoints ist in Listing 2 dargestellt.

Ein so übertragener **Alert** wird in das **event.alert.new** Exchange geschrieben. Es werden dabei keine AMQP spezifischen Funktionalitäten benötigt. Microservices sollen

```
1 @RestController
2 @RequestMapping(value = "v1/alert", consumes = "application/json")
3 public class AlertController {
4
5     @Autowired
6     private AlertService alertService;
7
8     @PreAuthorize("hasRole('ALERT_CREATE')")
9     @PutMapping
10    public void put(@RequestBody Alert alert) {
11        alertService.publish(alert);
12    }
13 }
```

Listing 2: REST Controller

über ein pub/sub Modell kommunizieren. Aus diesem Grund wird an dieser Stelle das Spring Cloud Stream Framework verwendet. Das Versenden von Nachrichten erfolgt über das `AlertGateway`, zu sehen in Listing 3. Über dieses können in der Applikation Nachrichten versendet werden.

```
1 @MessagingGateway
2 public interface AlertGateway {
3
4     @Gateway(requestChannel = Source.OUTPUT)
5     void publish(Alert alert);
6 }
```

Listing 3: Spring Cloud Stream Gateway

In einer YAML Datei wird die Applikation konfiguriert. Für das Versenden von Nachrichten muss lediglich die `destination` und der `content-type` eingestellt werden. Das Exchange `event.alert.new` wird bei Bedarf automatisch erstellt. Das `AlertGateway` sendet Nachrichten dadurch an die eingestellte Adresse und im eingestellten Format. Die Konfigurationsdatei des REST Gateways ist in Listing 4 dargestellt.

```
1 server:
2   port: 8081
3 spring:
4   cloud:
5     stream:
6       bindings:
7         output:
8           destination: event.alert.new
9           content-type: application/json
```

Listing 4: Spring Cloud Stream Konfiguration

Neue Alarmer sollen über Gateway Applikationen erstellt werden. Dadurch ist gewährleistet, dass das System flexibel bleibt. Alarmquellen können so zur Laufzeit, über Gateway Applikation, hinzugefügt oder entfernt werden. Durch den Einsatz des Spring

Cloud Stream Frameworks konnte der Quellcode der Applikation sehr einfach gehalten werden.

5.5 Alarmierungsapplikation

Die Alarmierungsapplikation besteht aus fünf Komponenten. Diese sind in Abbildung 26 dargestellt.

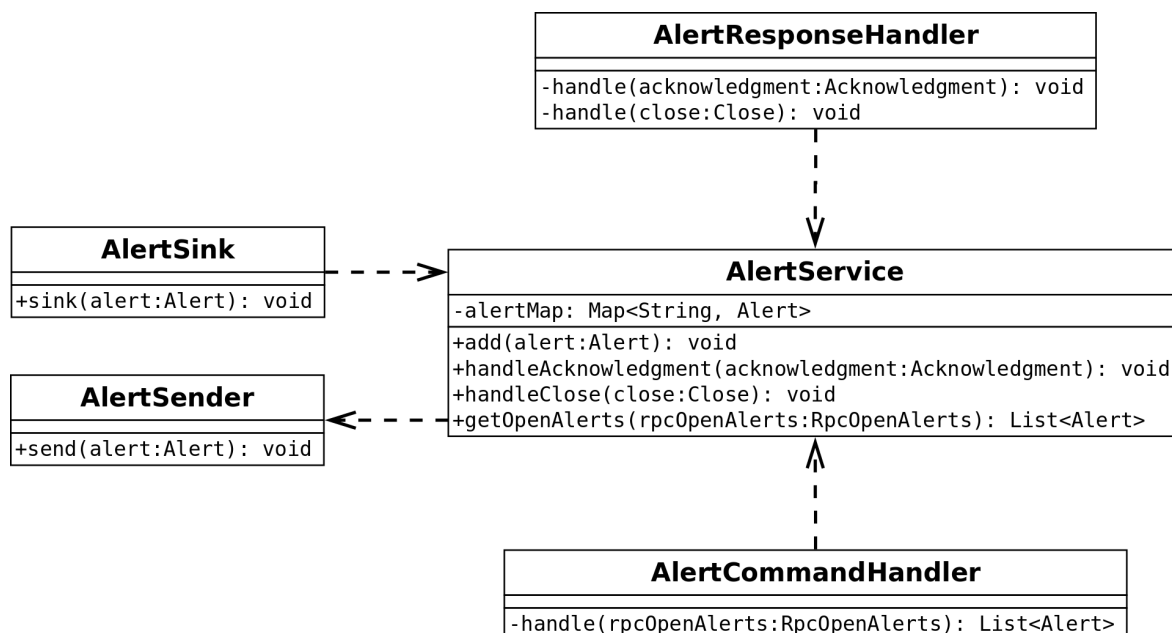


Abbildung 26: Das Alert Modul als UML Diagram

Die zentrale Komponente ist der **AlertService**. Dieser verwaltet alle im System vorhandenen Alarme. Dafür wird jedem neuen Alarm eine UUID als pseudo-Id zugewiesen und er wird *In-Memory* gespeichert. Dafür wird eine **ConcurrentHashMap** verwendet. Er verarbeitet alle empfangenen Nachrichten und veranlasst das Versenden von Alarmen an die User.

Neue Alarme werden über den **AlertSink** empfangen. Umgesetzt wurde das mit dem Spring Cloud Stream Framework, ähnlich wie in Kapitel 5.4. Bei diesem Framework werden Komponenten, die lediglich eine Nachricht empfangen, als *Sink* bezeichnet. Empfangene Alarme werden dem **AlertService** übergeben.

Der **AlertResponseHandler** wurde mit dem Spring AMQP Framework umgesetzt. Er empfängt Nachrichten aus der `event.response.alert` Queue. Das wird mit der **RabbitListener** Annotation angegeben. Es wird eine Queue und kein Exchange verwendet, da die Rückmeldungen von einem einzigen Consumer empfangen werden sollen. Es können hier zwei verschiedene Typen von Events empfangen werden. Diese werden in unterschiedlichen Methoden verarbeitet, die mit der **RabbitHandler** Annotation

versehen sind, wie in Listing 5 zu sehen ist. Der `AlertService` ist verantwortlich für die Verarbeitung der empfangenen Events.

```
1 @Component
2 @RabbitListener(queues = "event.response.alert")
3 public class AlertResponseHandler {
4
5     @Autowired
6     private AlertService alertService;
7
8     @RabbitHandler
9     private void handle(Acknowledgment acknowledgment) {
10         alertService.handleAcknowledgment(acknowledgment);
11     }
12
13     @RabbitHandler
14     private void handle(Close close) {
15         alertService.handleClose(close);
16     }
17 }
```

Listing 5: Nachrichten empfangen mit Spring AMQP

Ähnlich wie der `AlertResponseHandler`, ist auch der `AlertCommandHandler` mit dem Spring AMQP Framework umgesetzt worden. Diese Komponente empfängt Nachrichten aus der `event.command.alert` Queue. In der Methode `handle` werden die eingehenden RPCs verarbeitet. Vom Framework wird automatisch der Rückgabewert der Methode, in die Queue aus dem `reply_to` Argument einer eingehenden Nachricht geschrieben. RPCs können so sehr einfach verarbeitet werden.

```
1 @Component
2 @RabbitListener(queues = "event.command.alert")
3 public class AlertCommandHandler {
4
5     @Autowired
6     private AlertService alertService;
7
8     @RabbitHandler
9     private List<Alert> handle(RpcOpenAlerts rpcOpenAlerts) {
10         return alertService.getOpenAlerts(rpcOpenAlerts);
11     }
12 }
```

Listing 6: RPC mit Spring AMQP

Der `AlertSender` verschickt die Alarmnachrichten an die Clients. Dafür werden sie mit einem speziellen Routing Key in das `event.alert` Exchange geschrieben. Zum Versenden wird hier das `RabbitTemplate` von Spring AMQP verwendet, zu sehen in Listing 7.

Mit dem Spring AMQP Framework werden auch die Alarm Exchanges und die Bindings zwischen ihnen erstellt. In Listing 8 wird gezeigt, wie mit Spring AMQP das


```
1 @Component
2 public class AlertSender {
3
4     @Autowired
5     private RabbitTemplate rabbitTemplate;
6
7     @Autowired
8     private Exchange alertExchange;
9
10    public void send(Alert alert) {
11        rabbitTemplate.convertAndSend(alertExchange.getName(),
12            buildRoutingKey(alert), alert);
13    }
14
15    private String buildRoutingKey(Alert alert) {
16        return String.format("event.alert.%s.%s", alert.getType().name
17            (), alert.getSeverity().name());
18    }
19 }
```

Listing 7: Nachrichten versenden mit Spring AMQP

`event.alert.fire` Exchange erstellt und mit dem `event.alert` Exchange über ein Binding verbunden wird. Als Routing Key wird beim Binding `event.alert.FIRE.#` verwendet, was dafür sorgt, dass alle Alarme des Typs `FIRE` in das neue Exchange geroutet werden. Beim Starten der Applikation, werden die hier definierten Strukturen automatisch in dem Broker erstellt.

```
1 @Configuration
2 public class MessagingConfiguration {
3
4     // ...
5
6     @Bean
7     public Exchange fireExchange() {
8         return ExchangeBuilder.topicExchange("event.alert.fire")
9             .durable()
10             .build();
11     }
12
13     @Bean
14     public Binding fireBinding(Exchange fireExchange,
15         Exchange alertExchange) {
16         return BindingBuilder.bind(fireExchange)
17             .to(alertExchange)
18             .with("event.alert.FIRE.#")
19             .noargs();
20     }
21     // ...
22 }
```

Listing 8: Exchanges und Bindings erstellen mit Spring AMQP

5.6 User Applikation

Die User Applikation besteht aus einer Webanwendung und einer Serveranwendung, die standardmäßig über den Port 8080 erreichbar sind. Die Serveranwendung stellt einen WebSocket Endpoint über den Pfad `/ws` zur Verfügung. Über diesen kann die Webanwendung mit dem System interagieren. Über eine rollen-basierte Zugriffskontrolle werden die verfügbaren STOMP Destinations geschützt. In Tabelle 3 sind alle Destinations aufgelistet, mit der Information, welches STOMP Kommando darauf angewendet werden kann und welche Rollen dafür autorisiert sind.

Destination	Kommando	Rolle
<code>/user/queue/errors</code>	SUBSCRIBE	USER
<code>/exchange/event.alert/#</code>	SUBSCRIBE	ADMIN
<code>/exchange/event.alert.fire/#</code>	SUBSCRIBE	FIREMAN
<code>/exchange/event.alert.coronary/#</code>	SUBSCRIBE	DOCTOR
<code>/app/alerts.open</code>	SUBSCRIBE	USER
<code>/app/event.response.alert</code>	SEND	USER

Tabelle 3: STOMP Zugriffskontrolle

In der folgenden Aufzählung werden die Funktionen der Destinations erläutert:

- `/user/queue/errors`: Empfangen von persönlichen Fehlermeldungen.
- `/exchange/event.alert/#`: Empfangen von allen Alarmen.
- `/exchange/event.alert.fire/#`: Empfangen von allen Alarmen des Typs FIRE
- `/exchange/event.alert.coronary/#`: Empfangen von allen Alarmen des Typs CORONARY.
- `/app/alerts.open`: Empfangen von allen offenen Alarmen.
- `/app/event.response.alert`: Senden einer `Acknowledgment` oder `Close` Nachricht.

Die Nachrichten an Destinations mit dem `/app` Präfix werden nicht an den Broker weitergeleitet. Sie werden von dem, in Listing 9 dargestellten, `WebSocketController` verarbeitet. In der Methode `handleResponse` werden die an die Destination `/app/event.response.alert` versendeten `AlertResponses` verarbeitet. Über den Parameter `principal` kann auf den Namen des Users zugegriffen werden, der die Nachricht versendet hat. Dieser Name wird in die `AlertResponse` eingetragen und diese wiederum wird in das `event.response.alert` geschrieben. Damit wird sichergestellt, dass keine Nachricht unter einem falschen Namen verschickt werden kann. Das Versenden von Rückmeldungen und RPCs wird mit dem Spring AMQP Framework durchgeführt.

```

1 @Controller
2 public class WebSocketController {
3
4     // ...
5     @Autowired
6     private AlertResponseService alertResponseService;
7
8     @RequestMapping("/event.response.alert")
9     public void handleResponse(AlertResponse response,
10                               Principal principal) {
11         alertResponseService.execute(response, principal);
12     }
13     // ...
14 }

```

Listing 9: Websocket Controller

Die Webanwendung ist eine Single-Page-Webanwendung, die mit dem AngularJS Framework erstellt wurde. Die Oberfläche ist in Abbildung 27 zu sehen. Als erstes werden alle vorhandenen Alarme abgerufen und in einer einfachen Liste angezeigt. Diese Liste wird fortlaufend über eintreffende Alarm Nachrichten aktualisiert. Über zwei Buttons können sie entweder akzeptiert oder zurückgewiesen werden. Daraufhin wird eine `AlertResponse` versendet. Alarme können nach dem Akzeptieren geschlossen werden.

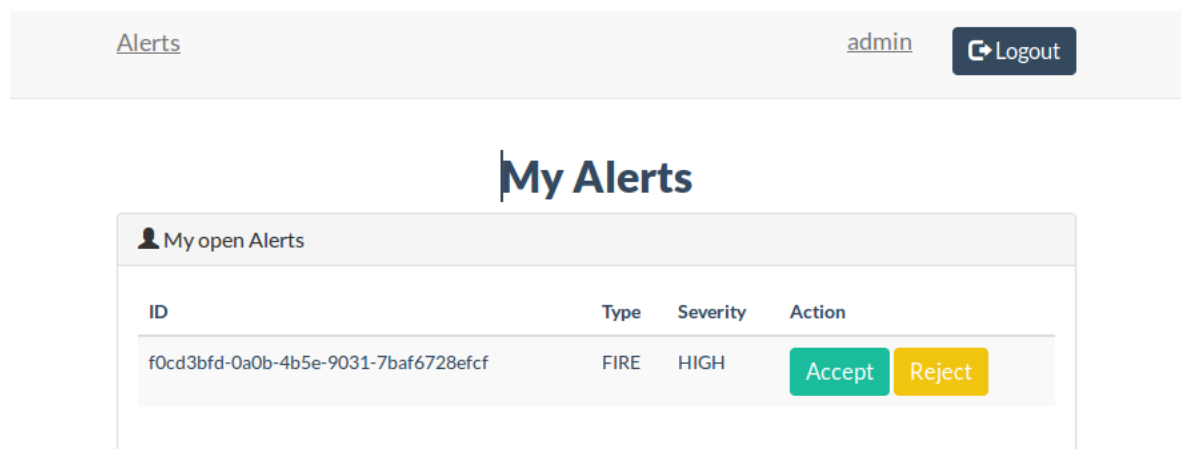
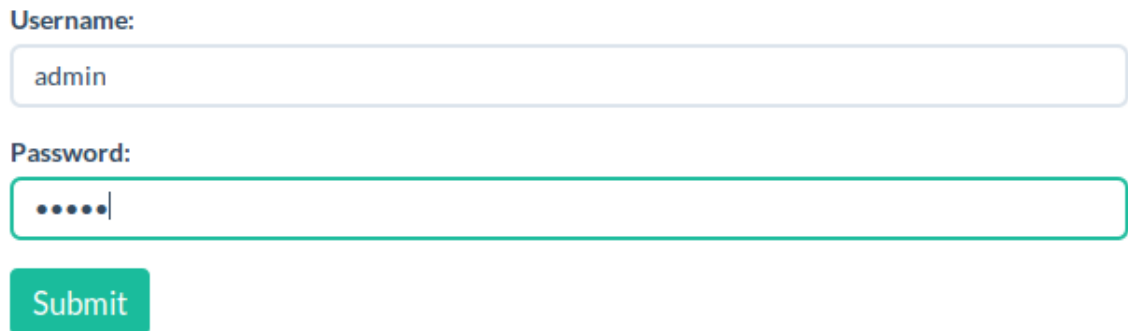


Abbildung 27: Webanwendung

5.7 Authentifizierungs Applikation

Zur Authentifizierung von Usern wird ein OAuth2 Provider verwendet, der mit dem Spring Security Framework erstellt wurde. Er hat die Aufgabe, für die User **admin**, **doctor** und **fireman** eine Authentifizierung und rollen-basierte Autorisierung bereitzustellen. Nicht authentifizierte Benutzer werden von der Webanwendung zum OAuth2 Provider weitergeleitet, der über den Port 9999 erreichbar ist. Dort müssen sie sich an-

melden und anschließend der Webanwendung, den Zugriff auf geschützte Ressourcen gewähren. Diese Schritte sind in Abbildung 28 und Abbildung 29 dargestellt.



Username:

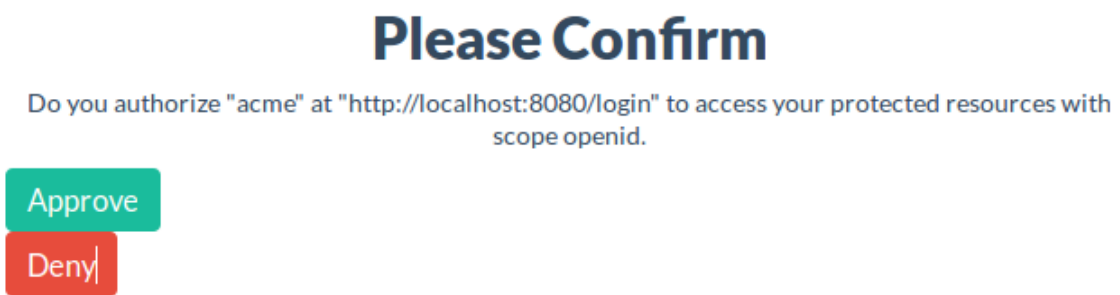
admin

Password:

.....

Submit

Abbildung 28: OAuth2 Login



Please Confirm

Do you authorize "acme" at "http://localhost:8080/login" to access your protected resources with scope openid.

Approve

Deny

Abbildung 29: OAuth2 Autorisierung

5.8 Fehlende Funktionalitäten

Folgende Funktionalitäten konnten aus zeitlichen Gründen nicht mehr im Prototyp umgesetzt werden:

- Alarmieren einzelner User, über das PTP Modell von Spring Websocket.
- Das RpcOpenAlerts Kommando soll nur User spezifische Alarmer abrufen. Momentan werden immer alle offenen Alarmer zurückgeliefert.
- Funktionierender Logout in der Webanwendung.

6 Zusammenfassung und Ausblick

Die Aufgabe dieser Arbeit bestand darin, ein Alarmierungssystem zu konzeptionieren und implementieren. Für einen Alarm zuständige Personen, sollen dabei über eine Message Oriented Middleware alarmiert werden.

Eine Analyse ergab, dass mit AMQP und STOMP zwei Protokolle zur Verfügung stehen, auf denen der Prototyp basieren kann. Der ausgesuchte Broker RabbitMQ, unterstützt beide Protokolle und bietet mit einem erweiterten AMQP 0-9-1 Modell sehr gute Routing Möglichkeiten.

Spring Cloud Stream wird verwendet, um Microservices zu erstellen, die Alarmer aus externen Systemen entgegen nehmen. Das Spring AMQP Projekt kann im Prototyp genutzt werden, um ein Routing der Alarm Nachrichten und die Kommunikation mit dem Broker umzusetzen. Dies ermöglicht, dass lediglich, die für einen Alarm zuständigen Personen, benachrichtigt werden. Über das Spring Websocket Projekt und eine Webanwendung können die User mit dem System interagieren.

Abbildungsverzeichnis

1	Message Oriented Middleware	4
2	JMS Queue	5
3	JMS Topic	5
4	AMQP pub/sub	6
5	AMQP PTP	7
6	AMQP Fanout Exchange	7
7	AMQP Direct Exchange	8
8	AMQP Topic Exchange	8
9	AMQP Headers Exchange	9
10	Apache Kafka Modell	9
11	Trend ActiveMQ und RabbitMQ	13
12	RabbitMQ Exchange to Exchange Bindings	15
13	RabbitMQ Sender-selected Distribution	15
14	RabbitMQ RPC	16
15	Spring AMQP	17
16	Spring Integration	18
17	Spring Cloud Stream	19
18	Spring WebSocket	20
19	Konzept Exchange to Exchange Bindings	23
20	Konzept Headers Exchange	24
21	Konzept Sender-selected Distribution	25
22	Konzept Spring Websocket	25
23	Prototyp Architektur	27
24	Projektstruktur	28
25	Prototyp Datenmodell	30
26	Das Alert Modul als UML Diagram	32
27	Webanwendung	36
28	OAuth2 Login	37
29	OAuth2 Autorisierung	37

Tabellenverzeichnis

1	Von Brokern angebotene Schnittstellen und Protokolle	12
2	Client Implementierungen von Schnittstellen und Protokolle	12
3	STOMP Zugriffskontrolle	35

Listingverzeichnis

1	Jackson Polymorphic Deserialization	29
2	REST Controller	31
3	Spring Cloud Stream Gateway	31
4	Spring Cloud Stream Konfiguration	31
5	Nachrichten empfangen mit Spring AMQP	33
6	RPC mit Spring AMQP	33
7	Nachrichten versenden mit Spring AMQP	34
8	Exchanges und Bindings erstellen mit Spring AMQP	34
9	Websocket Controller	36

Abkürzungsverzeichnis

Ajax	asynchronous JavaScript and XML
AMQP	Advanced Message Queuing Protocol
API	application programming interface
BCC	Blind Carbon Copy
C	Consumer
CC	Carbon Copy
Java EE	Java Platform, Enterprise Edition
JMS	Java Message Service
JSON	JavaScript Object Notation
M2M	machine-to-machine
MOM	Message Oriented Middleware
MQTT	Message Queue Telemetry Transport
P	Publisher
PTP	point-to-point
pub/sub	publish/subscribe
REST	Representational state transfer
RPC	Remote Procedure Call
STOMP	Simple/Streaming Text Orientated Messaging Protocol
TTL	Time to Live
UUID	Universally Unique Identifier
X	Exchange

Literaturverzeichnis

- [Apa15] APACHE SOFTWARE FOUNDATION: *How does ActiveMQ compare to Artemis*. <http://activemq.apache.org/how-does-activemq-compare-to-artemis.html>, 2015. – [Abrufdatum: 09.11.2016]
- [Apa16a] APACHE KAFKA: *Apache Kafka Documentation*. https://kafka.apache.org/0100/documentation.html#intro_consumers, 2016. – [Abrufdatum: 09.11.2016]
- [Apa16b] APACHE SOFTWARE FOUNDATION: *ActiveMQTM*. <http://activemq.apache.org/>, 2016. – [Abrufdatum: 09.11.2016]
- [Apa16c] APACHE SOFTWARE FOUNDATION: *Spring Support*. <http://activemq.apache.org/spring-support.html>, 2016. – [Abrufdatum: 09.11.2016]
- [Che06] CHEN, Nicholas: *Convention over Configuration*. http://softwareengineering.vazexqi.com/files/convention_over_configuration.pdf, 2006. – [Abrufdatum: 08.11.2016]
- [Cur04] CURRY, Edward: *Message Oriented Middleware*. http://www.edwardcurry.org/publications/curry_MfC_MOM_04.pdf, 2004. – [Abrufdatum: 08.11.2016]
- [Der16] DERDACK: *Andere Systeme*. <http://www.derdack.com/de/enterprise-alert/integration/andere/>, 2016. – [Abrufdatum: 07.11.2016]
- [Fas11] FASTERXML: *JacksonPolymorphicDeserialization*. <https://github.com/FasterXML/jackson-docs/wiki/JacksonPolymorphicDeserialization>, 2011. – [Abrufdatum: 14.11.2016]
- [Goo16] GOOGLE TRENDS: *Vergleichen*. <https://www.google.de/trends/explore?date=2008-11-09%202016-11-09&q=activemq,rabbitmq>, 2016. – [Abrufdatum: 09.11.2016]
- [HW03] HOHPE, Gregor ; WOOLF, Bobby: *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2003. – ISBN 0133065103
- [JCNE⁺13] JENDROCK, Eric ; CERVERA-NAVARRO, Ricardo ; EVANS, Ian ; GOLLAPUDI, Devika ; HAASE, Kim ; MARKITO, William ; SRIVATHSA, Chinmayee: *The Java EE 6 Tutorial*. <http://docs.oracle.com/javaee/6/tutorial/doc/bncdx.html>, 2013. – [Abrufdatum: 08.11.2016]
- [MQT16] MQTT: *MQTT*. <http://mqtt.org/>, 2016. – [Abrufdatum: 09.11.2016]
- [Pad15] PADRO, Xavier: *Spring Integration Fundamentals*. <https://www.javacodegeeks.com/2015/09/spring-integration-fundamentals.html>, 2015. – [Abrufdatum: 10.11.2016]
- [Pip13] PIPER, Andy: *Choosing Your Messaging Protocol: AMQP, MQTT, or STOMP*. <https://blogs.vmware.com/vfabric/2013/02/choosing-your-messaging-protocol-amqp-mqtt-or-stomp.html>, 2013. – [Abrufdatum: 09.11.2016]

- [Piv07a] PIVOTAL SOFTWARE, INC.: *RabbitMQ STOMP Adapter*. <https://www.rabbitmq.com/stomp.html>, 2007. – [Abrufdatum: 09.11.2016]
- [Piv07b] PIVOTAL SOFTWARE, INC.: *Which protocols does RabbitMQ support?* <https://www.rabbitmq.com/protocols.html>, 2007. – [Abrufdatum: 08.11.2016]
- [Piv16a] PIVOTAL SOFTWARE, INC.: *Protocol Extensions*. <https://www.rabbitmq.com/extensions.html>, 2016. – [Abrufdatum: 10.11.2016]
- [Piv16b] PIVOTAL SOFTWARE, INC.: *Spring AMQP*. <https://projects.spring.io/spring-amqp/>, 2016. – [Abrufdatum: 10.11.2016]
- [Piv16c] PIVOTAL SOFTWARE, INC.: *Spring Boot*. <https://projects.spring.io/spring-boot/>, 2016. – [Abrufdatum: 08.11.2016]
- [Piv16d] PIVOTAL SOFTWARE, INC.: *Spring Cloud*. <https://projects.spring.io/spring-cloud/>, 2016. – [Abrufdatum: 08.11.2016]
- [Piv16e] PIVOTAL SOFTWARE, INC.: *Spring Cloud Stream*. <https://cloud.spring.io/spring-cloud-stream/>, 2016. – [Abrufdatum: 10.11.2016]
- [Piv16f] PIVOTAL SOFTWARE, INC.: *Spring Security*. <https://projects.spring.io/spring-security/>, 2016. – [Abrufdatum: 08.11.2016]
- [Qua16] QUADRI, Moinuddin: *Apache Kafka v/s RabbitMQ - Message Queue Comparison*. <http://www.cloudhack.in/index.php/2016/02/29/apache-kafka-vs-rabbitmq/>, 2016. – [Abrufdatum: 09.11.2016]
- [Rao12] RAO, Jun: *Clients*. <https://cwiki.apache.org/confluence/display/KAFKA/Clients>, 2012. – [Abrufdatum: 09.11.2016]
- [Sto13] STOYANCHEV, Rossen: *WebSocket Apps Spring Framework 4 (part1)*. <http://rstoyanchev.github.io/springx2013-websocket/#91>, 2013. – [Abrufdatum: 10.11.2016]

Anhang

Auf der beigefügten CD befindet sich folgender Inhalt:

- Der Quellcode des entwickelten Prototypen befindet sich im Verzeichnis *prototype_source*. Mit dem Build Tool Maven können ausführbare JAR Dateien der Module erstellt werden. Im Unterverzeichnis *docker* befindet sich eine Docker Compose Datei. Mit ihr kann der benötigte RabbitMQ Broker in einem Docker Container gestartet werden. Drei User können in diesem System verwendet werden: *admin*, *doctor* und *fireman*. Ihr Passwort lautet gleich wie ihr Benutzername.
- Das Verzeichnis *prototype_jar* enthält bereit erstellte JAR Dateien der Prototyp Module.
- Die Datei *postman_collection.json*, welche eine Collection an REST Calls enthält. Sie kann in den Postman REST Client importiert werden, um REST Calls mit dem Prototyp durchzuführen.
- Die Datei *Bachelorarbeit_Philipp_Bobek.pdf*. Sie entspricht diesem Dokument.

