HOCHSCHULE KONSTANZ TECHNIK, WIRTSCHAFT UND GESTALTUNG
UNIVERSITY OF APPLIED SCIENCES

CAD-Project - Mom based Information Live Flow

Paul Drautzburg, Lukas Hansen, Georg Mohr, Kim De Souza, Sebastian Thuemmel, Sascha Drobig

Vorwort Das vorliegende Dokument beschreibt die Umsetzung für das Projekt im Rahmen des MSI-Kurses *Cloud Application Development*.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Motivation

1.1 Zielsetzung

ToDo: Verweis auf 12 Faktor APP Standard !!! Die Tabelle soll am Anfang stehen, am besten in der Zielsetzung. Die folgende Tabelle beschreibt die Kernanforderungen der 12 Faktor APP,

12 Faktor APP Anforderungen		
ID	**Anforderung**	**Beschreibung**
1.	Codebase	Eine im Versionsmanagementsystem verwaltete Codebase, viele Deployments.
2.	Abhängigkeiten	Abhängigkeiten explizit deklarieren und isolieren.
3.	Konfiguration	Die Konfiguration in Umgebungsvariablen ablegen.
4.	Unterstützende Dienste	Unterstützende Dienste als angehängte Ressourcen behandeln.
5.	Build, release, run	Build- und Run-Phase strikt trennen.
6.	Prozesse	Die App als einen oder mehrere Prozesse ausführen.
7.	Bindung an Ports	Dienste durch das Binden von Ports exportieren.
8.	Nebenläufigkeit	Mit dem Prozess-Modell skalieren.
9.	Einweggebrauch	Robuster mit schnellem Start und problemlosen Stopp.
10.	Dev-Prod-Vergleichbarkeit	Entwicklung, Staging und Produktion so ähnlich wie möglich halten.
11.	Logs	Logs als Strom von Ereignissen behandeln.
12.	Admin-Prozesse	Admin/Management-Aufgaben als einmalige Vorgänge behandeln.

Tab. 1: 12 Faktor App Anforderungen

1.2 Die 12 Faktor-APP Anforderungen

2 Einleitung

3 Kommunikation der Komponenten

Die entwickelte Anwendung besteht mit einem Java-Service für die verwendete Wetter-API, der Complex Event Processing Engine und den Anwender-Clients aus drei Komponenten. Diese Komponenten müssen möglichst stark entkoppelt miteinander kommunizieren können. Durch eine starke Entkopplung wird erreicht ,dass die jeweiligen Komponenten keine Kenntnisse über vorhandene Schnittstellen oder die verwendete Programmiersprache besitzen müssen. Um dies zu realisieren, wird RabbitMQ als Message-oriented Middleware (MoM) eingesetzt.

3.1 RabbitMQ

Bei RabbitMQ handelt es sich um einen auf Erlang basierenden OpenSource Message Broker, welcher Bibliotheken für alle gängigen Programmiersprachen wie Java, JavaScript, Swift und C# anbietet. Dadurch wird die Kommunikation mit Android-, iOS- und Webapplikationen möglich. Durch die Verwendung von Queues und Topics wird die asynchrone Verteilung der Nachrichten ermöglicht. RabbitMQ verwendet als Standard das Messaging Protokoll AMQP, bietet aber Plugins für alternative Protokolle wie MQTT und STOMP. Da auch mobile Geräte zu den eingesetzten Komponenten gehören, wird das Protokoll MQTT eingesetzt, da dieses speziell für den Einsatz in Mobilgeräten entwickelt wurde. Die Kommunikation der einzelnen Komponenten erfolgt mit MQTT über Topics. Damit der Nachrichtenaustausch stattfinden kann, müssen sich die miteinander kommunizierenden Komponenten auf ein oder mehrere gemeinsame Topics einigen. Der Aufbau eines Topics ist mit REST-Schnittstellen vergleichbar und kann aus mehreren Topic-Leveln bestehen. Zusätzlich können beim Abonnement von Topics Platzhalter wie + und # eingesetzt werden. Diese funktionieren wie reguläre Ausdrücke und ersetzen im Falle des Platzhalters + eine einzelne Topic-Ebene und beim Platzhalter # alle nachfolgenden Ebenen. Die Topics und mögliche Abonnements dieser Anwendung sind nachfolgend aufgelistet:

78467/today
: Abonnement des Wetters von Postleitzahl 78467 des heutigen Tages

+/today
: Abonnement des Wetters aller verfügbaren Postleitzahlen des heutigen Tages

78467/today/alert
: Abonnement der Wetterwarnungen für die Postleitzahl 78467

+/weekly
: Abonnement der Vorhersage der nächsten Woche aller verfügbaren Postleitzahlen

#
: Abonnement aller verfügbaren Topics

Damit Daten durch einen Client versendet oder empfangen werden kann, muss er sich beim Verbindungsaufbau authentifizieren und für den Zugriff auf das entsprechende Topic autorisiert sein. Die Authentifizierung erfolgt über eine gewöhnliche Benutzername / Passwort - Abfrage. Um den Zugriff auf MQTT-Topics zu beschränken, ermöglicht RabbitMQ die Verwendung virtueller Hosts (vHosts). Durch diese erlangen die Nutzer nur Zugriff auf ein Topic, wenn sie für den vHost des Publishers autorisiert sind. Die Erstellung neuer Nutzer und die Verwaltung der Rechte erfolgt über unter ande-

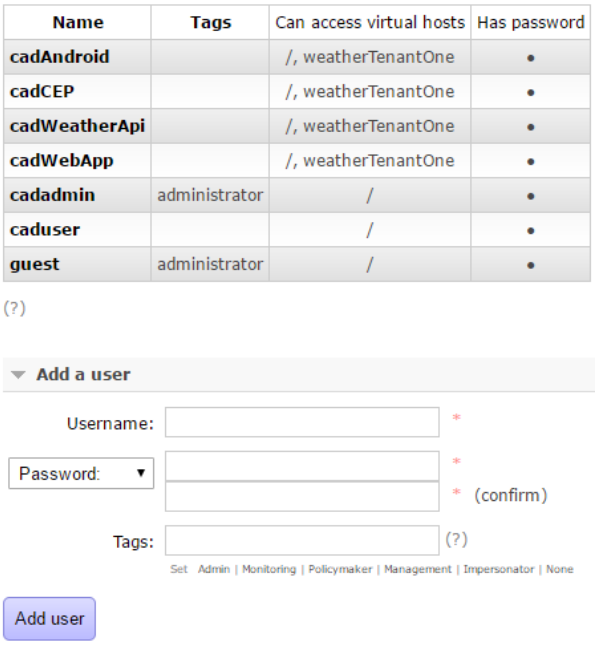

Abbildung 1: Administrationsmenü zur Benutzererstellung

rem über das Management Plugin. In Abb. 1 ist ersichtlich, dass die User *cadAndroid*, *cadCEP*, *cadWeatherApi* und *cadWebApp* Zugriff auf den gemeinsamen vHost weatherTenantOne haben. Durch dieses Verfahren kann das gleiche Topic von mehreren Nutzern mit unterschiedlichen vHosts verwendet werden, ohne das sie die Nachrichten anderer vHosts des gleichen Topics lesen können. Eine Übersicht über die einzelnen

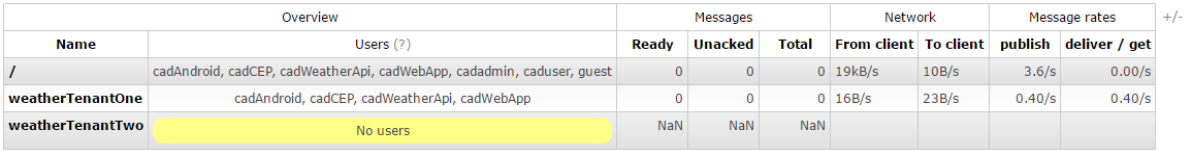

Abbildung 2: Übersicht über die virtuellen Hosts

vHosts und die berechtigten Nutzer wird in Abb. 2 dargestellt. Diese zeigt noch einmal die erwähnte Zugriffsbeschränkung auf die vier Nutzer dieses Use-Cases sowie den aktuell verursachten Datentransfer der vHosts. Auf diese Weise erfüllt die Anwendung

die Anforderung der Multi-Tenancy. Zusätzlich zum Management Plugin bietet RabbitMQ eine HTTP-Schnittstelle. Diese ermöglichen es dem Administrator zum einen über eine Kommandozeile in Verbindung mit Kommandozeilenprogrammen wie cURL (Client for URLs) die angebotenen Schnittstellen aufzurufen und dadurch unter anderem Nutzer anzulegen oder die Verbindungsraten der vHosts auszugeben und auszuwerten (vgl. https://pulse.mozilla.org/api/). Aufgrund der vorhandenen Schnittstellen bietet sich dem Entwickler die Möglichkeit, die Administration über eine eigene Applikation durchzuführen.

4 Wetter-API(Datenquelle)

4.1 Die API

iashfuisdhfuahfukdhskjfsdhukfsjhsjdhfjkshkjfhsdkjh

4.2 Die Schnittstelle

4.3 Anforderungen

Validierung nach ”12 Faktor APP”			
ID	**Anforderung**	**Validierungs Element**	**Erfüllt**
1.	Codebase	....	Nein
2.	Abhängigkeiten	....	Nein
3.	Konfiguration	....	Nein
4.	Unterstützende Dienste	....	Nein
5.	Build, release, run	....	Nein
6.	Prozesse	....	Nein
7.	Bindung an Ports	....	Nein
8.	Nebenläufigkeit	....	Nein
9.	Einweggebrauch	....	Nein
10.	Dev-Prod-Vergleichbarkeit	....	Nein
11.	Logs	....	Nein
12.	Admin-Prozesse	....	Nein

Tab. 2: Validierung nach ”12 Faktor APP”

ToDo: Verweis auf 12 Faktor APP Standard !!!

5 Datenverarbeitung

5.1 Datenbank

6 Complex Event Process

6.1 Zielsetzung

Das Ziel der Anwendung ist es, die Daten welche an die MoM gesendet wurden auszulesen und dann weiterzuverabeiten. Dabei wird mithilfe einer Complex Event Process Engine dann nach besonderen Ereignissen gesucht. Sollte ein solches eintreffen wird eine Warnung an alle Clients über ein besonderes Topic gesendet. Die Anwendung soll sich dabei ein die 12 Faktoren einer Cloud Anwendung halten. Ein weiteres Ziel ist, dass die Anwendung in jeder Cloud laufen kann und nicht abhängig von einer bestimmten Cloud ist.

6.2 Umsetzungsentscheidungen

Als Complex Process Engine wurde Esper ausgewählt. Da Esper eine Java API besitzt und die MOM ebenenfalls über eine Javaschnittstelle verfügt (Siehe Kapitel 3.1) kann Esper leicht mit den anderen Komponenten der Anwendung verbunden werden. Esper besitzt zudem eine ausführliche Dokumentation, was die Umsetzung deutlich erleichtert hat. Es gibt neben Esper andere Engines welche ebenfalls mit Java verbunden werden können. Eine davon ist Apache Spark. Spark wird von Amazon direkt unterstüzt und ist eine eigene Engine welche nur noch mit Statements befüllt werden muss angeboten. Wir haben uns aber gegen Spark entschieden, da es zum einen uns stärker an die Amazon Cloud binden würde und zum anderen die Amazon Spark Engine sehr viel abnimmt und so Kontrolle über das Verhalten der Anwendung nimmt.
Die Anwendung wurde mit Java geschrieben, da es mit Java leicht ist die verschiedenen Komponenten der Anwendung zu verbinden. Java kann überall laufen, sofern eine JVM vorhanden ist. Dadurch haben wir sichergestellt, dass die Anwendung auf jeder Cloud laufen kann. Zusätzlich ist Java die Programmiersprache in der unsere Gruppe die meiste Erfahrung besitzt.

6.3 Esper

Esper ist eine CEP Engine welche von EsperTech entwickelt wurde. Esper ist eine open source Anwendung. Eine Kommerzielle Lizenz wird nur benötigt, wenn die Anwendung weiterverkauft werden soll. Esper arbeitet mit Statements, Listener und Consumption Modes. Esper speichert die eingehenden Events in einer eigenen Datenbank ab und löscht diese von selbst wenn sie nicht mehr benötigt werden. Espers Datenbank wird beim abschalten der Anwendung gelöscht. Esper ist in der Lage verschiedene unabhängige Engines gleichzeitig Laufen zu lassen.
Die Statements basieren auf SQL. Es ist möglich Statements mit Listener zu verknüpfen. Zusätzlich wird der Consumption Mode über die Statements angegeben. Statements

können auch verwendet werden um einen Kontext anzugeben. Das wird aber in dieser Anwendung nicht benötigt, daher wird darauf nicht näher eingegangen. In einen Statement wird immer angegeben was genau gesucht wird, woher es kommen soll und was für Filterbedingungen vorhanden sind. Dabei kann in der from-Klausel auch ein Muster stehen. So kann Esper für die Mustererkennung verwendet werden. Die Statements werden auf jedes Event im Eventstrom angewendet. Sollte ein Statement zutreffen wird der dazugehörige Listener aufgerufen. Dieser kann auf die ausgewählten Werte zugreifen. Der Listner kann selbst eigene Events absenden. Der Consumption Mode legt fest ob ein weiterer Listener auf das Event zugreifen kann. Es kann auch festgelegt werden ob ein Längen oder Zeitfenster die Zahl der zuberücksichtigen Events einschränlen soll.

6.4 Umsetzung

Die Anwendung wird von der InformationLifeFlow Klasse gesteuert. Diese stellt die Main-klasse der Anwendung dar und ließt beim Start die Postlleitzahlen der Städte aus. Danach wird eine Verbindung zu einer MoM erstellt. Für jede Postleitzahl wird ein sogennanter MoMReader erstellt. Diese sind eigene Thread welche für ein bestimmtes Topic auf eingehende Nachrichten warted. Der MoMSender ist das Gegenstück dazu. Der Sender sendet Nachrichten zurück zur MoM damit diese von Clients gelesen werden können. Das Klassendiagramm zeigt die Methoden von der Mainklasse und dem Reader. Die MoM wird mithilfe einer Factory Klasse erstellt. Diese erstellt beim ersten

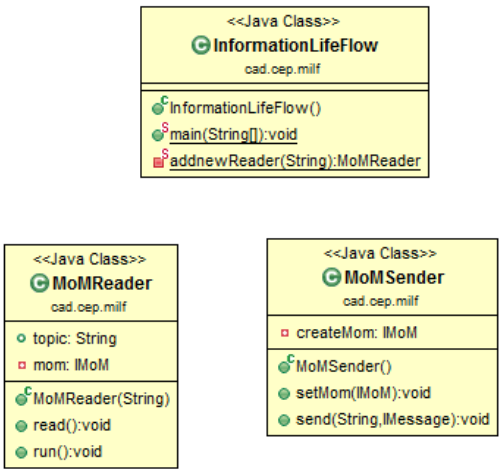

Abbildung 3: Klassendiagramm der Steuerungsklassen

Aufruf eine MoM und gibt diese bei jeden Aufruf wieder zurück. Derzeit unterstüzt die Anwendung nur eine MQTTMoM. Mithilfe des IMoM interfaces kann aber jederzeit eine weitere MoM hinzugefügt werden. Die MQTT MoM benötigt einen Hostpfad, einen Usernamen und ein Passwort, Diese werden von der Factory aus den Umgebungsvariablen ausgelesen und dann der MoM weitergegeben. Die MoM öffenet eine Verbindung ist

dannach in der Lage Nachrichten zu senden oder zu Empfangen. Jede Empfangene Nachricht wird von einer CallBack-Methode ausgewertet. Diese überprüft was für ein Typ die Nachricht besitzt und behandelt diese dann entsprechenend. Das bedeutet ein Forcast wird umgeformt und weitergeleitet während einer Tagesmeldung weiter an die CEP Engine gesendet wird. Beim Umformen des Forcasts wird auch die maximale Temperatur, die minimale Temperatur und das durchschnittswetter für jeden Tag ermittelt. Das folgende Klassendiagramm zeigt die Komponenten der MoMverbindung. Die Anwendung

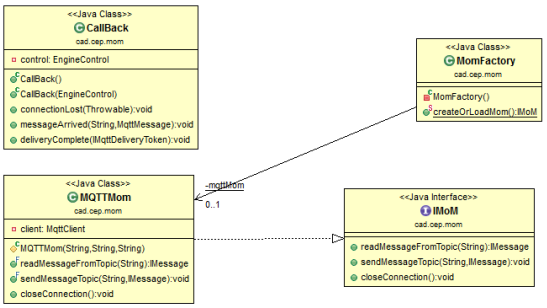

Abbildung 4: Klassendiagramm der MoM Komponenten

kennt verschiedene Nachrichtentypen. Die Nachrichten die eingehen sind entweder eine Nachricht vom Typ "JSONMessage" oder von Typ "WeejkyForcast". Die JSOnMessage stellt dabei die Nachricht für den aktuellen Wetterstand dar. Das sind die häufigsten Nachrichten die eingehen. Wie der Name schon andeutet kommen die Nachrichten in einen JSON-Format an das direkt mithilfe von GSON in eine Instanz der Klasse umgeformt wird. Die WeeklyForcast Nachrichten kommen zwar auch in einen JSON-Format an müssen aber nochmal umgewandelt und gefiltert werden, bis die benötigte Nachricht entsteht. Diese Nachrichten kommen deutlich seltener bei der Anwendung an da diese einen Abschätzung der nächsten Tage darstellt. Die JSONMessage wird zur weiteren Auswertung an die CEP Engine übergeben. Der Wochenbericht wird direkt weiterverarbeitet. Im Folgenden Klassendiagramm werden die Nachrichten mit ihren Methoden gezeigt. Sollte die CEP Engine etwas finden wird ein Alert gesendet. Diese Nachrichten

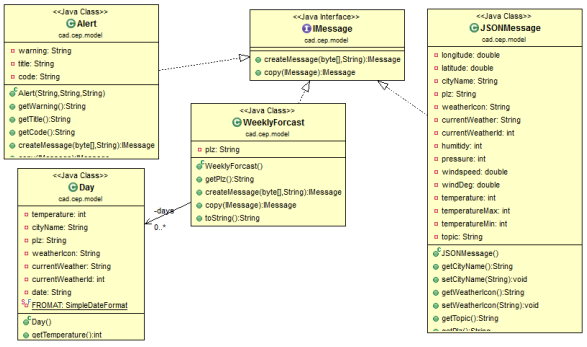

Abbildung 5: Das Klassendiagramm der Nachrichten

beinhalten einen Titel, eine Warnung und einen Warnungscode. Der Title kann verwendet werden um rauszufinden worüber gewarnt wird, die Warnung selbst ist der englische default Text und der Code kann verwendet werden um die Warnung in eine andere Sprache zu übersetzen. Die Folgende Tabelle Zeigt die Warnungcodes. Die CEP Engine wird

Die Wanung-Codes	
Code	**Bedeutung**
T1	Tropisches Wetter: Luftfeuchtigkeit über 90
W1	Forstchance: Es kann sein das es Frost auf der Straße gibt
W2	Starke Kälte: Die Temperatur ist unter -9 Grad
W3	Starker Schneefall: Besonderst heftiger Schneefall
S1	Gutes Wetter: Es scheint die Sonne und es hat über 25 Grad
H1	Herzinfakt Warnung: Für Herzkranke starke Luftdruck schwankungen
H2	Herzinfakt Warnung: Für Herzkranke zu hohe Temperatur (¿=25)

Tab. 3: Die Wanung-Codes

über die Klasse Engine Control gesteuert. Diese Singleton-Klasse geht sicher das die anderen Klassen zugriff auf von ihnen benötigten Fuktionen der Eingine haben aber dabei nicht mehr zugriff als Notwendig erhaltent. Die Klasse ist ein Singleton um sicherzugehen das jede Klasse auf die gleiche Engine zugreift. Die Klasse selbst erstellt eine Instanz der Klasse EsperService. Die Instanz wrid über eine Factory erstellt. Die Factory geht sicher das jedes benötigte Statement mit einen Listener verknüpft wird. Die EsperService-Klasse beinhaltet die tatsächliche Esper Engine und stellt einen Wrapper für diese dar. Das folgende Klassendiagramm zeigt die Methoden und die Abhängigkeiten der Klassen. Die Esper Statements sorgen dafür das Muster in den eingehenden Wetterdaten erkannt werden und reagieren mit passenden Nachrichten auf diese. Die Statements werden von der CEPFactory der Engine hinzugefügt. Jede eingehende Nachricht wird von midestens einen Statement erfasst. Der Entsprechende Listener sorgt dafür das die Clients die Daten erhalten und gibt eine Ausgabe aus. Dabei gibt es kein Fenster oder Consumption Mode. Jede Nachricht wird von diesen Statement erfasst und es dürfen auch weitere Statements greifen. Das Statement sieht wie folgt aus:

```
select * from JSONMessage
```

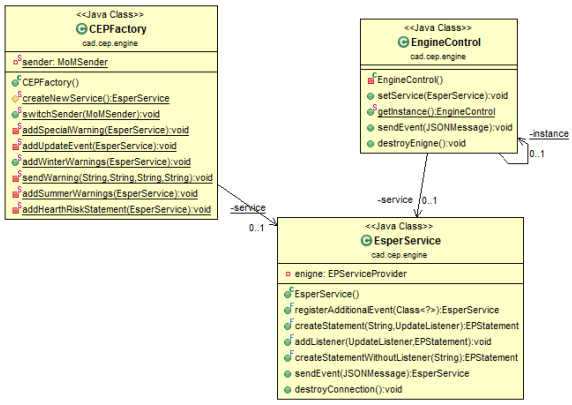

Abbildung 6: Das Klassendiagramm der CEP Klassen

Ein weiteres Statement prüft ob die Luftfeuchtigkeit zu hoch ist. Bei einer hohen Luftfeuchtigkeit liegen tropische Verhältniesse vor, welche in Deutschland eher ungewöhnlich wären. Da alle unsere Wetterdaten aus Deutschland stammen ist es daher wichtig die Clients über eine solche besonderheit zu informieren. Auch dafür gibt es kein Fenster und keinen Consumption Mode. Treten die Bedingungen ein, liegt also eine Luftfeuchtigkeit von über 90 vor, wird ein Alert gesendet. Es dürfen aber auch andere Ereignisse ausgewertet werden. Die Nachricht erhält den Code T1.

```
select * from JSONMessage where humidity >= 90
```

Einige Warnungen sind vorallen im Winter wichtig. Dabei geht es um Schneefall, niedrige Temperaturen und Frost. Die erste Winternachricht W1 warnt über eine Frostgefahr. Dabei werden zwei JSONMessages ausgewertet ob sie nacheinander ins System gekommen sind, dabei aus dem gleichen Ort stammen und Temperaturen unter 0 Grad angeben. Es werden zwei aufeinander folgende Nachrichten benötigt, da Wasser nicht sofort gefriert sondern Zeit braucht. Wenn also zwei Messungen eine neagtive Temperatur angeben, hatte das Wasser Zeit zu gefrieren. Sollte auf eine Nachricht mti negativer Temperatur eine Nachricht mit positiever Temperatur oder 0 Grad folgen, wird davon die erste Nachricht entfernt da das Wasser nicht genug zeit zum gefrieren hatte. Das Statement sieht wie folgt aus:

```
select * from pattern[m1=JSONMessage ->
m2=JSONMessage
(m1.plz = m2.plz and m1.temperature <0 and m2.temperature <0]
```

Über die Identifier m1 oder m2 können die einzelnen Nachrichten im Listener angesprochen werden. Der Listener sendet die Nachricht wieder als Alert weiter. W2 ist die zweite Nachricht und gibt eine Meldung bei besonderst kalten Temperaturen aus. Dabei benötigt es nur eine Nachricht mit einer Temperatur von unter -9 Grad um eine Warnung auszugeben. Niedirgere Temperaturen sind normalerweise selten in Deutschland und generell eher gefährlich. Daher muss eine Warnung an die Clients weitergeben werden. Das Statement funktioniert wie die bereits beschriebenen Statements.

```
select * from JSONMessage where temperature <=-9
```

Die Letzte Winternachricht befasst sich mit starken Schneefall. Dabei wrid der Wettercode von der Wetterapi ausgewertet. Der Code 622 bedeutet starker Schneefall. Die Clients können den Code zwar auswerten und auch entsprechend Meldungen ausgeben aber starker Schneefall kann gefährlich werden. Daher wird eine Meldung weitergesendet um sicherzugehen das alle Clients auch ihre Nutzer warnen. Das Statement sieht wie folgt aus.

```
select * from JSONMessage where currentWeatherId = 622
```

Für den Sommer gibt es eine eigene Warnung. Diese warnt den Nutzer vor sehr guten Wetter. Das ist normalerweise nicht gefährlich aber es kann Clients interessieren ob sie ihren Nutzern sagen können, dass diese Baden gehen können. Dabei wurden als Bedingungen eine Temperatur über 25 Grad und ein klarer Himmel verwendet. Die ID dieses Alarms ist S1.

```
select * from JSONMessage where temperature > 25 and currentWeatherId >= 8
```

Die letzte Warnungskategorie beschäftigt sich mit Herzinfaktrisiken. Die Anwendung ist keine Medizinische Anwendung und folgt auch nicht den denentsprechenden Richtlinien. Daher kann die Anwendung nicht eine genaue Aussage darüber treffen ob ein Herzkranker Tabletten nehmen muss. Die Anwendung kann aber Aussgaben darüber treffen was laut Statistiken gefährlich für Menschen mit Herzproblemen werden könnte. Das stellt aber keine Empfehlung Medikamente zu nehmen dar. Es soll die Nutzer nur darauf hinweisen das sie sich Gedanken machen sollten, ob sie davon betroffen werden können. Schwankt der Luftdruck zu strark, kann bei einigen Menschen es zu einen Herzinfakt kommen. Daher gibt es ein Statement welches überprüft ob in den letzten 30 Minuten eine starke Schwankung stattgefunden hat. Wenn eine Meldung in den Eventstream gelangt wird diese aufbewahrt bis das Zeitfenster abgelaufen ist oder das Muster erkannt wurde. Das Statement sieht wie folgt aus:

```
select * from pattern[d1=JSONMessage ->
d2=JSONMessage
(p1.plz = p2.plz and
 (p2.pressure - p1.pressure <-5 or
p1.pressure - p2.pressure <-10))
 where timer:within(30 min)]
```

Das Statement W2 beschäftigt sich mit der Gefahr einer zu hohen Temperatur. Sollte diese über 25 Grad sein kann es in eingien Fällen das herzinfaktrisiko erhöhen. Das Statement dazu entspricht den anderen Updatestatements.

```
select * from JSONMessage wgere temperature >=25
```

Zu jeden Eintrag in der Tabelle 3 gibt es ein eigenes Statement. Dazu gibt es ein Statement um jede Nachricht zu erfassen. Dabei werden keine unique Consumptionmodes

verwendet da eine Nachricht mehrere Statements auslösen kann. Es kann zum Beispiel gutes Badewetter, ein hoher Luftddruck und eine hohe Luftfeuchtigkeit gleichzeitig eintreten. Genauso wie eine Nachricht nicht im Eventstream bis zum erreichen eines Fensterendes oder eines Musters blockiert werden kann. Während die erste Nachricht darauf wartet wegen der temperatur Schwankungen festgehalten zu werden, kann eine andere Nachricht eingehen die in Kompination mit der ersten Nachricht ein anderes Muster erfüllt.

6.5 Ablauf

Sobald die Anwendung gestartet wurde, ließt diese alle ihr bekannten Städte aus. Für jede Postleitzahl werden zwei Reader-threads erstellt. Einer davon wartet auf alle Nachrichten unter den Topic plz/today und der andere wartet auf alle Nachrichten unter dem Topic plz/weekly. Damit der Reader die Nachrichten auch erhält muss eine Verbindung zur MoM erstellt werden. Dafür wird die Adresse und die Credentials für die MoM aus den Umgebungsvariablen ausgelesen. Die Nachrichten kommen in einen JSON Format an. Dieses Format wird versucht in eine Instanz der Klasse JSONMessage umzuwandeln. Klappt das nicht wird verscuht daraus eine WeeklyForcast Nachricht zu machen. Klappt das auch nicht gibt die Anwendugn eine Fehlermeldung aus und ignoriert die Nachricht, Jede JSONMessage wird als Event den Eventstream der CEP weitergegeben. Sollte die Engine eine Besonderheit finden wird eine Warnung an plz/alert gesendet. Auf jedenfall wird die Nachricht an plz/today/cep weitergeleitet. Die Wochenvorhersage wird an plz/weekly/cep gesendet.

6.6 Erfüllte Anforderungen

Die Anwendung muss sich natürlich an die Anforderungen des 12-Faktor App Standards halten. Die erste Anforderung Codebase wird dadurch erfüllt das die Anwendung selbständig und alleine Funktionieren kann. Die anderen Komponenten, wie zum Beispiel die Wetter API sind nicht notwendig. Sie wurden nur mit entwickelt um ein Vollständiges Scenario darstellen zu können. Die einzige notwendige andere komponente ist eine MoM. Dabei kann die MoM aber jederzeit ausgetauscht werden. Die MoM wurde auch nicht von uns entwickelt sondern nur auf dem Server aufgesetzt.
Die Abhänigkeiten werden mit Maven verwaltet. Jede Benötigte Bibliothek wird einfach als Dependency hinzugefügt und diese wird, mit allen von dieser Benötigten Bibliotheken, runtergeladen und im Buildpath hinzugefügt. Neue Versionen oder Abhänigkeiten können einfach mit einer Zeile mehr in der Konfiguration hinzugefügt werden. Es wird nichts an Code Verwendet was nicht entweder von Java selbst kommt (Version 1.8) oder aus einer in Maven definierten Bibilothek vorhanden ist. Die Anwendung geht niemals davon aus das etwas implizit vorhanden ist.
Die Addresse der MoM, der Nutzername und das Passworrt sind alle in Umgebungsvariablen festgelegt. Die einzige Konsigurationsdatei im Projekt ist die pom.xml. Diese hat

aber mit der laufenden Anwendung nach dem erstellen des Builds nichts mehr zu tun.
Die Anwendung verwendet als anderen Service nur die MoM. Diese wird über eine API angesprochen. Die API ist allgemeingültig für jede MoM, sofern sie MQTT unterstüzt.
Der Relasebuild kann nur von Jenkins erstellt werden. Dieses erstellt alle zusätzlichen für eine veröffentlichte Version benötigten Datein. Jenkins erlangt den Code über das Github Repository. Zu den Dateien gehört unter anderen die Konfiguration für das Deployment auf der Cloud.
Säntliche Daten werden nur kurzeitig von der CEP Engine gespeichert. Die Anwendung legt keine Files an welche länger als die Ausführungszeit auf dem Server vorhanden sind. Daten die nicht mehr vorhanden sind, werden auch nicht mehr aufgerufen. Die CEP kennt nur die Temporär gespeicherten Daten. jede Datei die nicht mehr gespeichert ist, wird von der Engine nicht mehr berücksichtigt.
Die Anwendung braucht nur eine JVM zum Laufen. Ansonsten wird davon ausgegangen das über die Amazon-Cloud oder die Cloud eines anderen Anbieters über eine Einstellung die Anwendung mit einen Port verknüpft werden kann.
Es werden verschiedene Threads verwendet. Diese werden aber nicht wegen der zum skalieren verwendet, sondern um sicherzugehen das eine Nachricht ausgewertet werden kann während eine andere gesendet wird. Es soll also nur die Ausführungszeit verschiedener nebenläufig ausführbarer Schritte optimiert werden.
Es macht keinen unterschied ob di Anwendung normal geschlossen wurde oder abgestürzt ist. Die CEP Engine und die MoM Verbindung werden geschlossen und können ohne Probleme wieder geöffnet werden. Wenn eine Verbindung wieder gebraucht wird, wird diese automatsich wieder geöffnet. Nur die bisherigen Muster gehen Verloren wenn die Anwendung plötzlich abstürtzt. Wenn die Anwendung nicht läuft, kann sie aber auch keine Warnungen versenden. Daher ist es dann egal ob ein Muster vorhanden ist. Wenn die Anwendung neu gestartet wird, wird das Muster wiedererkannt. Da ohnehin jede Minute Daten in die MoM gelangen kann es also nur zu einer Verzögerung von bis zu zwei Minuten + die Ausfallzeit kommen. Zumindest wenn die Situation noch besteht.
TODO: Absatz über Deployment
Es werden keine Log-Dateien angelegt. Die Anwendug gibt Mledungen über den Output- und den ErrorStream aus. An einer anderen Stelle muss daraus ein Log oder eine sonstige Auswertung erfolgen.
TODO Admin Process absatz
In der folgenden Tabelle werden die Anforderungen nochtmal zusammengefasst dargestellt.

Validierung nach "12 Faktor APP"			
ID	**Anforderung**	**Validierungs Element**	**Erfüllt**
1.	Codebase	Andere Komponenten sind nur für das Testszenario da. Deployment verschiederner Versionen über Repo möglich	Ja
2.	Abhängigkeiten	Abhängigkeiten werden über Maven verwaltet	Ja
3.	Konfiguration	Bis auf Buildconfigurations gibt es keine Konfigurationsdatei. Alles andere über Umgebungsvariablen	Ja
4.	Unterstützende Dienste	Einziger Unterstüztender Dienst ist die MoM. Diese kann mit einer Umgebungsvariable ausgetauscht werden	Ja
5.	Build, release, run	Wird von Jenkins verwaltet	Ja
6.	Prozesse	Es gibt keine langfristig gespeicherten Daten	Ja
7.	Bindung an Ports	Wird von Amazoneinstellungen übernommen	Ja
8.	Nebenläufigkeit	Es wird nicht mit threads skaliert	Ja
9.	Einweggebrauch	Verbindungen können beliebig geschlossen oder geöffnet werden	Ja
10.	Dev-Prod-Vergleichbarkeit	....	Nein
11.	Logs	Es wird mit den Streams gearbeitet.	Ka
12.	Admin-Prozesse	....	Nein

Tab. 4: Validierung der CEP nach "12 Faktor APP"

7 Anwendersicht

7.1 Android-APP

7.2 Web-Client

8 Deployment

8.1 Allgemein

8.2 Cloudfoundry

8.3 Docker

Die in Absatz 3.1 erläuterte message-oriented Middleware RabbitMQ wird als Docker Container deployed. Durch die Verwendung von Docker Containern ist es möglich, lauffähige Software in isolierten Containern zu starten. Dies hat den Vorteil, dass die Software immer in identischen Umgebungen gestartet wird, unabhängig davon ob der Container lokal auf dem Entwicklungsrechner oder auf dem Produktivsystem läuft. Dies betrifft auch die Abhängigkeiten von notwendigen Installationen. So basiert RabbitMQ wie in Absatz 3.1 erwähnt auf der Sprache Erlang, weshalb diese auf jedem Entwicklungs- und Produktivsystem installiert werden müsste. Durch die Verwendung von Docker können notwendige Installationen bereits im Dockerfile definiert werden. So wird beispielsweise die Installation von Erlang in Abb. 7 veranschaulicht.

```
# install Erlang
RUN apt-get update \
        && apt-get install -y --no-install-recommends \
                erlang-asn1 \
                erlang-base-hipe \
                erlang-crypto \
                erlang-eldap \
                erlang-inets \
                erlang-mnesia \
                erlang-nox \
                erlang-os-mon \
                erlang-public-key \
                erlang-ssl \
                erlang-xmerl \
        && rm -rf /var/lib/apt/lists/*
```

Abbildung 7: Installation von Erlang im Dockerfile Quelle: https://github.com/docker-library/rabbitmq/blob/a6cb36022a5c1a17df78cfafe45a73d941ad4eb8/3.6/-debian/Dockerfile

Dies ist ein Ausschnitt aus dem Dockerfile des offiziellen Rabbitmq-Baseimages, einer Abbildung des Containers. Der Abschnitt stellt die Installation von Erlang auf einem Linux-System unter Verwendung des Paketmanagers *Advanced Packaging Tool* (APT) dar.

8.3.1 Dockerfile

Das Dockerfile für diesen Use-Case besteht neben dem erwähnten Baseimage aus Aktivierungen des in Absatz 3.1 vorgestellten Management Plugins, des MQTT Plugins sowie des MQTT-Websocket Plugins. Durch die Verwendung des Websocket Plugins ist die Kommunikation mit RabbitMQ auch über Webseiten möglich. Ein weiterer Bestandteil des Dockerfiles ist die rabbitmq.config. Diese ist notwendig um den Zugriff auf die MOM einzuschränken, da andernfalls eine Default-Konfiguration verwendet werden würde und durch diese der Nutzer *guest* vollen Zugriff hätte. Um dennoch Administrations-Zugriff zu erlangen, wird über das Shell-Skript *init.sh* ein Nutzer mit Administratorrechten angelegt. Die Zugangsdaten dieses Nutzers sind als Umgebungsvariablen hinterlegt. Die Ausführung des Skripts wird über den CMD-Befehl des Dockerfiles gesteuert. Zusätzlich ist es notwendig, Docker über den Befehl EXPOSE zu informieren, welche Ports der Container abhört. Im Anschluss an die Erstellung des Dockerfiles kann durch den Befehl *docker build* ein Image erzeugt werden. Dieses Image wird auf den Amazon Container Service deployed.

8.3.2 Amazon Container Service

Dabei handelt es sich um einen hoch skalierbaren Container Managementservice welcher eine unkomplizierte Handhabung von Docker Containern in Amazon EC2 Instanzen anbietet. Die Erstellung eines Clusters ist in wenigen Schritten möglich. Zunächst wird ein Repository zur Speicherung des erstellten Images angelegt. Im Anschluss daran kann nach der erforderlichen Installation des Amazon Web Service Command Line Interfaces (AWS CLI) der Zugriff auf das Repository erfolgen. Nachdem das Image gepusht wurde, muss die Task Definition erstellt werden. Dabei handelt es sich um eine Anleitung für den Start des Containers. Ein Bestandteil der Task Definition ist das Port Mapping. Dabei wird dem Service mitgeteilt, wie die im Dockerfile deklarierten Ports ausserhalb des Containers erreichbar sein sollen. Zusätzlich werden in den erweiterten Optionen die Umgebungsvariablen und somit die Zugangsdaten für den in der *init.sh* erstellten Administator hinterlegt. Im nächsten Schritt erfolgt die Konfiguration des Services, in welcher ein Load Balancer erstellt und ausgewählt werden kann. Im abschließenden Schritt erfolgt die Konfiguration des eigentlichen Clusters. Hierbei erfolgt die Auswahl der Art und der Anzahl des gewünschten Instanz Typs. Durch diesen wird festgelegt, welche Resourcen für eine einzelne Instanz verfügbar sein werden. Darüber hinaus kann in diesem Abschnitt ein Schlüsselpaar für den SSH-Zugriff erstellt und ausgewählt werden. Erfolgt dies nicht, ist kein Zugriff über die EC2 Konsole auf den Container möglich. Die Skalierbarkeit erfolgt durch den optionalen Auto Scaling Service. Dieser kann dahingehend konfiguriert werden, dass die minimale, eine maximale und eine gewünschte Anzahl Instanzen pro Task festgelegt wird.

8.4 REDIS

8.5 AWS

9 Kostenmodell