Bachelorarbeit

Evaluierung einer Amazon Web Services© Lösung zur Erfassung und Verarbeitung von Sensordaten

An der Fachhochschule Dortmund
im Fachbereich Informatik
Studiengang SWT Dual
erstellte Bachelorarbeit

von
Hendrik Hagmans
geb. am 02.04.1992
Matr.-Nr. 7082973

Betreuer:
Prof. Dr. Martin Hirsch
Prof. Dr. Sabine Sachweh

Dortmund, 4. März 2016

Zusammenfassung

Amazon Web Services gibt es nun schon seit einigen Jahren und bietet verschiedenste Cloud Services für unterschiedlichste Anforderungen. Amazon Web Services unterscheidet sich von vielen anderen Anbietern vor allem durch die variable Leistungsabrechnung und die Vielfalt der Angebote. In dieser Arbeit soll nun mittels eines Beispiels evaluiert werden, ob Amazon Web Services auch für große Datenströme wie beispielsweise Sensordaten geeignet ist.

Abstract

Amazon Web Services are present for quite a few years and offer several services for different requirements. The most important differences between Amazon Web Services and other cloud computing providers is flexible service billing and the diversity of their offers. In this work it will be evaluated by means of an example if Amazon Web Services are suitable for use with big data streams like sensor data.

Inhaltsverzeichnis

1. Einleitung

1.1 Motivation

Cloud Computing ist ein immer größer werdendes Thema in der IT. Immer mehr Daten werden nicht mehr lokal, sondern in der Cloud gespeichert und werden somit für jeden überall verfügbar. Auch Anwendungen werden immer häufiger nicht mehr lokal betrieben, sondern nutzen die großen Rechnerleistungen von Cloud Lösungen, um möglichst skalierbar zu sein und mittels Redundanz höhere Verfügbarkeiten und bessere Latenzzeiten zu erreichen. Zudem bieten Cloud Lösungen die Möglichkeit, Investitionskosten in Betriebskosten umzuwandeln, da keine neuen Server und andere Geräte gekauft werden müssen, sondern für die Nutzung bezahlt wird. Webapplikationen bilden hier keine Ausnahme. Anstatt eigene Rechenzentren aufbauen zu müssen, werden Webapplikationen immer häufiger bei externen Cloudanbietern betrieben um Kosten und den Administrationsaufwand solcher Systeme zu reduzieren. Ein großer Anbieter solcher Plattformen ist Amazon, die mit ihren Amazon Web Services [3] eine Reihe von Cloud Services bieten, die von vielen weltweit operierenden Unternehmen genutzt wird. Doch wie kann man mittels Amazon Web Services große Datenflüsse wie beispielsweise die Aufzeichnung von Sensordaten in einem automatisierten Heimsystem am besten verwalten?

1.2 Zielsetzung

Es soll eine Wettersimulation erstellt werden, die aus mehreren Services besteht, die dauerhaft Daten liefern. Beispielsweise mehrere virtuelle Temperatursensoren, die Zufallszahlen innerhalb eines bestimmten Werteintervalls liefern. Die Temperaturwerte

können sich auch gegenseitig beeinflussen. So beeinflusst eine höhere Außentemperatur auch die Innentemperatur. Diese Komponenten werden jeweils in einem Docker Container [4] auf Amazon ECS [5] deployed und liefern einen konstanten Datenstrom von Temperaturdaten. Der Datenstrom soll skaliert werden können und bspw. die Daten eines ganzen Tages in einer Stunde erzeugen. Die Daten werden in einer SQL Datenbank auf Amazon RDS [7] oder Amazon DynamoDB [8] gespeichert. Eine eigene Lösung mit Apache Cassandra [24] wäre auch möglich. Im Dashboard soll eine Übersicht der Daten in Verlaufsdiagrammen verfügbar sein. Hier ist auch eine Anpassung der Zeitskalierung möglich.
Der Fokus liegt hierbei nicht auf einer akkuraten Wettersimulation, sondern auf der Evaluation der AWS Services für große Datenströme und dem Vergleich zwischen den einzelnen AWS Services. Bspw. könnten Teile der Architektur auch mit Amazon Kinesis [9] umgesetzt werden, das speziell für große Datenströme konzipiert wurde. Hier könnte man beide Architekturentwürfe vergleichen.
Dieses Projekt wurde in einer Projektarbeit bereits geplant und analysiert und soll in dieser Bachelorarbeit nun implementiert werden. Auf Basis dieser Implementierung soll eine Evaluation der verwendeten AWS Komponenten durchgeführt werden.

1.3 Vorgehensweise

Als erstes wird die Projektplanung zusammengefasst. Diese wurde bereits im Rahmen einer Projektarbeit [1] erarbeitet und muss daher nicht mehr ausführlich behandelt werden. Es wird nur kurz auf die einzelnen Planungsschritte und die Ergebnisse dieser eingegangen.
Dann wird auf die Implementierung eingegangen, wobei zunächst die Projektarchitektur beschrieben wird, aus welchen Komponenten das Projekt aufgebaut ist und wie diese Komponenten arbeiten und miteinander in Verbindung stehen. Es wird auch auf Abweichungen im Vergleich zur Projektplanung eingegangen.
Im nächsten Schritt wird AWS IoT als weitere Alternative im Vergleich der im Projekt genutzten Amazon Web Services vorgestellt. Hierbei wird gezeigt, wie auch echte Sensoren mit AWS verbunden werden können, welche Funktionen die IoT Lösung von Amazon bietet und welche Vor- und Nachteile AWS IoT vorweist.
Zuletzt werden die genutzten AWS Komponenten evaluiert und es wird auch auf den

Nutzen von AWS Iot eingegangen. Zudem werden weitere Implementierungsmöglichkeiten und deren Vor- und Nachteile erörtert.

2. Projektplanung

Das Projekt wurde bereits in einer vorher bearbeiteten Projektarbeit [1] geplant. Dabei wurden mehrere Analysen durchgeführt, die in diesem Kapitel vorgestellt werden. Als Ergebnis der Anforderungsanalyse wurde folgende Liste von Anforderungen erstellt:

Thema	Beschreibung	Kano Bewertung
Dauerhafter Datenstrom	Producer sollen dauerhaft Temperaturdaten liefern	Basismerkmal
Daten Persistenz	Die Daten sollen dauerhaft in einer Datenbank persistiert werden	Basismerkmal
Start und Stopp des Datenstroms	Der Datenstrom soll vom Nutzer gestartet und gestoppt werden können	Basismerkmal
Datenstrom Skalierung	Der Datenstrom soll skaliert werden können. Beispielsweise sollen die Daten eines Tages in einer Stunde ausgegeben werden können	Basismerkmal
Dashboard	Nutzer soll die aktuellen Daten in einem Dashboard einsehen können	Basismerkmal
Mehrere AWS Services	Die Applikation soll auf mehreren AWS Services deployed werden, um diese vergleichen zu können	Basismerkmal
Dashboard Darstellung	Die Daten werden im Dashboard in verschiedenen Diagrammen wie bspw. Verlaufsdiagrammen dargestellt sowie in Tabellen.	Leistungsmerkmal

Tabelle 2.1: Anforderungen an das Projekt Teil 1

Thema	Beschreibung	Kano Bewertung
Verfügbarkeit	Die Services sollen hochverfügbar sein	Leistungsmerkmal
Aktualität der Daten	Die Daten sollen im Dashboard aktuell gehalten werden, auch wenn die Seite nicht neu geladen wird	Begeisterungsmerkmal
Temperaturen beeinflussen sich gegenseitig	Die Temperaturen sollen sich gegenseitig beeinflussen, z.B. bedingt eine höhere Außentemperatur eine höhere Innentemperatur	Leistungsmerkmal

Tabelle 2.2: Anforderungen an das Projekt Teil 2

Die Anforderungsliste ist nach der Priorisierung sortiert, die im Rahmen einer Kano Bewertung der einzelnen Punkte erstellt wurde. Wie in den Tabellen 2.1 und 2.2 zu sehen, wurden die Basismerkmale am höchsten priorisiert, da diese den Erfolg des Projektes ausmachen. Auf der Basis dieser Anforderungsanalyse wurde zudem folgendes Pflichtenheft erstellt:

Thema	Beschreibung	Aufwand
AWS kennenlernen	Kennenlernen der AWS Services und erste Deployments von Containern	10
Docker Image	Docker Image mit allen benötigten Ressourcen erstellen	3
Docker Container	Docker Container aus dem Image mit der fertigen Applikation erzeugen	3
Producer	Es müssen mehrere Producer geschrieben werden, die konstant Temperaturdaten liefern. Die Producer werden in Java geschrieben	20
Consumer	Es muss mindestens ein Consumer geschrieben werden, der die Daten der Producer verarbeitet. Der Consumer wird in Java geschrieben	20

Tabelle 2.3: Pflichtenheft Teil 1

Thema	Beschreibung	Aufwand
Consumer DB	Es muss eine Datenbank entweder auf Amazon RDS oder Amazon DynamoDB (oder andere Lösungen bspw. mit Cassandra) eingerichtet werden	10
Consumer DB Zugriff	Der Consumer muss die Temperaturdaten in die DB schreiben können	10
Mehrere AWS Services	Die Applikation für mehrere AWS Services kompatibel machen und auf mehreren Services deployen	20
Dashboard	Es muss ein Dashboard geschrieben werden, das die Temperaturdaten anzeigen kann	15
Dashboard Start Stopp	Es muss im Dashboard die Funktion geben den Datenstrom anhalten oder wieder starten zu können	3
Dashboard Zeitskalierung	Es muss im Dashboard die Funktion geben den Datenstrom verschnellern oder verlangsamen zu können	7
Dashboard Diagramme	Die Daten sollten im Dashboard in Form von Diagrammen dargestellt werden	10
Dashboard Diagramme Aktualität	Die Daten sollten im Dashboard immer aktuell gehalten werden, auch wenn der Nutzer die Seite nicht aktualisiert	5
Producer Temperatur Beeinflussung	Die Temperaturwerte der Producer müssen sich gegenseitig beeinflussen. Die Producer müssen also untereinander kommunizieren und zumindest Wechsel in der Temperaturtendenz interessierten anderen Producern mitteilen.	10

Tabelle 2.4: Pflichtenheft Teil 2

Das Pflichtenheft ist genau wie die Anforderungsliste nach der Priorisierung sortiert, die im Rahmen einer Kano Bewertung der einzelnen Punkte erstellt wurde. Genauere

Erläuterungen der einzelnen Anforderungen und Punkte des Pflichtenhefts finden sich in der Projektarbeit.
Des weiteren wurde im Rahmen der Projektplanung eine Risikoanalyse durchgeführt, um die größten Risiken des Projekts zu erkennen und dementsprechende Gegenmaßnahmen anwenden zu können. Hierbei haben sich folgende 4 Risiken herausgestellt:

Nummer	Risiko	Eintrittswahrscheinlichkeit in %	geschätzter Schaden in €	Risikofaktor
1	AWS Zugriff zu spät bekommen	30	300	90
2	Producer erzeugt zu viele Daten und damit zu viele Kosten bei AWS	150	20	75
3	Teile des Projekts werden nicht rechtzeitig vor Abgabe erstellt	10	1000	100
4	Unterschätzen des Umfangs oder der Schwierigkeit des Projektes	10	1000	100

Tabelle 2.5: Risikoliste

Der Risikofaktor entspricht der Formel $\frac{Eintrittswahrscheinlichkeit \cdot Schaden}{100}$.Dementsprechend befinden sich gerade die Risiken Nummer 3 und 4 durchaus in einem gefährlichen Bereich, der das Projekt gefährden könnte. In der Projektarbeit wurden allerdings Maßnahmen zur Verhinderung des Eintretens der Risiken sowie der Verminderung der Kosten bei Eintreten der Risiken ermittelt, die in der Ausführung des Projektes auch umgesetzt wurden.
Zu guter Letzt wurde eine Kostenplanung erstellt, da die Nutzung von Amazon Web Services Kosten verursachen kann und diese nicht zu hoch ausfallen sollten. Es wurde ein Budget von bis zu 200 € gewährt, das nicht überschritten werden sollte. Daher mussten die aktuellen Kosten ständig überwacht werden.
Im Rahmen der Projektarbeit wurden zudem die Themen und Komponenten, die Teil der Bachelorarbeit sein sollten, behandelt und beschrieben. Darunter zählten die Themen Cloud Computing, Amazon Web Services und deren einzelne Komponenten sowie Docker. Daher müssen diese Themen in dieser Bachelorarbeit nicht mehr ausführlich behandelt werden.
Eine erste Evaluation der in Frage kommenden Amazon Web Services wurde ebenfalls durchgeführt und dabei eine Kombination aus Amazon EC2 als Infrastruktur für die

Producer und Consumer, Amazon Kinesis als Übertragungskanal für die Temperaturdaten sowie Amazon RDS oder Amazon DynamoDB für die Persistierung der Daten als passende Services ausgemacht.

3. Implementierung

In diesem Abschnitt wird aus die Implementierung des Projekts eingegangen. Dabei wird zunächst die Projektarchitektur erklärt und dann werden die einzelnen Komponenten des Projekts, Producer, Consumer und die Webapplikation beschrieben. Zudem wird darauf eingegangen, wie die genutzten AWS Ressourcen gelöscht werden können und wie das Projekt in einem Docker Container verwendet werden kann. Zuletzt werden Abweichungen im Verglich zur Projektplanung besprochen.

3.1 Projektarchitektur

Um die Abhängigkeitsverwaltung und das Bauen des Projektes möglichst simpel zu gestalten, wurde für das Projekt Apache Maven [25] genutzt und das Projekt dementsprechend als Maven Projekt erstellt.
Abhängigkeiten des Projekts sind der Amazon Kinesis Client in der Version 1.6.1, der Amazon Kinesis Producer in der Version 0.10.2 sowie Eclipse Jetty Servlet [27] in der Version 9.2.14.v20151106.
Teile des Codes basieren auf den Beispielen, die Amazon im AWS Kinesis Developer Guide behandelt (s. [13] und [14]) Die Projektarchitektur ist im Grunde genommen genau so, wie sie von Amazon in der Dokumentation von AWS Kinesis vorgestellt wird.

Wie in Abbildung 3.1 zu sehen, setzt Amazon in der Architektur 4 Schichten voraus. Die Producer, den Kinesis Stream, die Consumer sowie weitere Services außerhalb von Kinesis. Die Daten werden von links nach rechts in der Architektur übertragen. Zunächst einmal werden die Daten in den Producern erzeugt und in den Kinesis Stream geschrieben, in denen sie in einem oder mehreren Shards einige Tage gespeichert bleiben. Ein Shard ist eine Gruppe von Datensätzen in einem Kinesis Stream, die eine

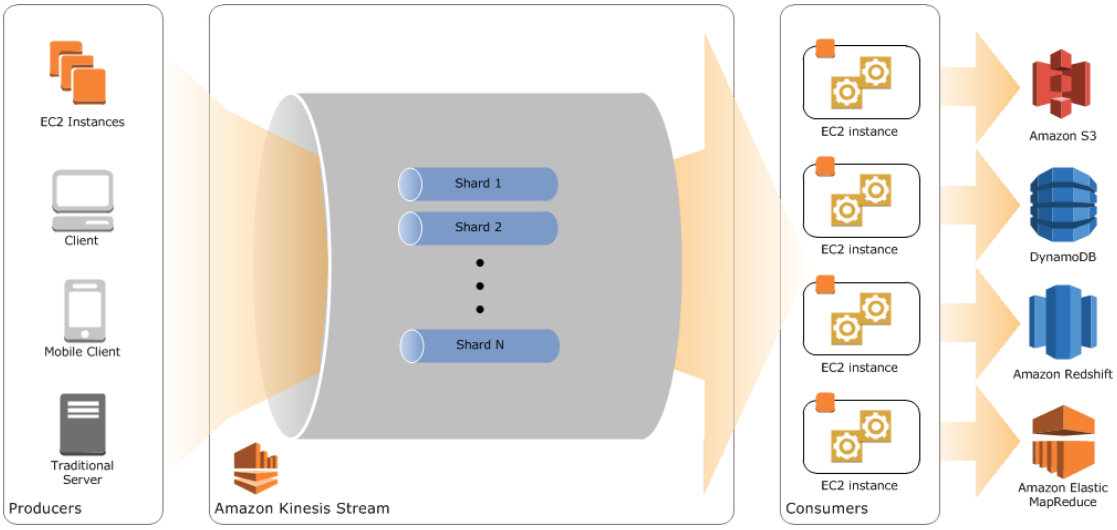

Abbildung 3.1: Kinesis Architektur, wie sie von Amazon vorgegeben wird. Quelle: [31]

feste Menge an Daten aufnehmen können.
Auf der anderen Seite des Kinesis Streams befinden sich ein oder mehrere Consumer, die die Daten aus den Shards des Kinesis Streams lesen. Nach dem Lesen können die Daten zudem an andere Services weitergeleitet werden, wie beispielsweise Amazon DynamoDB, mit dem die Daten in der NoSQL-Datenbank von DynamoDB persistiert werden können.
Genau diese Architektur wurde auch im Projekt umgesetzt. Es gibt eine oder mehrere Instanzen des Producers, der Temperaturdaten erzeugt. Der Producer schreibt die Daten in einen Kinesis Stream, meist nur mit einem Shard, da ein Shard für die Datenmengen dieses Projekts ausreicht. Zudem gibt es eine oder mehrere Instanzen eines Consumers, der die Daten aus dem Kinesis Stream liest und dann in eine DynamoDB Datenbank schreibt. Darüber hinaus enthält dieses Projekt zudem eine Webapplikation, die die Temperaturdaten aus DynamoDB liest und in Diagrammen ausgibt.
Zudem enthält das Projekt Utility-Klassen für DynamoDB, Kinesis sowie zur Temperaturgenerierung, in denen Methoden für die entsprechenden Anforderungsgebiete ausgelagert wurden. Darüber hinaus enthält das Projekt eine "DeleteResources"Klasse, die eine Methode zur Löschung aller verwendeten Ressourcen auf Amazon Webservices bereitstellt.
In der pom.xml des Projekts sind mehrere Profile eingetragen, die es ermöglichen, einzelne Klassen mit Startparametern auszuführen um somit beispielsweise den Producer mit anderen Parametern zu starten (siehe Kapitel 3.2).

3.2 Producer

Der Producer ist für die Erzeugung und Übermittlung der Temperaturwerte zuständig. Er erzeugt zunächst die Temperaturwerte und verschickt diese dann an AWS Kinesis zur weiteren Verarbeitung. Er ist also das Äquivalent zu einem Sensor, der in diesem Fall die Temperaturen aber generiert und nicht aus der Umwelt misst. Es können mehrere Instanzen des Producers erzeugt und gestartet werden um so mehrere Sensoren zu simulieren. Der Producer besteht aus einer einzelne Java-Klasse, die auf mehrere weitere Utility-Klassen zugreift. Darunter fällt die Utils-Klasse, die die Temperaturdaten erzeugt, sowie die StreamUtils-Klasse, die Methoden zum Zugriff auf AWS Kinesis Streams bereitstellt. Der Producer enthält eine Main-Methode, über die der er gestartet werden kann. Zunächst werden die übergebenen Parameter ausgelesen und gesetzt. Die Parameter sind der Kinesis Streamname, der Sensorname, wie lang der Producer ausgeführt werden soll sowie wie viele Temperaturwerte pro Sekunde erzeugt werden sollen. Zudem wird ein Objekt der StreamUtils Klasse initialisiert, damit die Methoden der Klasse benutzt werden können. Schon bei der Initialisierung der Klasse wird der erste Temperaturwert generiert, der nun als Basis für alle weiteren Temperaturwerte dieser Iteration fungiert. Es werden außerdem weitere Variablen initialisiert, zum Beispiel ein ScheduledExecutorService, mit dem die Temperaturen in bestimmten Intervallen abgesendet werden können.

```
1  public static double getFirstTemperature() {
2      return RANGE_MIN + (RANGE_MAX - RANGE_MIN) * RANDOM.nextDouble();
3  }
```

Quelltext 3.1: Utils.java (ll. 54-56): Initialisierung des Temperaturwertes

Quelltext 3.1 zeigt die Methode “getFirstTemperature”, die einen Zufallswert zwischen zwei Schranken ausgibt, der dann als erster Temperaturwert genutzt wird. Diese Schranken sind in diesem Fall -20 und 40, können aber auch im Code angepasst werden. Dementsprechend kann der initiale Temperaturwert nur innerhalb dieser Schranken liegen. Der Rückgabewert dieser Methode wird der Temperaturvariable zugewiesen.

```
1  public void createOrRenewStream(String streamName, int shards)
2          throws AmazonClientException {
3      try {
4          if (isActive(kinesis.describeStream(streamName))) {
5              LOG.info(String.format("Deleting␣stream␣%s....", streamName));
```

```
6               kinesis.deleteStream(streamName);
7               try {
8                   Thread.sleep(CREATION_WAIT_TIME_IN_SECONDS);
9               } catch (InterruptedException e) {
10                  LOG.warn(String
11                          .format("Interrupted while waiting for %s stream to become active. Aborting."
                                ,
12                                  streamName));
13                  return;
14              }
15          }
16      } catch (ResourceNotFoundException ex) {
17      }
18      createStream(streamName, shards);
19  }
```

Quelltext 3.2: StreamUtils.java (ll. 105-123): Erstellung eines neuen Kinesis Streams

Als nächstes wird ein Kinesis-Stream erzeugt, falls nicht schon ein Stream vorhanden ist. Dazu wird mit der Methode “describeStream” der Kinesis Client Library ein “DescribeStreamResult” Objekt des Streams ermittelt (s. Quelltext 3.2) und geprüft, ob der Stream schon erstellt wurde und aktiv ist. Wenn dies nicht der Fall ist, wird ein neuer Stream erzeugt und darauf gewartet, dass dieser aktiv wird.

```
1       final FutureCallback<UserRecordResult> callback = new FutureCallback<UserRecordResult>() {
2           @Override
3           public void onFailure(Throwable t) {
4               // We don't expect any failures during this sample. If it
5               // happens, we will log the first one and exit.
6               if (t instanceof UserRecordFailedException) {
7                   Attempt last = Iterables
8                           .getLast(((UserRecordFailedException) t)
9                                   .getResult().getAttempts());
10                  log.error(String.format("Record failed to put - %s : %s",
11                          last.getErrorCode(), last.getErrorMessage()));
12              }
13              log.error("Exception during put", t);
14              System.exit(1);
15          }
16
17          @Override
18          public void onSuccess(UserRecordResult result) {
19              temperature = Utils.getNextTemperature(temperature);
20              completed.getAndIncrement();
21          }
22      };
```

Quelltext 3.3: TemperatureProducer.java (ll. 229-250): Initialisierung eines Callbacks um Rückmeldungen der Threads zu bekommen

Wie in Quelltext 3.3 zu sehen, wird ein FutureCallback initialisiert, der die Möglichkeit bietet nach jeder Iteration eines Threads auf das Ergebnis dieser Iteration zu reagieren. Bei einem Fehler wird hier also eine Fehlermeldung ausgegeben, bei Erfolg wird ein neuer Temperaturwert erzeugt.

```
1  public static double getNextTemperature(double lastTemperature) {
2      float random = RANDOM.nextFloat();
3      if (random < 0.33) {
4          lastTemperature -= 0.1;
5      } else if (random > 0.66) {
6          lastTemperature += 0.1;
7      }
8      return (double) Math.round(lastTemperature * 10d) / 10d;
9  }
```

Quelltext 3.4: Utils.java (ll. 44-52): Generierung des nächsten Temperaturwertes

Quelltext 3.4 zeigt die Erzeugung eines neuen Temperaturwertes. Dazu wird mit einer jeweils etwa 33 prozentigen Wahrscheinlichkeit der Temperaturwert um 0,1 erhöht bzw. erniedrigt oder bleibt wie er vorher war. Der Rückgabewert wird zudem auf eine Dezimalstelle gerundet.

```
1   final Runnable putOneRecord = new Runnable() {
2       @Override
3       public void run() {
4           ByteBuffer data = Utils.generateData(temperature, sensorName,
5                   DATA_SIZE);
6           // TIMESTAMP is our partition key
7           ListenableFuture<UserRecordResult> f = producer.addUserRecord(
8                   streamName, TIMESTAMP, Utils.randomExplicitHashKey(),
9                   data);
10          Futures.addCallback(f, callback);
11      }
12  };
```

Quelltext 3.5: TemperatureProducer.java (ll. 253-264): Initialisierung eines Runnables zum Senden eines Records

Quelltext 3.5 zeigt die Initialisierung eines Runnables, der die Temperaturdaten an Kinesis schickt. Hier werden zunächst die Temperaturdaten erzeugt, die mithilfe des Kinesis Producers aus der Amazon Kinesis Java API verschickt werden. Das Ergebnis des Sendeversuchs wird an den Callback zurückgegeben. Die Erzeugung der Daten erfolgt in der Methode “Utils.generateData”

```
1   public static ByteBuffer generateData(double lasttemperature,
2           String sensorName, int totalLen) {
3       StringBuilder sb = new StringBuilder();
4       sb.append(Double.toString(getNextTemperature(lasttemperature)));
5       sb.append(";");
6       sb.append(sensorName);
7       sb.append(";");
8       sb.append(String.valueOf(System.currentTimeMillis()));
9       sb.append(";");
10      while (sb.length() < totalLen) {
11          sb.append("a");
12      }
13      try {
14          return ByteBuffer.wrap(sb.toString().getBytes("UTF-8"));
15      } catch (UnsupportedEncodingException e) {
16          throw new RuntimeException(e);
17      }
18  }
19 }
```

Quelltext 3.6: Utils.java (ll. 73-91): Generierung der zu verschickenden Daten

Quelltext 3.6 enthält die Methode “Utils.generateData”. Hier wird mit einem Stringbuilder ein String erzeugt, der dann als Bytebuffer der aufrufenden Methode zurückgegeben wird. Der erzeugte String besteht aus der Temperatur, dem Sensornamen und der aktuellen Zeit in Millisekunden, alle getrennt durch ein Semikolon. Es wird zudem nicht einfach nur die übergebene Temperatur genommen, sondern noch einmal die nächste Temperatur generiert, obwohl dieses auch im Callback wie oben genannt passiert. Dies erhöht die Zufälligkeit der Temperaturänderungen.

```
1       EXECUTOR.scheduleAtFixedRate(new Runnable() {
2           @Override
3           public void run() {
4               long put = sequenceNumber.get();
5               long total = recordsPerSecond * secondsToRun;
6               double putPercent = 100.0 * put / total;
7               long done = completed.get();
8               double donePercent = 100.0 * done / total;
9               log.info(String
10                      .format("Put␣%d␣of␣%d␣so␣far␣(%.2f␣%%),␣%d␣have␣completed␣(%.2f␣%%)",
11                              put, total, putPercent, done, donePercent));
12          }
13      }, 1, 1, TimeUnit.SECONDS);
```

Quelltext 3.7: TemperatureProducer.java (ll. 267-279): Status Updates jede Sekunde über ein Runnable im Executor

Im Producer wird nun dem ScheduledExecutorService (Variable “EXECUTOR”) ein weiteres Runnable hinzugefügt, das einmal pro Sekunde ausgeführt wird. Dieser übergebene Task startet sofort und gibt einmal pro Sekunde den aktuellen Status der Übertragung aus (s. Quelltext 3.7)

```
1        executeAtTargetRate(EXECUTOR, putOneRecord, sequenceNumber,
2                secondsToRun, recordsPerSecond);
```

Quelltext 3.8: TemperatureProducer.java (ll. 285-286): Start der Puts zu Kinesis

Nach der Initialisierung aller Variablen wird nun das Verschicken der Temperaturdaten gestartet. Dies geschieht über den Aufruf in Quelltext 3.8.
Die Methode “executeAtTargetRate” bekommt als Parameter den ScheduledExecutorService, der das Runnable zur Statusausgabe beinhaltet, sowie das Runnable zum Verschicken eines Records und zudem, welche Sequenznummer aktuell verschickt wird und wie lange Daten verschickt werden sollen und wie viele pro Sekunde.

```
1    private static void executeAtTargetRate(
2            final ScheduledExecutorService exec, final Runnable task,
3            final AtomicLong counter, final int durationSeconds,
4            final int ratePerSecond) {
5        exec.scheduleWithFixedDelay(new Runnable() {
6            final long startTime = System.nanoTime();
7
8            @Override
9            public void run() {
10               double secondsRun = (System.nanoTime() - startTime) / 1e9;
11               double targetCount = Math.min(durationSeconds, secondsRun)
12                       * ratePerSecond;
13
14               while (counter.get() < targetCount) {
15                   counter.getAndIncrement();
16                   try {
17                       task.run();
18                   } catch (Exception e) {
19                       log.error("Error running task", e);
20                       System.exit(1);
21                   }
22               }
23
24               if (secondsRun >= durationSeconds) {
25                   exec.shutdown();
26               }
```

```
27              }
28          }, 0, 1, TimeUnit.MILLISECONDS);
29      }
```

Quelltext 3.9: TemperatureProducer.java (ll. 327-355): Die Methode exectueAtTargetRate

Quelltext 3.9 zeigt die Methode “executeAtTargetRate”. Hier wird dem ScheduledExecutorService ein weiteres Runnable übergeben, das das Verschicken der Daten an Kinesis koordiniert. Je nachdem welche Parameter dem Producer übergeben wurden, wird der übergebene Task nun entsprechend lange und häufig ausgeführt. Der Task selbst ist das Runnable, das vor der Methode initialisiert wurde und die Temperaturdaten an Kinesis schickt (Runnable “putOneRecord”).
Wenn alle Puts abgeschlossen sind, terminiert die Methode. Daraufhin wird noch darauf gewartet, dass alle abgeschlossenen Puts auch wirklich von Kinesis verarbeitet wurden. Dann wird der Producer terminiert.
Wie man sieht, werden alle Funktionen des Producers asynchron in seperaten Threads ausgeführt. Diese werden zunächst als Runnables deklariert und im ScheduledExecutorService registriert, so dass sie in einem bestimmten Intervall ausgeführt werden. Die Methode “executeAtTargetRate” sorgt zudem dafür, dass nur im gewählten Intervall Daten an Kinesis verschickt werden.

3.3 Consumer

Der Consumer liest die vom Producer übermittelten Daten aus AWS Kinesis und persistiert diese in AWS DynamoDB zur weiteren Verarbeitung. Er ist eine einzelne Java-Klasse, die auf mehrere weitere Utility-Klassen zugreift. Darunter fällt die StreamUtils-Klasse, die Methoden zum Zugriff auf AWS Kinesis Streams bereitstellt sowie die DynamoDBUtils-Klasse, die Methoden zum Zugriff auf AWS DynamoDB bereitstellt. Der Consumer enthält eine Main-Methode, über die der er gestartet werden kann. Als Parameter wird dem Consumer der Streamname sowie der Tabellenname der DynamoDB Tabelle übergeben. Diese Parameter werden bei Start des Consumers zunächst einmal ausgelesen gesetzt. Außerdem werden jeweils eine Instanz der Streamutils Klasse sowie der DynamoDBUtils Klasse initialisiert, damit die Methoden der Klassen genutzt werden können.
Zunächst wird die DynamoDB Tabelle gelöscht, die Informationen eines Checkpointers sichert, da die dort gesicherten Daten nicht mehr aktuell sind. Der Checkpointer wird

nach dem Lesen einer bestimmten Menge an Temperaturen aufgerufen und speichert Informationen über die gelesen Daten in der DynamoDB Tabelle ab, zum Beispiel die Sequenznummer des letzten gelesenen Wertes. Außerdem wird die Tabelle, in der die Temperaturdaten gespeichert werden, auf Amazon DynamoDB mittels der Methode “createTemperatureTableIfNotExists” der DynamoDBUtils erzeugt, falls sie nicht schon existierte.

```
1        List<KeySchemaElement> ks = new ArrayList<>();
2        ks.add(new KeySchemaElement().withKeyType(KeyType.HASH)
3                .withAttributeName(ATTRIBUTE_NAME_HASH_KEY));
4        ks.add(new KeySchemaElement().withKeyType(KeyType.RANGE)
5                .withAttributeName(ATTRIBUTE_NAME_RANGE_KEY));
```

Quelltext 3.10: DynamoDBUtils.java (ll. 95-99): Setzung des Hashkeys und Rangekeys der Temperaturtabelle

Quelltext 3.10 zeigt einen Ausschnitt der Methode “createTemperatureTableIfNotExists”, in der die Hash- und Rangekeys der Tabelle gesetzt werden. Bei der Temperaturtabelle wird als Hashkey der Sensorname gesetzt und als Rangekey der Timestamp des Sensordurchlaufs. Der Hashkey partitioniert die Daten in mehrere Blöcke, hier also in Datenblöcke der einzelnen Sensoren. Mittels des Rangekeys können die partitionierten Daten zudem sortiert werden. Zusammengenommen ergeben der Hashkey und der Rangekey den Primary Key der Daten.

```
1        final  TemperatureConsumer consumer = new TemperatureConsumer();
2
3        new Worker.Builder().recordProcessorFactory(consumer).config(config)
4                .build () .run() ;
```

Quelltext 3.11: TemperatureConsumer.java (ll. 241-244): Erzeugung und Starten des IRecordProcessors

Nach dem Erzeugen der Temperaturtabelle wird eine Instanz des Temperatureconsumers erzeugt und über eine “IRecordProcessorFactory” ein “IRecordProcessor” erzeugt und gestartet (s. Quelltext 3.11). Der Consumer implementiert das “IRecordProcessorFactory” Interface und fungiert als Fabrik für den “IRecordProcessor”. In einer inneren Klasse ist zudem das “IRecordProcessor” Interface implementiert. Mittels des IRecordProcessor kann über die Records innerhalb eines Kinesis Streams iteriert werden und die Daten können weiterverarbeitet werden, also bspw. in einer Datenbank persistiert werden.

In der inneren Klasse wurden 3 Methoden des “IRecordProcessor” Interfaces implementiert. In der “initialize” Methode wird eine Instanz der DBUtils erzeugt. In der “shutdown” Methode wird der Grund des Abbruchs ins Log geschrieben und außerdem ein Eintrag in der Checkpointer Tabelle erzeugt. Die dritte implementierte Methode ist die “processRecords” Methode, die die Records verarbeitet.

```
1        long timestamp = 0;
2        HashMap<String, HashMap<String, String>> allTemperatures = new HashMap<>();
3        int count = 0;
```

Quelltext 3.12: TemperatureConsumer.java (ll. 121-123): Initialisierung von Variablen in der “processRecords” Methode

Zunächst werden wie in Quelltext 3.12 zu sehen einige Variablen initialisiert. Es wird die Variable timestamp initialisiert, die den Timestamp des aktuellen Temperaturwertes annimmt. Die Variable count entspricht der Anzahl der Temperaturwerte. Die HashMap allTemperatures nimmt die Werte aller Temperaturen auf. Der Key der Hashmap ist der Sensorname und damit der Hashkey der DynamoDB Tabelle. Die Values sind weitere HashMaps. Keys dieser Hashmaps sind die timeStamps der Temperaturen. Die Values sind die Temperaturdaten. Der Timestamp des Durchlaufs liest der “IRecordProcessor” als PartitionKey aus dem Stream und bildet den Rangekey.

```
1        for (Record r : records) {
2            // Get the timestamp of this run from the partition key.
3            timestamp = Math.max(timestamp,
4                    Long.parseLong(r.getPartitionKey()));
5            // Extract the data. All data are sperated with a semicolon
6            try {
7                byte[] b = new byte[r.getData().remaining()];
8                r.getData().get(b);
9                String[] splittedString = new String(b, "UTF-8").split(";");
10               String currentTemperature = splittedString[0];
11               String sensorName = (splittedString[1]);
12               String currentTimeStamp = (splittedString[2]);
13
14               // Create a new hashmap, if there isn't already one, and
15               // combine the old and new temperature data
16               HashMap<String, String> tempMap;
17               if (allTemperatures.containsKey(sensorName)) {
18                   tempMap = allTemperatures.get(sensorName);
19               } else {
20                   tempMap = new HashMap<>();
21               }
22               tempMap.put(currentTimeStamp, currentTemperature);
```

```
23                  allTemperatures.put(sensorName, tempMap);
24                  logResults(Long.valueOf(currentTimeStamp), count,
25                          sensorName, currentTemperature);
26                  count++;
```

Quelltext 3.13: TemperatureConsumer.java (ll. 124-149): Iteration über die Records innerhalb eines Streams

Der Quelltext 3.13 zeigt den Verarbeitungsprozess der Daten in der Methode “processRecords”. Es wird über alle Records iteriert und zunächst der Timestamp als Rangekey extrahiert. Dann werden die Daten des Records extrahiert. Der im Producer erzeugte String wird dabei in die einzelnen Teile aufgeteilt und enthält die Daten für einen Temperaturwert eines Sensors, gemessen an einem bestimmten Zeitpunkt.. Als nächstes wird geprüft, ob in der Hashmap allTemperatures schon eine Hashmap für den Sensor, der den Temperaturwert erzeugt hat, vorhanden ist. Wenn ja, wird diese Hashmap aus allTemperatures gelesen, ansonsten wird eine neue erzeugt. In diese Hashmap wird dann der Temperaturwert und der Zeitpunkt eingefügt und die aktualisierte Map des Sensors wird wieder in die allTemperatures Hashmap eingefügt. Es wird ein Log mit den gelesenen Daten ausgegeben und der Zähler der Temperaturen inkrementiert.

```
1           try {
2               // Persist tempertures in DynamoDB
3               dbUtils.putTemperatures(tableName, allTemperatures, timestamp);
4               checkpointer.checkpoint();
5           } catch (Exception e) {
6               log.error(
7                       "Error while trying to checkpoint during ProcessRecords",
8                       e);
9           }
```

Quelltext 3.14: TemperatureConsumer.java (ll. 157-165): Persistieren der Temperaturen auf DynamoDB

Nach dem Lesen aller Daten dieses Durchlaufs werden die gesammelten Temperaturdaten in der DynamoDB Tabelle, wie in Quelltext 3.14 zu sehen, persistiert. Außerdem wird der Checkpointer aufgerufen und ein Eintrag in der Checkpointer Tabelle erzeugt.

```
1       Table table = dynamoDB.getTable(tableName);
2
3       for (String sensor : temperatureMap.keySet()) {
4           QuerySpec spec = new QuerySpec().withHashKey(
```

```
5                   ATTRIBUTE_NAME_HASH_KEY, sensor).withRangeKeyCondition(
6                   new RangeKeyCondition(ATTRIBUTE_NAME_RANGE_KEY).eq(String
7                           .valueOf(timestamp)));
8
9           ItemCollection<QueryOutcome> items = table.query(spec);
10
11          Iterator<Item> iterator = items.iterator();
12          Item item = null;
13          Map<String, String> temperatures = null;
14          while (iterator.hasNext()) {
15              item = iterator.next();
16              temperatures = item.getMap(ATTRIBUTE_NAME_TEMPERATURE);
17          }
18
19          if (temperatures == null) {
20              temperatures = new HashMap<>();
21          }
22          temperatures.putAll(temperatureMap.get(sensor));
23          table.putItem(new Item()
24                  .withPrimaryKey(ATTRIBUTE_NAME_HASH_KEY, sensor,
25                          ATTRIBUTE_NAME_RANGE_KEY, String.valueOf(timestamp))
26                  .withMap(ATTRIBUTE_NAME_TEMPERATURE, temperatures));
27          System.out.println("PutItem succeeded!");
```

Quelltext 3.15: DynamoDBUtils.java (ll. 158-184): Persistierung der Daten auf DynamoDB

Der Quelltext 3.15 zeigt die Methode “putTemperatures” in der Klasse “DynamoDBUtils”, die im Consumer aufgerufen wird. Hier wird jeder einzelne Sensor einzeln in einer For-Schleife abgearbeitet. Für jeden Sensor wird eine Query auf der Tabelle ausgeführt, die die Temperaturdaten eines Sensordurchlaufs für den gegebenen Sensornamen und den Timestamp zurückgibt. Das Ergebnis ist eine Liste von Maps, die entweder genau eine Map enthält oder keine, falls es noch keine Temperaturdaten für diesen Sensordurchlauf gibt. Dann wird eine neue Map erzeugt. Alle der Methode übergebenen Temperaturen werden in die Map eingefügt. Daraufhin wird die Map wieder in die DynamoDB Tabelle eingefügt und diese Tabelle persistiert. Damit sind die Temperaturen auf DynamoDB aktualisiert.
Die Methode “processRecords” wird vom “IRecordProcessor” je nach Datenmenge mehrfach mit einem Teil der Daten, der aus dem Stream gelesen wurde, als Parameter aufgerufen. Wenn alle Daten gelesen wurden, bleibt der Consumer zunächst in einem Wartezustand, bis weitere Daten gelesen werden können.

3.4 Webapplikation

Die Webapplikation besteht aus einem Servlet sowie der Klasse Servletstarter, die einen Jetty Server startet und das Servlet einbindet. Auf der Website können dann die aktuellen Temperaturwerte eingesehen werden, die aus der DynamoDB-Tabelle ausgelesen werden.

```
1        if (args.length == 2) {
2            streamName = args[0];
3            db_name = args[1];
4        }
5
6        Server server = new Server(8080);
```

Quelltext 3.16: ServletStarter.java (ll. 34-39): Auslesen der übergebenen Argumente in ServletStarter

In Quelltext 3.16 werden die übergebenen Parameter ausgelesen und initialisiert. Zudem wird eine Server Instanz erzeugt, die auf dem Port 8080 betrieben wird.

```
1        ServletContextHandler context = new ServletContextHandler(
2                ServletContextHandler.NO_SESSIONS
3                        | ServletContextHandler.NO_SECURITY);
4        context.setContextPath("/api");
5        context.addServlet(new ServletHolder(new TemperatureServlet(streamName,
6                db_name, tableName)), "/GetTemperature/*");
```

Quelltext 3.17: ServletStarter.java (ll. 41-46): Servlet wird in Context gesetzt

In Quelltext 3.17 wird der Context initialisiert und das Servlet wird in den Context eingebunden. Das Servlet ist also nach Start des Server unter der Adresse “localhost:8080/api/GetTemperature” erreichbar.

```
1        HandlerList handlers = new HandlerList();
2        handlers.addHandler(context);
3        handlers.addHandler(new DefaultHandler());
4
5        server.setHandler(handlers);
6        server.start();
7        server.join();
```

Quelltext 3.18: ServletStarter.java (ll. 48-54): Context wird den Handlern zugefügt

In Quelltext 3.18 wird der Context dann den Handlern zugefügt, die Handler dem Server zugeordnet und der Server dann gestartet.
Das Servlet selbst besteht aus einer doGet-Methode, die bei einem GET auf die oben genannte URL aufgerufen wird.

```
1       HashMap<String, HashMap<String, HashMap<String, Object>>> allTemperatures = new HashMap
            <>();
2       if (dbUtils.doesTableExist(tableName)) {
3           allTemperatures = dbUtils.getAllSensorTemperatures(tableName);
4       }
```

Quelltext 3.19: TemperatureServlet.java (ll. 64-67): Aufruf der Methode zum Lesen aller Temperaturen von DynamoDB

Quelltext 3.19 zeigt die Initialisierung einer Hashmap, die alle Temperaturdaten enthalten wird, sowie den Aufruf der Methode “getAllSensorTemperatures”, die die HashMap befüllt.

```
1   public HashMap<String, HashMap<String, HashMap<String, Object>>> getAllSensorTemperatures(
2           String tableName) {
3       ScanRequest scanRequest = new ScanRequest().withTableName(tableName);
4
5       ScanResult result = client.scan(scanRequest);
6       HashMap<String, HashMap<String, HashMap<String, Object>>> allTemperatures = new HashMap
            <>();
7       for (Map<String, AttributeValue> item : result.getItems()) {
8           String sensorName = item.get(ATTRIBUTE_NAME_HASH_KEY).getS();
9           HashMap<String, HashMap<String, Object>> currentHashMap = getTemperaturesForSensor(
10              sensorName, tableName);
11          allTemperatures.put(sensorName, currentHashMap);
12      }
13
14      return allTemperatures;
15  }
```

Quelltext 3.20: DynamoDBUtils.java (ll. 228-242): Die Methode getAllSensorTemperatures

Quelltext 3.20 zeigt die Methode “getAllSensorTemperatures”, die die oben genannte HashMap erzeugt. In der Methode werden zunächst mittels eines “Scanrequests” die Namen aller in der DynamoDB Tabelle vorhandenen Sensoren ermittelt. Dann wird für jeden Sensor eine eigene Hashmap erzeugt, die die Temperaturdaten enthält.

```
1   public HashMap<String, HashMap<String, Object>> getTemperaturesForSensor(
2           String sensor, String tableName) {
3       Table table = dynamoDB.getTable(tableName);
4
5       QuerySpec spec = new QuerySpec().withHashKey(ATTRIBUTE_NAME_HASH_KEY,
6               sensor);
7
8       ItemCollection<QueryOutcome> items = table.query(spec);
9
10      Iterator<Item> iterator = items.iterator();
11      Item item = null;
12      HashMap<String, HashMap<String, Object>> temperatureMap = new HashMap<>();
13      while (iterator.hasNext()) {
14          item = iterator.next();
15          temperatureMap.put(item.getString(ATTRIBUTE_NAME_RANGE_KEY),
16                  new HashMap<>(item.getMap(ATTRIBUTE_NAME_TEMPERATURE)));
17      }
18
19      return temperatureMap;
20  }
```

Quelltext 3.21: DynamoDBUtils.java (ll. 198-217): Die Methode getTemperaturesForSensor

Die Hashmaps der einzelnen Sensoren werden in der Methode “getTemperaturesForSensor” erzeugt, die in Quelltext 3.21 zu sehen ist. Hier wird zunächst eine Query mit dem Hashkey als Parameter abgesetzt, die alle Temperaturen für einen Sensor zurückliefert. Daraufhin wird über die Items der Query iteriert und die HashMap der Temperaturdaten dieses Sensors erzeugt. In der Methode “getAllSensorTemperatures” werden alle Hashmaps der Sensoren in einer übergeordneten Hashmap zusammengefügt.
Die daraus generierte Hashmap zeigt auf eine weitere Menge von Hashmaps, die wiederum auf eine Menge von HashMaps zeigt. Die Temperaturdaten sind also schichtenweise aufgeteilt. Die erste Schicht ist die erste Hashmap, deren Keys die Sensornamen sind. Die Sensornamen zeigen jeweils auf eine weitere HashMap. Dessen Key ist der Timestamp des Laufs dieses Sensors. Wurde ein Sensor also mehrmals gestartet, hat dieser mehrere Einträge in dieser HashMap. Die Values dieser HashMap sind jeweils eine weitere HashMap. Diese enthält als Keys den Timestamp, an dem die Temperatur gemessen wurde und als Values den entsprechenden Temperaturwert. Geht man diese Hierarchie von oben nach unten durch, werden also den Sensoren eine Menge von Timestamps der Iterationen zugeordnet. Den Timestamps der Iterationen wird nun eine Menge von Temperaturen zugeordnet, die zudem jeder einen Timestamp enthalten, um sie in eine zeitliche Reihenfolge zu bringen.
In der Methode wird nun über die einzelnen Schichten dieser HashMap iteriert und da-

bei mittels einem PrintWriter das erzeugte HTML ausgegeben. Es wird CanvasJS [29] genutzt, um die Temperaturdaten als Graphen darzustellen.

```
1            int sensorCount = 0;
2            for (String sensor : allTemperatures.keySet()) {
3                HashMap<String, HashMap<String, Object>> hashMap = allTemperatures
4                        .get(sensor);
5                for (String timestamp : hashMap.keySet()) {
6                    int count = 0;
7                    out.println("var dataPoints" + sensorCount + " = [];");
8                    HashMap<String, Object> tempHashMap = hashMap
9                            .get(timestamp);
10                   for (String temperatureKey : tempHashMap.keySet()) {
11                       out.println("dataPoints" + sensorCount + ".push({ x: "
12                               + count + ", y: "
13                               + tempHashMap.get(temperatureKey) + "});");
14                       count++;
15                   }
```

Quelltext 3.22: TemperatureServlet.java (ll. 75-89): Iteration über die HashMaps

Quelltext 3.22 zeigt die Iteration über die HashMaps. CanvasJS nimmt ein Array an, das die Datenpunkte als y und y Werte enthält. Hier wird für jeden Sensordurchlauf ein eigenes Array erzeugt, da ja jeder Sensordurchlauf in einem eigenene Graphen dargestellt werden soll. In der innersten For-Schleife werden dem Array dann die Datenpunkte hinzugefügt. Als x Koordinate wird eine laufende Nummer vergeben, da die Timestamps der Temperaturen zu groß für die Anzeige wären und als y Koordinate wird die Temperatur gesetzt.
Wenn alle Datapoint-Arrays erzeugt wurden, werden weitere Metadaten für die Generierung der Graphen ausgegeben und es wird zudem für jeden Graphen ein div Element erzeugt, in das der Graph dann generiert wird. Bei Aufrufen der Webseite wird dann eine Javascript Funktion aufgerufen, die die Graphen in den divs rendert.
Die Webseite stellt nun alle Durchläufe der Sensoren dar. Für jeden Durchlauf gibt es einen Graphen, de wie oben beschrieben initialisiert wurde. CanvasJS bietet nun die Möglichkeit die Graphen zu manipulieren indem bspw. der betrachtete Zeitraum verändert wird und nur ein Teil der Daten dargestellt wird.

3.5 Löschung der AWS Ressourcen

Eine weitere Klasse ist die Klasse “DeleteResources”, mittels der man die gestarteten Ressourcen auf Amazon Webservices wieder löschen kann, um keine weiteren Kosten zu verursachen.
Die Klasse hat eine main-Methode, die zwei Argumente annimmt: Den Streamnamen sowie den Datenbank Namen der Dynamo DB Tabelle.

```
1       String streamName = TemperatureProducer.streamName;
2       String db_name = TemperatureConsumer.db_name;
3
4       if (args.length == 2) {
5          streamName = args[0];
6          db_name = args[1];
7       }
```

Quelltext 3.23: DeleteResources.java (ll. 24-30): Auslesen der übergebenen Argumente in DeleteResources

In Quelltext 3.23 sieht man, wie die Argumente ausgelesen und gesetzt werden. Wenn nicht genau 2 Argumente übergeben werden, wird ein Standardwert für die beiden Variablen genutzt.

```
1       Region region = RegionUtils.getRegion(TemperatureProducer.REGION);
2       AWSCredentialsProvider credentialsProvider = new DefaultAWSCredentialsProviderChain();
3       AmazonDynamoDB amazonDynamoDB = new AmazonDynamoDBClient(
4             credentialsProvider, new ClientConfiguration());
5       AmazonDynamoDBClient client = new AmazonDynamoDBClient();
6       client.setRegion(region);
7       DynamoDB dynamoDB = new DynamoDB(client);
8       amazonDynamoDB.setRegion(region);
9       DynamoDBUtils dbUtils = new DynamoDBUtils(dynamoDB, amazonDynamoDB,
10            client);
11      dbUtils.deleteTable(db_name);
12      dbUtils.deleteTable(TemperatureConsumer.tableName);
```

Quelltext 3.24: DeleteResources.java (ll. 32-43): Initialisierung der Dynamo DB Utilklasse und Löschen der Tabellen

In Quelltext 3.24 wird die DynamoDB Utilklasse initalisiert und dazu werden zunächst die benötigten Amazon Client Klassen erzeugt, die der Utilklasse bei der Initialisierung übergeben werden. Daraufhin wird die Übersichtstabelle sowie die Tabelle, die die Temperaturdaten enthält, gelöscht.

```
1        AmazonKinesis kinesis = new AmazonKinesisClient(credentialsProvider,
2                new ClientConfiguration());
3        kinesis.setRegion(region);
4        StreamUtils streamUtils = new StreamUtils(kinesis);
5        streamUtils.deleteStream(streamName);
```

Quelltext 3.25: DeleteResources.java: Initialisierung der Stream Utilklasse und Löschen des Streams (ll. 45-49)

Im nächsten Abschnitt in Quelltext 3.25 wird dann die Stream Utilklasse initialisiert und daraufhin der Stream gelöscht.
Damit sind alle Ressourcen, die auf Amazon Web Services genutzt wurden, gelöscht und es werden keine Kosten mehr zum Beispiel durch die temporäre Speicherung der Daten im Kinesis Stream verursacht.

3.6 Docker

Um die Applikation auf ein Deployment auf Amazon EC2 vorzubereiten, wurde ein Docker Image erstellt, um die Applikation letztendlich über ECS auf EC2 zu deployen.

```
1 FROM maven:3.2-jdk-7-onbuild
2 CMD ["mvn clean package"]
```

Quelltext 3.26: Dockerfile des Projekts

Dieses Dockerfile ist sehr simpel gehalten und enthält nur 2 Zeilen. Wie in Quelltext 3.26 zu sehen, basiert dieses Image auf dem Apache Maven Image [26]. Dieses Image enthält neben einer Java 7 Installation auch eine Maven 3.2 Installation. Zudem kopiert es bei der Generierung des Images das Projekt, wenn die Dockerfile im gleichen Ordner wie die pom.xml ist, nach “/usr/src/app” und führt “mvn install” aus. Zusätzlich dazu wird, wie in Zeile 2 beschrieben, noch “mvn clean package” ausgeführt, was die Applikation endgültig initialisiert.
Mithilfe dieses Dockerfiles kann nun mit “docker build -t project-image .” ein Docker Image erzeugt werden. Über dieses Image kann dann mit dem Befehl in Quelltext 3.27 der Producer gestartet werden.

```
1 docker run -it --name project-image project-container bash -c 'MAVEN_OPTS="-Daws.
      accessKeyId=ACCESSKEY␣Daws.secretKey=SECRETKEY␣-Dstream.name=StreamTest␣-
      Dsensor.name=SensorTest␣-Drun.seconds=10␣-Drecord.second=1''␣mvn␣compile␣-Pproducer
      ␣exec:java'
```

Quelltext 3.27: Konsolenbefehl zum Starten des Docker Images

Auch hier müssen wieder der AWS Access Key sowie der AWS Secret Key übergeben werden sowie die weiteren Parameter, die der Producer annimmt. In diesem Beispiel wird ein Stream namens “StreamTest” erzeugt, der Sensor Name ist “SensorTest” und der Producer erzeugt für 10 Sekunden 10 Records pro Sekunde.
Es ist natürlich auch möglich die anderen Klassen mithilfe dieses Images zu starten, indem die Übergabeparameter geändert werden.

3.7 Abweichungen im Vergleich zur Projektplanung

In der Projektplanung wurden die Anforderungen des Projekts ermittelt und darauf basierend ein Pflichtenheft erstellt, das die Anforderungen an das Projekt präzise darstellt. Einige dieser Anforderungen konnten aber im Laufe dieses Projektes nicht erfüllt werden. Zum einen konnte die Anforderung “Mehrere AWS Services” nicht erfüllt werden. Während der Entwicklung wurde klar, dass AWS Kinesis eine sehr effiziente und schnelle Lösung für die Übertragung der Daten ist und es zumindest von Amazon keine passende Alternative gibt, die für dieses Projekt genutzt werden konnte. Es wäre möglich gewesen, Amazon Kinesis Firehose [11] zu nutzen, das eine Variante von Kinesis darstellt, um Daten über Kinesis direkt an andere Datenbankservices wie Amazon S3 [12] weiterzuleiten, wo sie dann persistiert werden. Allerdings bietet Amazon Kinesis Firehose keine Integration für Amazon DynamoDB, das in diesem Projekt das gewählte Datenbanksystem darstellt. Eine Umstellung hätte zu viel Zeit gekostet. Genauso war für eine Eigenentwicklung, die beispielsweise auf Amazon EC2 hätte deployed werden können, am Ende des Projektes keine Zeit mehr. Daher blieb es bei der einen implementierten Variante.
Die Anforderung “Dashboard Start Stopp” wurde auch nicht komplett umgesetzt. Die

Producer lassen sich zwar stoppen und wieder starten, allerdings nicht über das Dashboard, sondern indem der EC2 Container gestoppt wird, in dem der Producer betrieben wird. “Dashboard Zeitskalierung” wurde auch nicht ganz umgesetzt, es ist nur möglich beim Start eines Producers die Menge der zu erzeugenden Daten zu bestimmen und nicht, wenn ein Producer schon gestartet ist.
Des weiteren wurden die Anforderungen “Dashboard Diagramme Aktualität” sowie “Producer Temperatur Beeinflussung” nicht vollständig umgesetzt. Die Diagramme sind beim Aufruf des Dashboards aktuell, werden aber nicht aktualisiert, wenn die Seite nicht aktualisiert wird. Zudem beeinflussen sich die Producer nicht gegenseitig.
Dementsprechend wurden die ursprünglichen Anforderungen “Start und Stopp des Datenstroms”, “Mehrere AWS Services”, “Aktualität der Daten” sowie “Temperaturen beeinflussen sich gegenseitig” nicht oder nur teilweise erfüllt.
In der Projektanalyse wurde zudem eine Risikoanalyse durchgeführt, die die Risiken des Projektes auflistet. Im Projekt sind die Risiken Nummer 3 und 4 zumindest teilweise auch eingetreten. Der Umfang des Projektes wurde etwas unterschätzt und dadurch wurden Teile des Projekts nicht fertig. Allerdings wurden im Rahmen der Risikoanalyse auch Gegenmaßnahmen entwickelt, um die Kosten der Risiken abzufedern. So wurde die Wichtigkeit der Anforderungen ermittelt und dementsprechend priorisiert, so dass die wichtigsten Anforderungen möglichst früh abgearbeitet wurden. Dabei wurde auch erkannt, dass einige Anforderungen für den Erfolg des Projektes nicht entscheidend sind und das Projekt nur abgerundet hätten. Daher konnte das Projekt trotz dem Eintritt zweier Risiken erfolgreich durchgeführt werden.
Das Budget von 200€ wurde innerhalb des Projektes nicht ausgereizt, da auch entsprechend der Risikoplanung die Kosten ständig überwacht wurden.
Zusammenfassend kann man also sagen, dass es einige Abweichungen im Vergleich zur Projektplanung gab, aber trotzdem alle wichtigen Anforderungen umgesetzt wurden und die Maßnahmen im Rahmen der Risikoplanung gegriffen haben.

3.8 Zusammenfassung

Insgesamt kann man sagen, dass das Projekt zufriedenstellend abgelaufen ist. Es sind zwar einige Anforderungen nicht erfüllt worden, allerdings wurden alle für das Projekt wichtigen Anforderungen erfüllt und die genutzten Amazon Webservices können evaluiert werden. Die nicht umgesetzten Anforderungen können unter Umständen in einem

weiteren Projekt nach dieser Bachelorarbeit erarbeitet werden, um noch weitere Kenntnisse über Amazon Web Services zu gewinnen und das Projekt in einigen Punkten noch zu verbessern.
Positiv ist die Planungsphase und insbesondere die Risikoplanung zu sehen, dank der die wichtigsten Anforderungen erfüllt werden konnten und auch das Budget eingehalten werden konnte.
Schade ist, dass die Amazon Kinesis Connectors [10] nicht genutzt werden konnten, da die Dokumentation dieser nur recht kurz ist und zu viel Einarbeitungszeit benötigt hätte. Mittels Kinesis Connectors hätte man die Persistierung der Temperaturdaten auf DynamoDB automatisieren können, so dass gar kein Consumer mehr nötig gewesen wäre, sondern die Daten direkt von Kinesis an DynamoDB weitergeleitet werden.
Zusammenfassend ist das Projekt also zufriedenstellend erfüllt worden.

4. AWS IoT

IoT ist ein Thema, das immer bedeutender wird in der heutigen Zeit, gerade durch das immer mehr aufkommende Thema Smart Home. Dieses Kapitel wird eine kurze Einführung in das Thema IoT geben und dann die Möglichkeiten von AWS IoT beschreiben.

4.1 Einführung IoT

Nach der Einschätzung von Daniele Miorandi u.a. [2] gibt es im Moment eine Weiterentwicklung des Internets, das nun nicht mehr nur Endbenutzergeräte miteinander verbindet, sondern auch ganz alltägliche Objekte verbindet, die miteinander kommunizieren oder auch mit dem Menschen kommunizieren und so ganz neue Möglichkeiten dem Endbenutzer geben. Dies wird als Internet of Things (IoT) bezeichnet. Das Internet of Things basiert auf 3 Punkten, die alltägliche Objekte nun können müssen, um Teil des Internet of Things zu sein:

1. Eindeutig identifizierbar sein (Jedes Objekt beschreibt sich selbst, z.b. mittels RFID)
2. Kommunikationsfähig sein
3. Interaktionsfähig sein, mit anderen Objekten oder mit Nutzern

Teil des Internet (of Things) werden nun also nicht mehr nur Rechner unterschiedlichster Art (PC, Smartphone etc.), sondern jeder alltägliche Gegenstand wird Teil des Internets und kann mit anderen Objekten kommunizieren. So wären in diesem Projekt die Sensoren Teil des Internet of Things, da sie die oben genannten 3 Punkte erfüllen und nicht nur isoliert für sich selbst funktionieren.
Das Internet of Things bietet also für die Zukunft viele Möglichkeiten, die Funktionen von

alltäglichen Gegenständen des Lebens zu erweitern. So können bspw. die Temperatursensoren mit der Klimaanlage kommunizieren, die daraufhin die Temperatur reguliert. Lampen können mittels eines Smartphones auch außerhalb des Hauses gesteuert werden oder regulieren sich selbst, wenn sie die Nachricht eines Lichtsensors empfangen, dass es im Moment zu dunkel oder zu hell ist.
Auch in der Industrie wird das Internet of Things immer mehr Einzug halten und Produktionsprozesse weiter automatisieren, da Produktionsmaschinen nun auch mit anderen Geräten kommunizieren können.
Großes Thema im Bereich IoT ist die Sicherheit, besonders da nun auch alltägliche Geräte Teil des Internet of Things werden. Dementsprechend wäre es möglich, dass sich eine Person Zugriff auf nahezu alle Geräte innerhalb eines Haushalts verschaffen könnte, ohne dass der Besitzer davon etwas weiß. Daher sind sichere Authentifizierungsverfahren und Kommunikationswege ein wichtiger Teil des IoT.
Insgesamt kann man also sagen, dass das Internet of Things sicherlich einer der nächsten großen Schritte in der IT sein wird und in einigen Jahren viele Objekte im Haushalt und in Firmen Teil des IoT sein werden. Die wohl größte Herausforderung wird das Thema Sicherheit sein, um die Sicherheit und Privatsphäre von Nutzern von Objekten im IoT zu schützen.

4.2 Einführung AWS IoT

AWS IoT [16] ist einer der neueren Web Services von Amazon, welcher zunächst im Oktober 2015 in eine Beta Phase gestartet ist und seit Januar 2016 in den regulären Betrieb gewechselt ist. Es ist eine verwaltete Cloud Plattform, mit der verbundene Geräte mit Cloud Anwendungen und anderen Geräten zusammenarbeiten können.
AWS IoT kann Millionen von Nachrichten von typischen IoT Geräten wie bspw. Sensoren annehmen und an weitere Amazon Web Services oder andere IoT Geräte verteilen. Dies bietet zum einen die Möglichkeit der Kommunikation von IoT Geräten untereinander sowie die Möglichkeit der IoT Geräte mit der Cloud zu kommunizieren um beispielsweise Daten zu übermitteln, die dann in Cloud Services verarbeitet und/oder persistiert werden. Beispiele wären zum Beispiel ein Temperatursensor, der mittels AWS IoT der Klimaanlage des Hauses die Nachricht übermittelt, dass die Temperatur höher geregelt werden muss oder der Temperatursensor schickt seine Temperaturdaten an andere Amazon Web Services wie bspw. Amazon DynamoDB, welches die Daten dann persis-

tieren kann. Amazon nennt als Beispiel eine Reihe von Temperatursensoren, die ihre Daten an AWS IoT senden. AWS IoT sendet dann bei Überschreiten eines bestimmten Grenzwertes ein Kommando an einen Ventilator im Haus, der sich daraufhin einschaltet. Der Ventilator kann aber auch bspw. über eine Mobilapplikation von einem Benutzer über AWS IoT manuell gestartet werden (s. Abbildung 4.1). Dementsprechend ist AWS

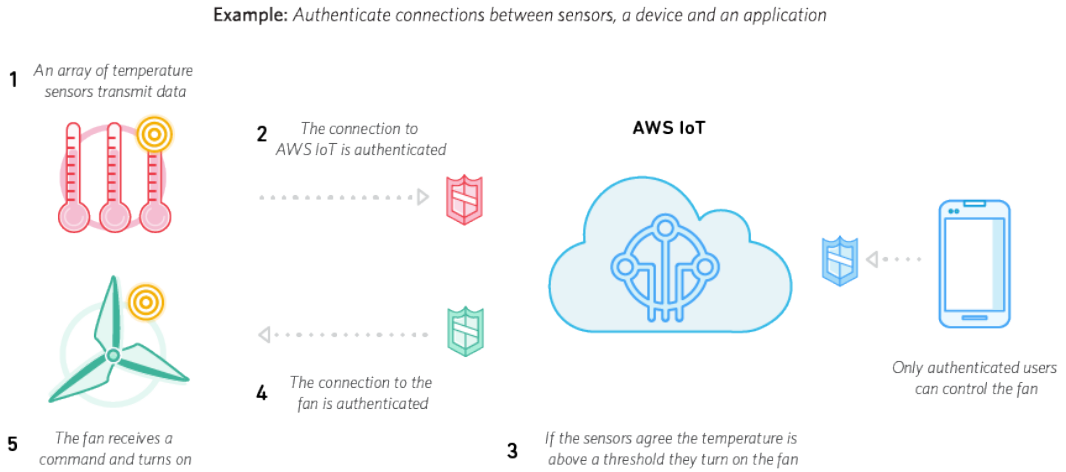

Abbildung 4.1: Kommunikation und Authentifikation von Temperatursensoren und einem Ventilator mit AWS IoT. Quelle: [16]

IoT eine gute Alternative, sollte man dieses Projekt mit echten Temperatursensoren ausführen wollen.

4.3 Funktionsweise

Ein wichtiger Punkt bei AWS IoT ist die Sicherheit der Kommunikation zwischen den einzelnen Endpunkten. Sicherheit ist bei IoT generell ein großes Thema, da unter Umständen mit sensiblen Daten gearbeitet wird, die im eigenen Haus erzeugt werden. AWS IoT stellt sicher, dass keine Kommunikation über AWS IoT unverschlüsselt stattfindet, indem sich jeder Endpunkt zunächst bei AWS IoT authentifizieren muss. Jede Kommunikation wird einzeln authentifiziert und verschlüsselt. Dies sieht man z.B. in Abbildung 4.1. Hier wird in Schritt 2 die Kommunikation der Temperatursensoren mit AWS IoT authentifiziert, bevor sie die Daten an AWS IoT senden. In Schritt 4 wird die Kommunikation mit dem Ventilator ebenfalls zunächst authentifiziert. Und natürlich können nur

authentifizierte Nutzer über Mobilapplikationen über AWS IoT wie in diesem Beispiel den Ventilator kontrollieren.

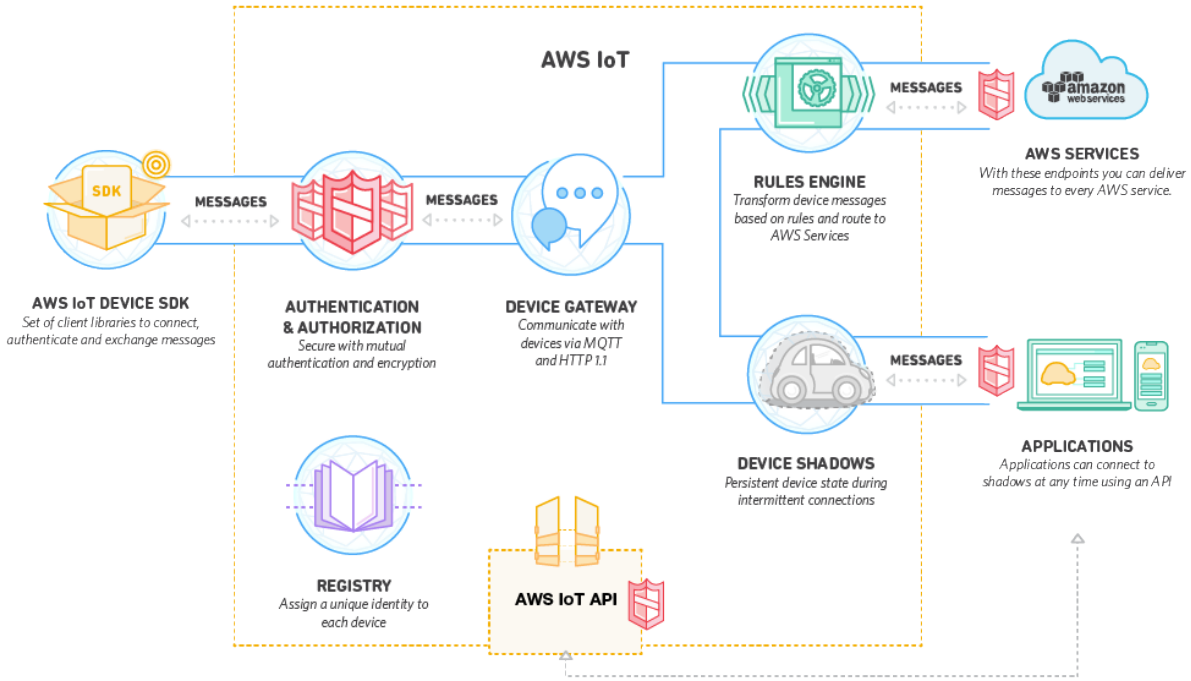

Abbildung 4.2: Funktionsübersicht von AWS IoT. Quelle: [17]

Abbildung 4.2 zeigt eine Funktionsübersicht von AWS IoT. Hier steht zunächst einmal auf der linken Seite die AWS Device SDK, die eine Reihe von Client Libraries bereitstellt, mit der Geräte mit AWS IoT kommunizieren können. Die Kommunikation findet dabei über die Protokolle MQTT [22] oder HTTP statt. MQTT ist ein leichtgewichtiges Nachrichtenprotokoll für die direkte Kommunikation zwischen Maschinen (M2M) und ist daher ein häufig genutztes Protokoll im Bereich des IoT. Mittels einem dieser beiden Protokolle kommuniziert das Gerät, das die AWS Service SDK nutzt, mit dem Device Gateway. Die Kommunikation wird dabei verschlüsselt und zunächst müssen sich beide Seiten auch authentifizieren, bevor überhaupt eine Kommunikation zustande kommt. AWS IoT unterstützt dabei die AWS-Methode der Authentifizierung (mit der Bezeichnung “SigV4”) sowie eine Authentifizierung auf der Basis von X.509 [17]. Den Geräten können einzelne Richtlinien vorgegeben werden, die ihren Zugriff entsprechend einschränken oder es können Rollen definiert werden, die Geräten eine genau spezifizierte Menge an Rechten geben.

Die Registry erstellt eine eindeutige Identität für die Geräte. Diese ist für alle Geräte egal welcher Art einheitlich formatiert. Außerdem werden Metadaten von Geräten gespeichert, die bspw. angeben, welche Funktionen dieses Gerät unterstützt wie zum Beispiel, dass ein Sensor Temperaturdaten meldet und in welcher Einheit diese Temperaturdaten übermittelt werden.
Das Device Gateway stellt für Applikationen REST APIs bereit, über die die Applikationen die Statusinformationen von Geräten auslesen und manipulieren können. Dazu legt das Device Gateway sogenannte “Device Shadows” bzw. “Schattengeräte” an, die den letzten Zustand des Gerätes darstellen. Das heißt, dass der Zustand von Geräten auch dann ausgelesen werden kann, wenn das Gerät gar nicht mehr online ist, da der letzte Zustand als “Schattengerät” gespeichert wurde. Zudem können Applikationen so auch den gewünschten zukünftigen Zustand festlegen. Auch dieser wird im “Schattengeräte” persistiert und wird auf das Gerät übertragen, wenn es wieder online ist. Damit ist es möglich, auch für Geräte, die nicht dauerhaft online sind, eine REST API bereitzustellen, die dauerhaft verfügbar ist und Applikationen ermöglicht, sich immer mit dem “Schattengerät” zu synchronisieren und den Status zu verändern. Die Kommunikation erfolgt auch hier wieder über verschlüsselte Nachrichten. Das Device Gateway kann

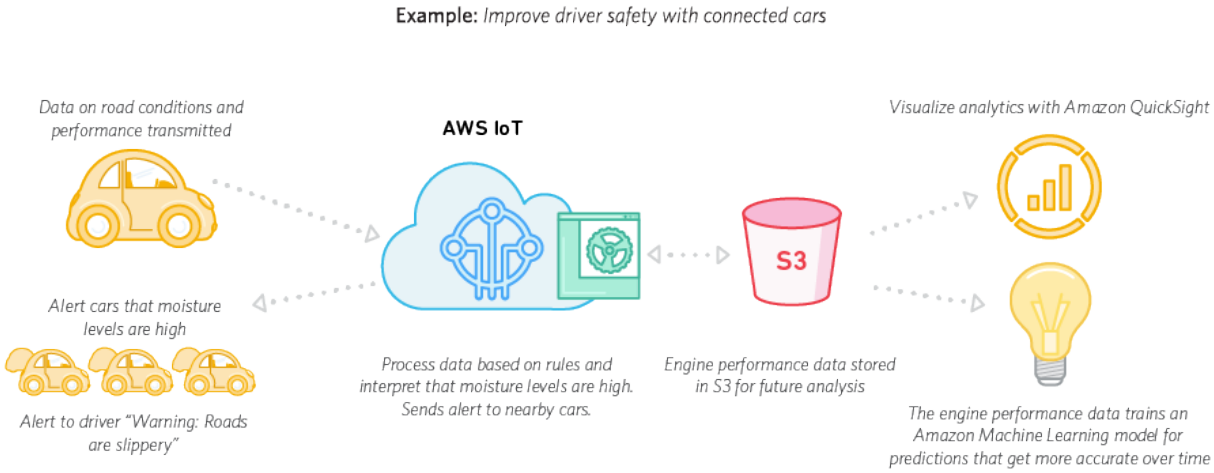

Abbildung 4.3: Beispiel der Kommunikation von AWS IoT mit anderen Amazon Services wie S3. Quelle: [16]

aber auch über die Rules Engine Daten von Geräten transformieren und an andere Services weiterleiten. So könnten beispielsweise Temperaturdaten an Amazon Dyna-

moDB oder andere Datenbank Services weitergeleitet werden. Die Abbildung **??** zeigt die Kommunikation von AWS IoT mit Amazon S3, bei der Performance Daten von Autos über AWS IoT an S3 übermittelt und zur weiteren Auswertung gespeichert werden.
Es können aber auch Regeln erstellt werden, die bei bestimmten Konditionen Aktionen auslösen. Zum Beispiel könnte beim Überschreiten einer bestimmten Menge an Temperaturen eine Nachricht an AWS Lambda [15] geschickt werden, das daraufhin den Mittelwert der gesammelten Daten berechnet. Nachrichten können aber auch an Geräte geschickt werden wie in Abbildung 4.1. Erreicht die Temperatur der Sensoren einen Grenzwert, greift eine Regel und sendet den Befehl an den Ventilator, der sich daraufhin einschaltet.
Die Regeln werden direkt in der Management Konsole eingetragen oder über eine SQL-ähnliche Syntax definiert.
AWS IoT bietet also passende Funktionen, um das Projekt mit echten Temperatursensoren umzusetzen. Man könnte die Temperaturdaten direkt an einen Datenbank Service wie Amazon DynamoDB weiterleiten und auch die Temperatursensoren über die REST API direkt steuern. Mittels der Rules Engine könnten Regeln festgelegt werden, nach denen bei bestimmten Temperaturen bestimmte Aktionen ausgeführt werden können. Dadurch könnte man auch andere Geräte wie beispielsweise Ventilatoren oder Klimaanlagen einbinden und mittels AWS IoT steuern lassen. Man benötigt aber neben den Sensoren auch Mikrocontroller wie bspw. Raspberry PIs [23], die mit den Sensoren und anderen Geräten kommunizieren können, um AWS IoT nutzen zu können.

4.4 AWS IoT CLI, SDK und APIs

Die AWS IoT CLI bietet die Möglichkeit über Kommandozeileneingaben verschiedene Funktionen von AWS IoT zu nutzen. Beispielsweise können Geräte zur Device Registry hinzugefügt werden und Einträge in der Registry verändert werden.

```
1 aws iot describe-thing --thing-name "MyDevice3"
```

Quelltext 4.1: Einfügen eines Geräts in die Registry. Quelle: [18]

Der Quellcode 4.1 zeigt den Befehl, mit dem ein Gerät mit dem Namen “MyDevice3” in die Registry eingefügt werden kann.

```
{
    "thingName": "␣MyDevice3",
    " defaultClientId ": "MyDevice3",
    " attributes ": {
        "Manufacturer": "Amazon",
        "Type": "IoT␣Device␣A",
        "Serial␣Number": "10293847562912"
    }
}
```

Quelltext 4.2: Geräteeintrag in der Registry. Quelle: [18]

Der Eintrag in der Registry sieht dann wie in Quelltext 4.2 aus. Weitere Attribute wie eine einzigartige Seriennummer werden automatisch generiert.
Die AWS IoT CLI bietet zudem auch die Möglichkeit alle Geräte in der Registry anzuzeigen oder die Suchergebnisse zu filtern.
Mittels SDK lassen sich auch Regeln erstellen, einsehen und bearbeiten.

```
aws iot create-topic-rule --rule-name my-rule --topic-rule-payload file://my-rule.json
```

Quelltext 4.3: Erstellen einer AWS IoT Regel. Quelle: [19]

In Quelltext 4.3 wird eine Regel mit dem Namen “my-rule” erzeugt. Zudem wird eine payload Datei namens “my-rule.json” ebenfalls mitgeschickt.

```
{
  "sql": "SELECT␣*␣FROM␣'iot/test'",
  "ruleDisabled": false,
  "actions": [{
      "dynamoDB": {
          "tableName": "my-dynamodb-table",
          "roleArn": "arn:aws:iam::123456789012:role/my-iot-role",
          "hashKeyField": "topic",
          "hashKeyValue": "${topic(2)}",
          "rangeKeyField": "timestamp",
          "rangeKeyValue": "${timestamp()}"
      }
  }]
}
```

Quelltext 4.4: Inhalt einer Payload Datei. Quelle: [19]

Die Payload Datei in Quelltext 4.4 enthält die Regel, alle Nachrichten, die an die Topic “iot-topic” gesendet werden, in die DynamoDB Tabelle “my-dynamodb-table” zu schreiben. Dabei wird noch eine Rolle übergeben, die in diesem Fall das Recht hat, auf die DynamoDB Tabelle zuzugreifen und in diese zu schreiben. Als HashkeyValue wird der Name der Topic übergeben und als rangeKeyValue der Timestamp der Nachricht.

```
1 HTTP GET https://endpoint/things/thingName/shadow
```

Quelltext 4.5: GET Anfrage an einen “Geräteschatten”. Quelle: [20]

Mittels einer REST API können die “Geräteschatten” von Geräten abgefragt und auch manipuliert werden. Quelltext 4.5 zeigt eine HTTP GET Anfrage an einen Geräteschatten, wobei “endpoint” und “thingName” Variablen sind, die man in der AWS IoT Konsole auslesen kann oder auch per AWS IoT SDK.

```
1 HTTP 200
2 BODY: response state document
```

Quelltext 4.6: GET Antwort auf die Anfrage an einen “Geräteschatten”. Quelle: [20]

Quelltext 4.6 zeigt die Antwort auf eine Anfrage an einen “Geräteschatten”. “response state document” ist hierbei ein größeres Dokument, das Informationen über den Status des “Geräteschattens” sowie Metadaten, einen Timestamp und andere Informationen enthält.
Über die API ist es auch möglich, “Geräteschatten” zu bearbeiten und zum Beispiel einen neuen Status zu setzen, der dann im Gerät gesetzt wird, wenn es online ist oder auch “Geräteschatten” zu löschen.
Es gibt außerdem noch eine AWS IoT SDK [21], die die Integration von Geräten in AWS IoT vereinfacht. Die SDK ist für C, Arduino Yún [30] und Node.js [28] verfügbar.
Sie bietet Funktionen zum Anmelden der Geräte bei AWS IoT, dem Aufbauen von Verbindungen und Senden von Nachrichten über die Verbindung sowie zum Bearbeiten von “Geräteschatten”.

4.5 Zusammenfassung

IoT wird ein wichtiger Teil der IT in den nächsten Jahren sein und AWS IoT kann helfen, IoT auch dem Massenmarkt verfügbar zu machen. Im Weg stehen im Moment die

geringe Einsteigerfreundlichkeit der zu AWS IoT kompatiblen Geräte. Direkt kompatibel sind nur einige von Amazon angebotene Geräte und auch dort ist noch eine Menge Konfigurationsarbeit nötig. Die Dokumentation dieser Geräte ist leider bisher auch nicht allzu gut und es lassen sich nur wenige Erfahrensberichte finden. Da AWS IoT Stand Februar 2016 allerdings auch erst einen Monat offiziell nutzbar ist, kann sich dies im Laufe der Zeit noch ändern. Wenn AWS IoT in der Zukunft auch mit weniger Aufwand betreibbar ist, kann es ein wichtiger Teil des IoT Marktes werden.

5. Evaluation

In diesem Kapitel werden die genutzten Amazon Web Services Kinesis und DynamoDB betrachtet und deren Nutzen für ein Projekt dieser Art evaluiert. Zudem wird der potentielle Nutzen von AWS IoT bei Nutzung echter Sensoren evaluiert und noch einige andere Implementierungsmöglichkeiten betrachtet.

5.1 Evaluation Kinesis und DynamoDB

Amazon Kinesis war in diesem Projekt eine sehr wichtige Komponente, die es ermöglichte das Projekt schnell durchzuführen, aber auch gute Ergebnisse erzielen zu können. Kinesis ist leicht nutzbar und bietet einen guten Durchsatz der Daten. So war es ohne Probleme möglich mehrere tausend Temperaturdaten von verschiedenen Sensoren pro Sekunde über Amazon Kinesis zu übertragen und zu verarbeiten. Dabei wurde zur Vereinfachung ein einfacher kommaseparierter String erzeugt, der die Temperaturdaten enthielt, und mittels Amazon Kinesis übertragen. Dieser String wurde vom Consumer verarbeitet und die Temperaturdaten dann manuell in DynamoDB persistiert. Es ist aber auch möglich, eigene Modelklassen für die Daten zu erstellen und mittels Annotationen direkt anzugeben, welches Attribut ein Key Attribut ist usw. und die Modelklasse direkt auf DynamoDB zu persistieren. Für das vergleichsweise einfache Datenmodell dieses Projekts war dies nicht notwendig, aber für komplexere Projekte ist das ein großer Vorteil. Zudem wurde in diesem Projekt ein Kinesis Stream mit nur einem einzigen Shard genutzt. Für größere Projekte können mehr Shards genutzt werden, die den Datendurchsatz noch weiter erhöhen können.
Die Nutzung von DynamoDB war in diesem Projekt ebenfalls recht einfach. Es mussten

zu jedem Temperaturwert ein Hashkey sowie ein Rangekey angegeben werden, die eigentlichen Temperaturwerte konnten dann direkt in diesem Datensatz gespeichert werden. Auch eine Map kann in DynamoDB so direkt und ohne Anpassung des Formats im Gegensatz zu relationalen Datenbanken in diesem Datensatz gespeichert werden . Zudem war DynamoDB in den Testphasen recht performant und konnte mehrere tausend Datensätze pro Sekunde persistieren. Hier war es allerdings notwendig das Datenformat etwas anzupassen und pro Sensordurchlauf nur noch einen Datensatz zu erstellen und die einzelnen Temperaturwerte als Map in diesem Datensatz zu speichern und diesen Datensatz nur noch zu bearbeiten. Wurden alle Datensätze einzeln persistiert, konnte DynamoDB nur noch einige hundert Temperaturen pro Sekunde maximal persistieren. Trotz gutem Datendurchsatz ist ein passendes Datenmodell also dennoch wichtig für eine gute Performance von DynamoDB.
Ein negativer Punkt in Sachen Kinesis ist die permanente Speicherung der Daten innerhalb eines Kinesis Streams. Daten, die über einen Kinesis Stream gesendet werden, werden noch bis zu einer Woche gespeichert, damit sie weiterhin von einem Consumer gelesen werden können. Das ist zunächst einmal ein positiver Punkt, allerdings ist die Speicherung vergleichsweise kostenintensiv. Wurde ein Stream nach Einsatz nicht gelöscht, wurden die übertragenen Daten im Stream persistiert, was schon nach wenigen Tagen Kosten von mehreren Dollar zur Folge hatte. Für dieses Projekt mit dieser Datenmenge kein Problem, da ja auch ein gewisses Budget gewährt worden war, bei größeren Projekten können dadurch aber unter Umständen größere Kosten entstehen, die aber vollkommen unnötig sind. Im Laufe dieses Projekts wurde keine Möglichkeit gefunden, die Persistierung der Daten in Streams schon im Vorhinein zu unterbinden, was das Monitoring der Amazon Web Services und besonders von Amazon Kinesis sehr wichtig machte, um keine unnötigen Kosten zu verursachen.
Positiv zu erwähnen ist, dass die Übertragung der Daten über Kinesis in den Tests immer kostenlos war, da sich die Datenmenge noch im kostenlosen Bereich von Kinesis befand. Um innerhalb dieses Bereichs zu bleiben, wurden die Producer bewusst nur maximal einige Minuten betrieben. Bei dauerhaftem Einsatz würde die Datenmenge auch in den kostenpflichtigen Bereich steigen. Genauso waren auch die Kosten für die Nutzung von DynamoDB und Amazon EC2 im Cent- oder maximal im niedrigen Dollarbereich.
Ein weiterer positiver Punkt ist die recht gute Dokumentation der verschiedenen Webservices von Amazon, die einen schnellen Einstieg ermöglicht. Die Grundlagen aller Services sind gut erklärt und auch die APIs und Client Libraries werden beschrieben. Be-

sonders die Beispielprojekte einzelner Services sind ein guter Einstieg in AWS. Negativ ist, dass die Dokumentation über die Grundlagen meist nicht hinaus geht und im Bereich Amazon Kinesis die Kinesis Connectors nur sehr kurz beschrieben wurden, so dass diese in diesem Projekt nicht genutzt werden konnten, da es aufgrund der geringen Dokumentation zu viel Zeit gebraucht hätte die Funktionsweise dieser Library zu verstehen.
Die Nutzung von Docker war im Rahmen dieses Projektes auch sehr simpel umzusetzen, da es von vornherein als Maven Projekt geplant wurde, für das es ein eigenes Docker Image gibt. Somit war nur wenig Arbeit notwendig, um das Projekt in einem Docker Container zu starten und damit war auch das Deployment auf Amazon EC2 mittels Amazon ECS kein Problem mehr. Mit Amazon ECS lassen sich in Docker Container gepackte Applikationen direkt auf Amazon EC2 deployen, was eine effiziente Möglichkeit für dieses Projekt darstellte.

5.2 Evaluation AWS IoT

AWS IoT ist eine Service von Amazon, der zwar nicht in diesem Projekt verwendet wurde, im Rahmen dieses Themas aber so interessant ist, dass er trotzdem als weitere Alternative betrachtet werden sollte. Es bietet eine Plattform, über die IoT Geräte wie bspw. Sensoren miteinander kommunizieren können sowie AWS IoT auch mit den Geräten kommunizieren kann (s. Kapitel 4).
In diesem Projekt könnte AWS IoT die Rolle von Amazon Kinesis als Kommunikationsweg der Sensoren übernehmen mit dem Unterschied, dass IoT direkt für physikalische Geräte entwickelt wurde. Wenn man also physikalische Geräte einsetzen möchte, könnte AWS IoT die effizientere Lösung sein.
Vorteile sind also die Möglichkeit der direkten Anbindung physikalischer Geräte sowie die simple Kommunikationsmöglichkeit zwischen den Geräten. Zudem ist eine Anbindung an weitere Amazon Services wie bspw. DynamoDB zur Datenhaltung möglich.
Negativ ist vor allem die geringe Anzahl der zu AWS IoT direkt kompatiblen Geräte sowie die bisher recht geringe Dokumentation dieser Geräte. Daher ist ein Einstieg im Moment nicht allzu einfach und mit einiger Konfigurationsarbeit verbunden. Eventuell könnte sich dieser Zustand zukünftig aber noch ändern, vor allem im Hinblick darauf, dass AWS IoT noch nicht allzu lange verfügbar ist.

5.3 Evaluation anderer Implementierungsmöglichkeiten

Bei der Ausführung dieses Projektes wurden noch einige andere Implementierungsmöglichkeiten ins Auge gefasst, die allerdings aus verschiedenen Gründen in diesem Projekt nicht umgesetzt werden konnten.
Dazu zählen zum einen die bereits in Kapitel 5.1 erwähnten Kinesis Connectors. Diese hätten die Projektarchitektur erheblich verändert und den Consumer überflüssig gemacht. Vorteil dieser Variante wäre gewesen, dass die Übermittlung der Daten direkt über Kinesis an DynamoDB gelaufen wäre ohne den Umweg über den Consumer, was die Aktualität der Daten näher an Echtzeit herangebracht hätte, welches eine der geplanten Anforderungen darstellt.
Eine weitere Möglichkeit wäre die Nutzung anderer Datenbanksysteme gewesen. Amazon bietet neben DynamoDB zudem noch Amazon RDS an. Eine Lösung mit RDS wäre ebenfalls möglich gewesen, da das Datenmodell so simpel ist, dass auch relationale Datenbanken voraussichtlich eine ähnliche Performance hätten liefern können. Eine eigene Lösung bspw. mit Apache Cassandra wurde ebenfalls erörtert, letztendlich aufgrund von Zeitmangel aber verworfen. Es wurde auf die Lösung mit DynamoDB gesetzt, um eine weitere neue Technologie kennen zu lernen und weil diese Technologie gute Performance liefert und dank guter Dokumentation einen leichten Einstieg bot.
Eine Anforderung an das Projekt war die Nutzer mehrerer AWS Services, um die unterschiedlichen AWS Services im Rahmen des Projektes evaluieren zu können, im Besonderen bei der Verwendung von großen Datenströmen. Hier hätte sich beispielsweise eine eigene Lösung mit Amazon EC2 angeboten, die allerdings für dieses Projekt zu komplex gewesen wäre. Es hätten die Funktionen von Amazon Kinesis selbst umgesetzt werden können, was bei der Zuverlässigkeit von Amazon Kinesis innerhalb des Projekts aber auch nicht notwendig ist.

5.4 Zusammenfassung

Insgesamt kann also gesagt werden, dass die genutzten Amazon Web Services für große Datenströme gut geeignet sind. Im Test konnten mehrere tausend Temperaturdaten innerhalb einer Sekunde übermittelt werden und es kann immer noch nach oben skaliert werden für einen wesentlich höheren Datendurchsatz. Für dieses Projekt war die Dokumentation der AWS Services recht gut, könnte in einigen Punkten aber noch

etwas weiter gehen.
In weiteren Tests könnte noch überprüft werden, ob der Durchsatz linear nach oben skalierbar ist und auch Millionen von Temperaturdaten pro Sekunde verschickt werden könnten. Dies konnte aufgrund des begrenzten Budgets im Rahmen dieser Projektarbeit nicht durchgeführt werden.

6. Fazit

Das Projekt verlief insgesamt recht gut und es konnten alle entscheidenden Anforderungen an das Projekt umgesetzt werden. Die Evaluation der genutzten Amazon Web Services zeigte, dass sie für die Verarbeitung größerer Datenströme geeignet sind und auch bei größerem Datendurchsatz eine gute Performance bieten. Dies bezieht sich besonders auf AWS Kinesis, das die Übertragung der Temperaturdaten stark vereinfachte. Zudem zeigte sich auch eine gute Einsteigerfreundlichkeit bei der Nutzung der Client SDKs durch eine gute Dokumentation, die allerdings bei weiterführenden Themen nicht immer so umfangreich war. Daher war es im Rahmen dieses Projektes leider nicht möglich, einige Technologien zu nutzen, die weitere Erkenntnisse zu AWS hätten bringen können.
Positiv ist zudem die Projektplanung und insbesondere die Risikoplanung zu nennen, durch die einige Risiken abgeschwächt werden konnten und das Projektziel somit nicht gefährdet wurde. Auch das Budget konnte dank der dauerhaften Überwachung der Kosten eingehalten werden, obwohl einige unerwartete Kosten anfielen.
Die Integration des Projektes in ein Docker Image war dank der vorhandenen passenden Basisimages wie dem hier genutzten Maven Image ebenfalls sehr einfach und konnte gut umgesetzt werden.
Zudem wurde in AWS IoT ein neuer Amazon Web Service evaluiert, der für die Nutzung von echten Sensoren konzipiert wurde und damit auch einen Blick in den potentiellen IoT-Markt der Zukunft ermöglicht. Hier kann ein generell positives Fazit gezogen werden, da AWS IoT einige Funktionen bietet, die die Kommunikation mit IoT-Geräten vereinfacht und die Integration von anderen Amazon Web Services zur Datenhaltung oder Verarbeitung der Daten ermöglicht.
Zukünftig könnte dieses Projekt entsprechend der bisher nicht umgesetzten Anforde-

rungen erweitert werden oder die bisher umgesetzten Klassen könnten anders implementiert werden, um weitere Funktionen von Amazon Web Services zu testen.

Abkürzungsverzeichnis

AWS	Amazon Web Services
EC2	Amazon Elastic Compute Cloud
ECS	Amazon EC2 Container Service
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
RDS	Amazon Relational Database Service
REST	Representational State Transfer
S3	Amazon Simple Storage Service

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

Literaturverzeichnis

[1] Hendrik Hagmans, “Projektarbeit Evaluierung einer Amazon Web Services LÃ¶sung zur Erfassung und Verarbeitung von Sensordaten”; im Rahmen des WS 15 an der FH Dortmund, abgerufen am 28. Oktober 2015

[2] Daniele Miorandi, Sabrina Sicari, Francesco De Pellegrini, Imrich Chlamtac, “Internet of things: Vision, applications and research challenges”; https://irinsubria.uninsubria.it/retrieve/handle/11383/1762288/2389/IOT.pdf, abgerufen am 28. Oktober 2015

[3] Amazon Inc, “Amazon Web Services”; http://aws.amazon.com/de/, abgerufen am 28. Oktober 2015

[4] Docker Inc, “Docker”; https://www.docker.com/, abgerufen am 28. Oktober 2015

[5] Amazon Inc, “Amazon ECS”; http://aws.amazon.com/de/ecs/, abgerufen am 28. Oktober 2015

[6] Amazon Inc, “Amazon EC2”; http://aws.amazon.com/de/ec2/, abgerufen am 28. Oktober 2015

[7] Amazon Inc, “Amazon RDS”; http://aws.amazon.com/de/rds/, abgerufen am 28. Oktober 2015

[8] Amazon Inc, “Amazon Dynamo DB”; http://aws.amazon.com/de/dynamodb/, abgerufen am 28. Oktober 2015

[9] Amazon Inc, “Amazon Kinesis”; http://aws.amazon.com/de/kinesis/, abgerufen am 28. Oktober 2015

[10] Amazon Inc, “Amazon Kinesis Connector Library”; https://github.com/awslabs/amazon-kinesis-connectors, abgerufen am 28. Oktober 2015

[11] Amazon Inc, “Amazon Kinesis Firehose”; https://aws.amazon.com/de/firehose/, abgerufen am 28. Oktober 2015

[12] Amazon Inc, “Amazon S3”; https://aws.amazon.com/de/s3/, abgerufen am 28. Oktober 2015

[13] Amazon Inc, “Amazon Kinesis Data Visualization Sample Application”; https://github.com/awslabs/amazon-kinesis-data-visualization-sample, abgerufen am 28. Oktober 2015

[14] Amazon Inc, “KPL Java Sample Application”; https://github.com/awslabs/amazon-kinesis-producer/tree/master/java/amazon-kinesis-producer-sample, abgerufen am 28. Oktober 2015

[15] Amazon Inc, “AWS Lambda”; http://aws.amazon.com/de/lambda/, abgerufen am 28. Oktober 2015

[16] Amazon Inc, “AWS IoT”; http://aws.amazon.com/de/iot/, abgerufen am 28. Oktober 2015

[17] Amazon Inc, “AWS IoT Funktionsweise”; https://aws.amazon.com/de/iot/how-it-works/, abgerufen am 28. Oktober 2015

[18] Amazon Inc, “AWS IoT Thing Registry”; http://docs.aws.amazon.com/iot/latest/developerguide/thing-registry.html, abgerufen am 28. Oktober 2015

[19] Amazon Inc, “AWS IoT Creating an AWS IoT Rule”; http://docs.aws.amazon.com/iot/latest/developerguide/aws-iot-create-rule.html, abgerufen am 28. Oktober 2015

[20] Amazon Inc, “AWS IoT GetThingShadow”; http://docs.aws.amazon.com/iot/latest/developerguide/API_GetThingShadow.html, abgerufen am 28. Oktober 2015

[21] Amazon Inc, “AWS IoT SDK”; http://docs.aws.amazon.com/iot/latest/developerguide/iot-sdks.html, abgerufen am 28. Oktober 2015

[22] MQTT Org, “MQTT”; http://mqtt.org/, abgerufen am 28. Oktober 2015

[23] Raspberry Pi Foundation, “Raspberry PI”; https://www.raspberrypi.org/, abgerufen am 28. Oktober 2015

[24] Apache Software Foundation, “Apache Cassandra”; http://cassandra.apache.org/, abgerufen am 28. Oktober 2015

[25] Apache Software Foundation, “Apache Maven”; https://maven.apache.org/, abgerufen am 28. Oktober 2015

[26] Apache Software Foundation, “Apache Maven Image”; https://hub.docker.com/_/maven/, abgerufen am 28. Oktober 2015

[27] Eclipse Foundation, “Eclipse Jetty”; http://www.eclipse.org/jetty/, abgerufen am 28. Oktober 2015

[28] Node.js Foundation, “Node.js”; https://nodejs.org/en/, abgerufen am 28. Oktober 2015

[29] fenopix, “CanvasJS”; http://canvasjs.com/, abgerufen am 28. Oktober 2015

[30] Arduino Foundation, “Arduino YÃºn”; https://www.arduino.cc/en/Main/ArduinoBoardYun, abgerufen am 28. Oktober 2015

[31] Amazon Inc, “Amazon Kinesis Key Concepts”; http://docs.aws.amazon.com/kinesis/latest/dev/key-concepts.html, abgerufen am 28. Oktober 2015

Erklärung der Urheberschaft

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne die Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung noch nicht vorgelegt worden.

Ort, Datum	Unterschrift